

O'REILLY®

Fourth
Edition

Head First

C#

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

Andrew Stellman
& Jennifer Greene

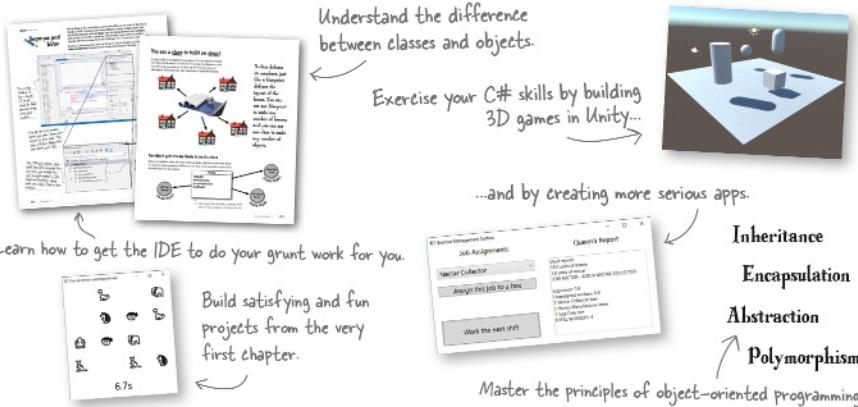


A Brain-Friendly Guide

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!



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.

.NET

US \$64.99

CAN \$85.99

ISBN: 978-1-491-97670-8



5 6 4 9 9
9 781491 976708

"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

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**

Related books from O'Reilly

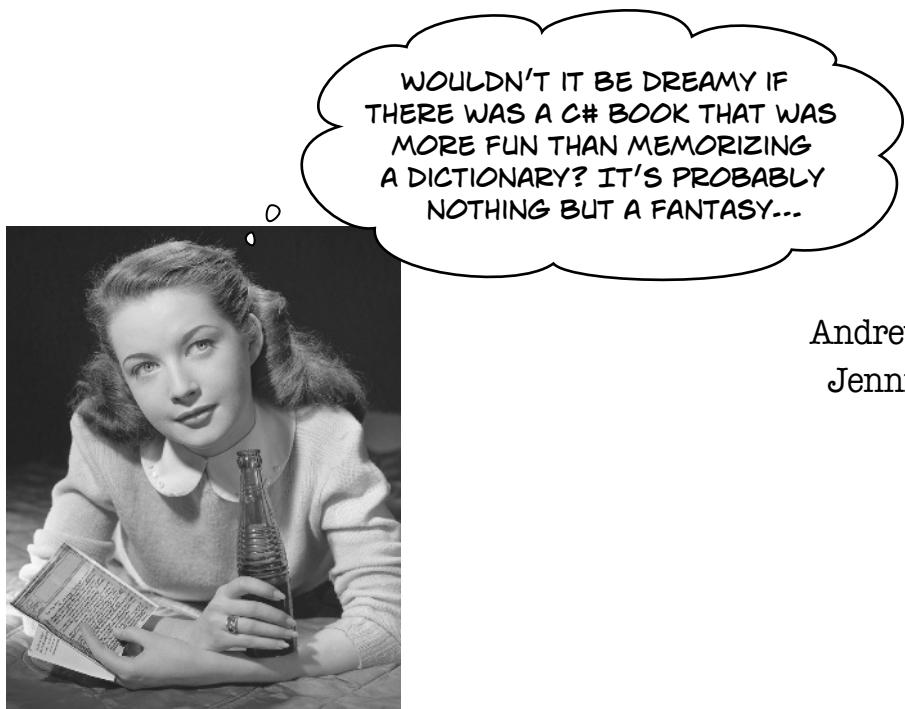
C# 8.0 in a Nutshell
C# 8.0 Pocket Reference
C# Database Basics
C# Essentials, 2nd Edition
Concurrency in C# Cookbook, 2nd Edition
Mobile Development with C#
Programming C# 8.0

Other books in O'Reilly's *Head First* series

Head First 2D Geometry	Head First Networking
Head First Agile	Head First Object-Oriented Analysis and Design
Head First Ajax	Head First PHP & MySQL
Head First Algebra	Head First Physics
Head First Android Development	Head First PMP
Head First C	Head First Programming
Head First Data Analysis	Head First Python
Head First Design Patterns	Head First Rails
Head First EJB	Head First Ruby
Head First Excel	Head First Ruby on Rails
Head First Go	Head First Servlets and JSP
Head First HTML5 Programming	Head First Software Development
Head First HTML with CSS and XHTML	Head First SQL
Head First iPhone and iPad Development	Head First Statistics
Head First Java	Head First Web Design
Head First JavaScript Programming	Head First WordPress
Head First Kotlin	
Head First jQuery	
Head First Learn to Code	
Head First Mobile Web	

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

Copyright © 2021 Jennifer Greene, Andrew Stellman. All rights reserved.

Printed in the United States of America.

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



The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First C#*, and related trade dress are trademarks of O'Reilly Media, Inc.

Microsoft, Windows, Visual Studio, MSDN, the .NET logo, Visual Basic, and Visual C# are registered trademarks of Microsoft Corporation.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

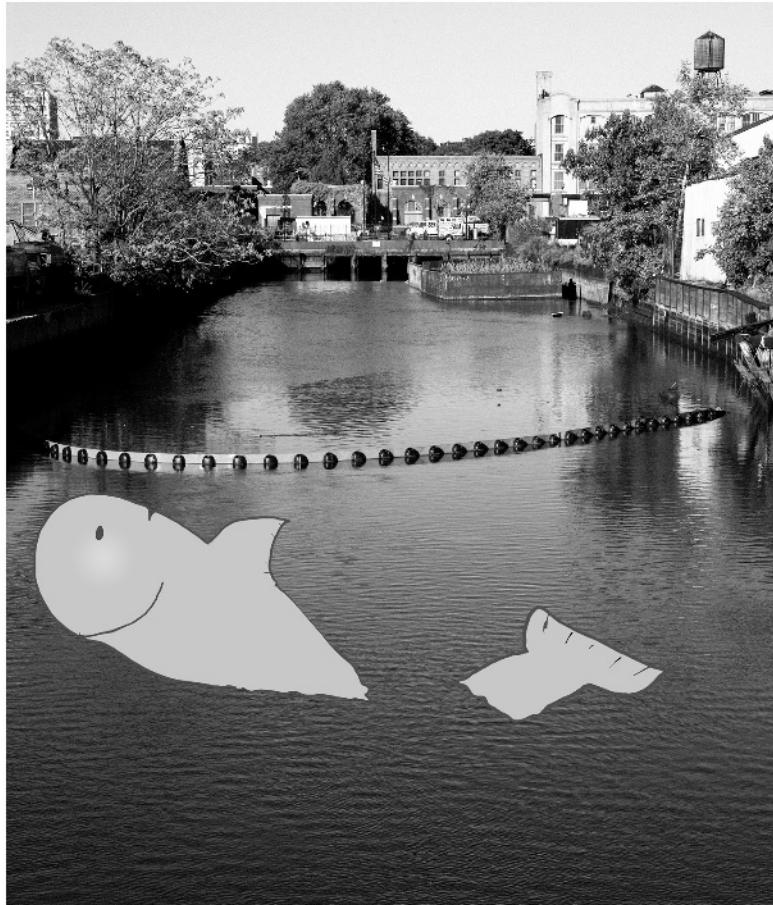
No bees, space aliens, or comic book heroes were harmed in the making of this book.

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.*



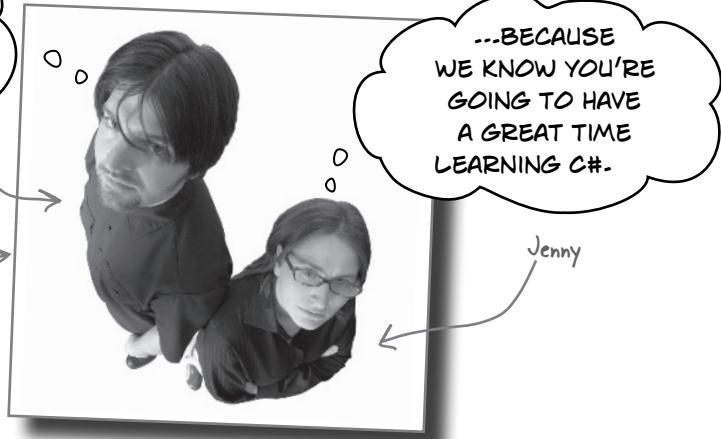
*You were only in our canal for a day,
but you'll be in our hearts forever.*

the authors

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

Andrew

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.

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.

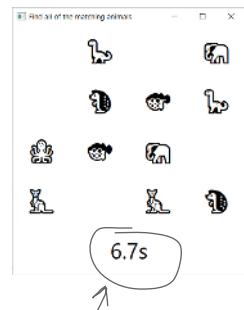
They founded Stellman & Greene Consulting in 2003 to build a really neat software project for scientists studying herbicide exposure in Vietnam vets. In addition to building software and writing books, they've consulted for companies and spoken at conferences and meetings of software engineers, architects,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

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

Follow @AndrewStellman and @JennyGreene on Twitter

Peace, Love, and 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.

Intro	xxxii
1 Start building with C#: <i>Building something great...fast!</i>	1
2 Dive into C#: <i>Statements, classes, and code</i>	49
<i>Unity Lab 1: Explore C# with Unity</i>	87
3 Objects...get oriented: <i>Making code make sense</i>	117
4 Types and references: <i>Getting the reference</i>	155
<i>Unity Lab 2: Write C# Code for Unity</i>	213
5 Encapsulation: <i>Keep your privates...private</i>	227
6 Inheritance: <i>Your object's family tree</i>	273
<i>Unity Lab 3: GameObject Instances</i>	343
7 Interfaces, casting, and "is": <i>Making classes keep their promises</i>	355
8 Enums and collections: <i>Organizing your data</i>	405
<i>Unity Lab 4: User Interfaces</i>	453
9 LINQ and lambdas: <i>Get control of your data</i>	467
10 Reading and writing files: <i>Save the last byte for me!</i>	529
<i>Unity Lab 5: Raycasting</i>	577
11 Captain Amazing: <i>The Death of the Object</i>	587
12 Exception handling: <i>Putting out fires gets old</i>	623
<i>Unity Lab 6: Scene Navigation</i>	651
Downloadable exercise: Animal match boss battle	661
i Visual Studio for Mac Learner's Guide	663
ii Code Kata: <i>A learning guide for advanced and impatient readers</i>	725



Table of Contents (the real thing)

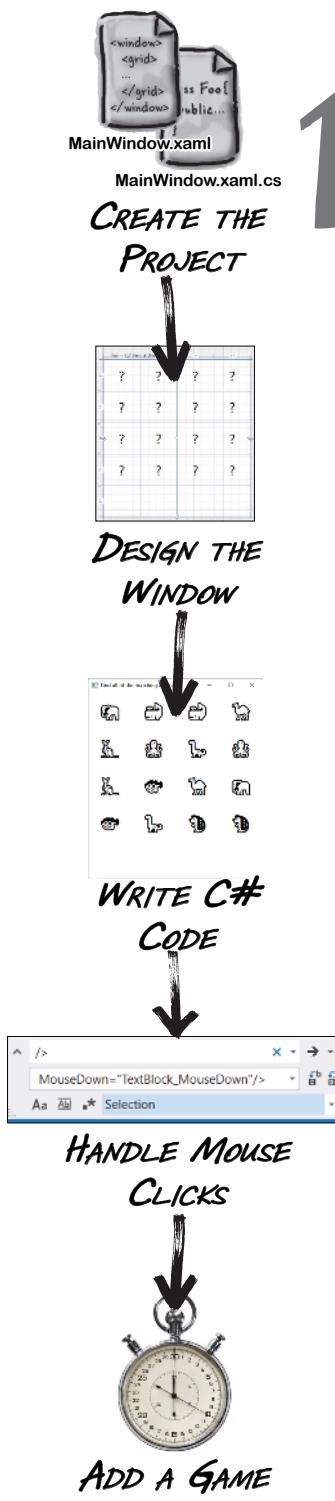
Intro

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#?



Who is this book for?	xxx
We know what you're thinking.	xxxii
And we know what your brain is thinking.	xxxii
Metacognition: thinking about thinking	xxxiii
Here's what WE did	xxxiv
Here's what YOU can do to bend your brain into submission	xxxv
README	xxxvi
The technical review team	xli
Acknowledgments	xli
O'Reilly online learning	xlii





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#	2
Visual Studio is a tool for writing code and exploring C#	3
Create your first project in Visual Studio	4
Let's build a game!	6
Here's how you'll build your game	7
Create a WPF project in Visual Studio	8
Use XAML to design your window	12
Design the window for your game	13
Set the window size and title with XAML properties	14
Add rows and columns to the XAML grid	16
Make the rows and columns equal size	17
Add a TextBlock control to your grid	18
Now you're ready to start writing code for your game	21
Generate a method to set up the game	22
Finish your SetUpGame method	24
Run your program	26
Add your new project to source control	30
The next step to build the game is handling mouse clicks	33
Make your TextBlocks respond to mouse clicks	34
Add the TextBlock_MouseDown code	37
Make the rest of the TextBlocks call the same MouseDown event handler	38
Finish the game by adding a timer	39
Add a timer to your game's code	40
Use the debugger to troubleshoot the exception	42
Add the rest of the code and finish the game	46
Update your code in source control	47

drive into C#

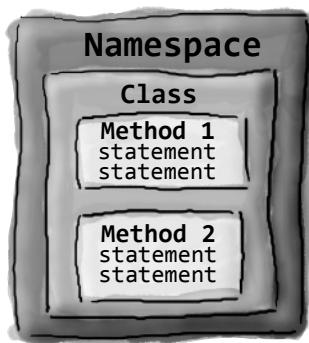
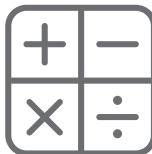
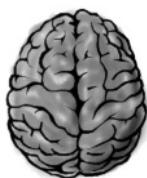
2

Statements, classes, and code

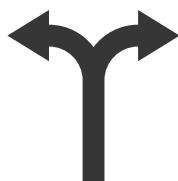
You're not just an IDE user. You're a developer.

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.



Let's take a closer look at the files for a console app	50
Two classes can be in the same namespace (and file!)	52
Statements are the building blocks for your apps	55
Your programs use variables to work with data	56
Generate a new method to work with variables	58
Add code that uses operators to your method	59
Use the debugger to watch your variables change	60
Use operators to work with variables	62
“if” statements make decisions	63
Loops perform an action over and over	64
Use code snippets to help write loops	67
Controls drive the mechanics of your user interfaces	71
Create a WPF app to experiment with controls	72
Add a TextBox control to your app	75
Add C# code to update the TextBlock	78
Add an event handler that only allows number input	79
Add sliders to the bottom row of the grid	83
Add C# code to make the rest of the controls work	84

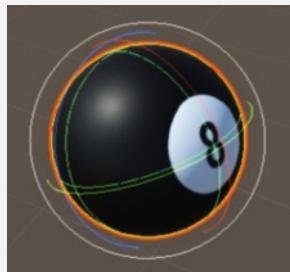
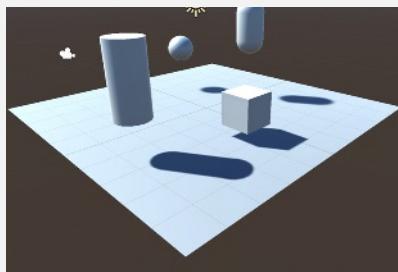


Unity Lab 1

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.

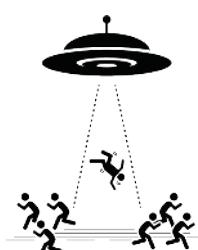
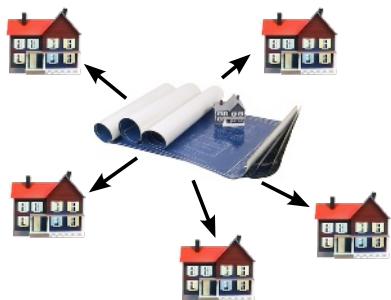
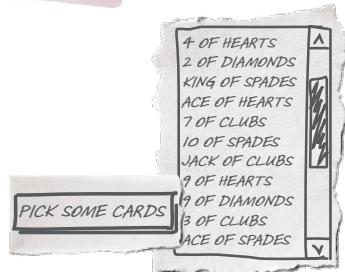
Unity is a powerful tool for game design	88
Download Unity Hub	89
Use Unity Hub to create a new project	90
Take control of the Unity layout	91
Your scene is a 3D environment	92
Unity games are made with GameObjects	93
Use the Move Gizmo to move your GameObjects	94
The Inspector shows your GameObject's components	95
Add a material to your Sphere GameObject	96
Rotate your sphere	99
Get creative!	102



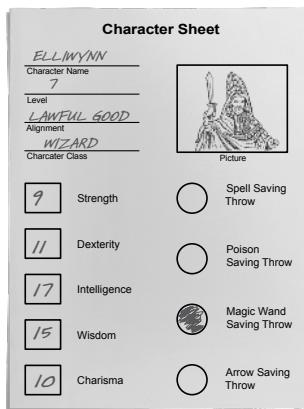
objects...get oriented!

Making code make sense

3



If code is useful, it gets reused	104
Some methods take parameters and return a value	105
Let's build a program that picks some cards	106
Create your PickRandomCards console app	107
Finish your PickSomeCards method	108
Your finished CardPicker class	110
Ana's working on her next game	113
Build a paper prototype for a classic game	116
Up next: build a WPF version of your card picking app	118
A StackPanel is a container that stacks other controls	119
Reuse your CardPicker class in a new WPF app	120
Use a Grid and StackPanel to lay out the main window	121
Lay out your Card Picker desktop app's window	122
Ana can use objects to solve her problem	126
You use a class to build an object	127
When you create a new object from a class, it's called an instance of that class	128
A better solution for Ana... brought to you by objects	129
An instance uses fields to keep track of things	133
Thanks for the memory	136
What's on your program's mind	137
Sometimes code can be difficult to read	138
Use intuitive class and method names	140
Build a class to work with some guys	146
There's an easier way to initialize objects with C#	148
Use the C# Interactive window to run C# code	154



**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.**



types and references

Getting the reference

4

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!	156
Character sheets store different types of data on paper	157
A variable's type determines what kind of data it can store	158
C# has several types for storing integers	159
Let's talk about strings	161
A literal is a value written directly into your code	162
A variable is like a data to-go cup	165
Other types come in different sizes, too	166
10 pounds of data in a 5-pound bag	167
Casting lets you copy values that C# can't automatically convert to another type	168
C# does some conversion automatically	171
When you call a method, the arguments need to be compatible with the types of the parameters	172
Let's help Owen experiment with ability scores	176
Use the C# compiler to find the problematic line of code	178
Use reference variables to access your objects	186
Multiple references and their side effects	190
Objects use references to talk to each other	198
Arrays hold multiple values	200
Arrays can contain reference variables	201
null means a reference points to nothing	203
Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!	208

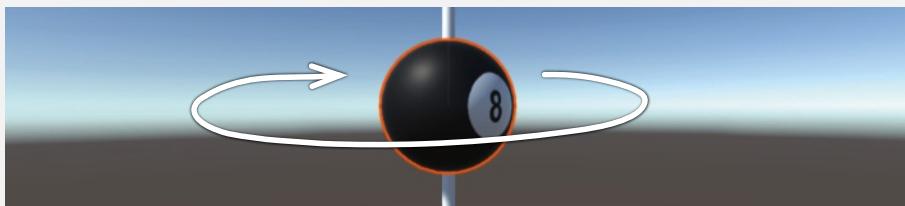


Unity Lab 2

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.

C# scripts add behavior to your GameObjects	214
Add a C# script to your GameObject	215
Write C# code to rotate your sphere	216
Add a breakpoint and debug your game	218
Use the debugger to understand Time.deltaTime	219
Add a cylinder to show where the Y axis is	220
Add fields to your class for the rotation angle and speed	221
Use Debug.DrawRay to explore how 3D vectors work	222
Run the game to see the ray in the Scene view	223
Rotate your ball around a point in the scene	224
Use Unity to take a closer look at rotation and vectors	225
Get creative!	226





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.

SwordDamage	
Roll	Let's help Owen roll for damage
MagicMultiplier	228
FlamingDamage	Create a console app to calculate damage
Damage	229
CalculateDamage	Design the XAML for a WPF version of the damage calculator
SetMagic	231
SetFlaming	The code-behind for the WPF damage calculator
	232
	Tabletop talk (or maybe...dice discussion?)
	233
	Let's try to fix that bug
	234
	Use Debug.WriteLine to print diagnostic information
	235
	It's easy to accidentally misuse your objects
	238
	Encapsulation means keeping some of the data in a class private
	239
	Use encapsulation to control access to your class's methods and fields
	240
	But is the RealName field REALLY protected?
	241
	Private fields and methods can only be accessed from instances of the same class
	242
	Why encapsulation? Think of an object as a black box...
	247
	Let's use encapsulation to improve the SwordDamage class
	251
	Encapsulation keeps your data safe
	252
	Write a console app to test the PaintballGun class
	253
	Properties make encapsulation easier
	254
	Modify your Main method to use the Bullets property
	255
	Auto-implemented properties simplify your code
	256
	Use a private setter to create a read-only property
	257
	What if we want to change the magazine size?
	258
	Use a constructor with parameters to initialize properties
	259
	Specify arguments when you use the new keyword
	260



RealName: "Herb Jones"

Alias: "Dash Martin"

Password: "the crow flies at midnight"



inheritance

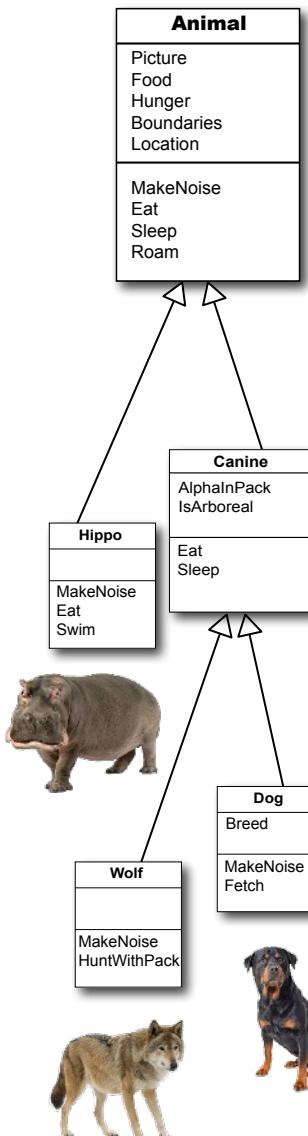
Your object's family tree

6

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**.



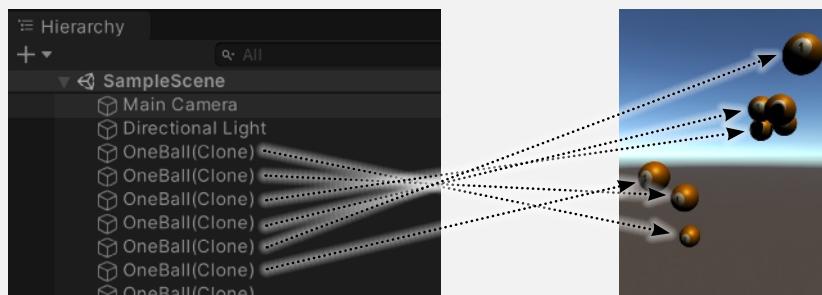
Calculate damage for MORE weapons	274
Use a switch statement to match several candidates	275
One more thing...can we calculate damage for a dagger? and a mace?	277
When your classes use inheritance, you only need to write your code once	278
Build up your class model by starting general and getting more specific	279
How would you design a zoo simulator?	280
Every subclass extends its base class	285
Use a colon to extend a base class	290
A subclass can override methods to change or replace members it inherited	292
Some members are only implemented in a subclass	297
Use the debugger to understand how overriding works	298
Build an app to explore virtual and override	300
A subclass can hide methods in the base class	302
Use the override and virtual keywords to inherit behavior	304
When a base class has a constructor, your subclass needs to call it	307
It's time to finish the job for Owen	309
Build a beehive management system	316
The Queen class: how she manages the worker bees	318
The UI: add the XAML for the main window	319
Feedback drives your Beehive Management game	328
Some classes should never be instantiated	332
An abstract class is an intentionally incomplete class	334
Like we said, some classes should never be instantiated	336
An abstract method doesn't have a body	337
Abstract properties work just like abstract methods	338

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.

Let's build a game in Unity!	344
Create a new material inside the Materials folder	345
Spawn a billiard ball at a random point in the scene	346
Use the debugger to understand Random.value	347
Turn your GameObject into a prefab	348
Create a script to control the game	349
Attach the script to the Main Camera	350
Press Play to run your code	351
Use the Inspector to work with GameObject instances	352
Use physics to keep balls from overlapping	353
Get creative!	354



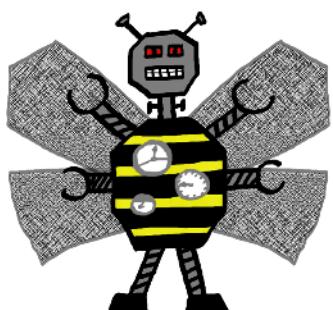
interfaces, casting, and “is”

Making classes keep their promises

7

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?



The beehive is under attack!	356
We can use casting to call the DefendHive method...	357
An interface defines methods and properties that a class must implement...	358
Get a little practice using interfaces	360
You can’t instantiate an interface, but you can reference an interface	366
Interface references are ordinary object references	369
The RoboBee 4000 can do a worker bee’s job without using valuable honey	370
The IWorker’s Job property is a hack	374
Use “is” to check the type of an object	375
Use “is” to access methods in a subclass	376
What if we want different animals to swim or hunt in packs?	378
Use interfaces to work with classes that do the same job	379
Safely navigate your class hierarchy with “is”	380
C# has another tool for safe type conversion: the “as” keyword	381
Use upcasting and downcasting to move up and down a class hierarchy	382
Upcasting turns your CoffeeMaker into an Appliance	384
Upcasting and downcasting work with interfaces, too	386
Downcasting turns your Appliance back into a CoffeeMaker	385
Interfaces can inherit from other interfaces	388
Interfaces can have static members	395
Default implementations give bodies to interface methods	396
Add a ScareAdults method with a default implementation	397
Data binding updates WPF controls automatically	399
Modify the Beehive Management System to use data binding	400
Polymorphism means that one object can take many different forms	403



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.

Strings don't always work for storing categories of data	406
Enums let you work with a set of valid values	407
Enums let you represent numbers with names	408
We could use an array to create a deck of cards...	411
Lists make it easy to store collections of...anything	413
Lists are more flexible than arrays	414
Let's build an app to store shoes	417
Generic collections can store any type	420
Collection initializers are similar to object initializers	426
Let's create a List of Ducks	427
Lists are easy, but SORTING can be tricky	428
IComparable<Duck> helps your list sort its ducks	429
Use IComparer to tell your List how to sort	430
Create an instance of your comparer object	431
Overriding a ToString method lets an object describe itself	435
Update your foreach loops to let your Ducks and Cards write themselves to the console	436
You can upcast an entire list using IEnumerable<T>	440
Use a Dictionary to store keys and values	442
The Dictionary functionality rundown	443
Build a program that uses a dictionary	444
And yet MORE collection types...	445
A queue is FIFO—first in, first out	446
A stack is LIFO—last in, first out	447
Downloadable exercise: Two Decks	452

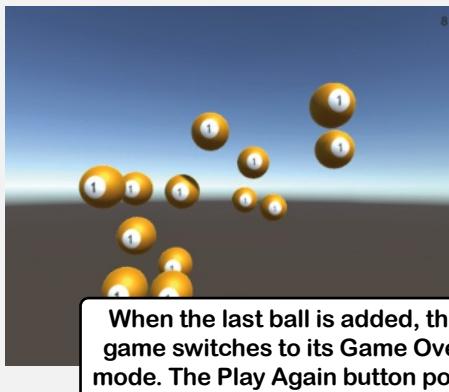
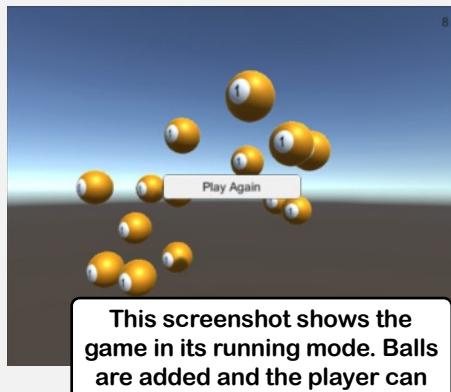


Unity Lab 4

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.

Add a score that goes up when the player clicks a ball	454
Add two different modes to your game	455
Add game mode to your game	456
Add a UI to your game	458
Set up the Text that will display the score in the UI	459
Add a button that calls a method to start the game	460
Make the Play Again button and Score Text work	461
Finish the code for the game	462
Get creative!	466



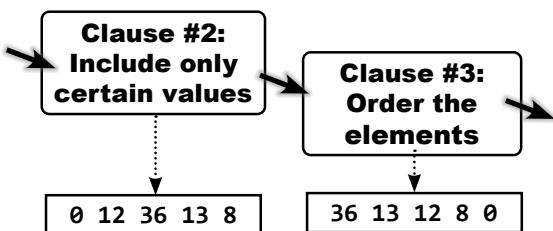
LINQ and lambdas

Get control of your data

9

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.



Jimmy's a Captain Amazing super-fan...	468
Use LINQ to query your collections	470
LINQ works with any IEnumerable<T>	472
LINQ's query syntax	475
LINQ works with objects	477
Use a LINQ query to finish the app for Jimmy	478
The var keyword lets C# figure out variable types for you	480
LINQ queries aren't run until you access their results	487
Use a group query to separate your sequence into groups	488
Use join queries to merge data from two sequences	491
Use the new keyword to create anonymous types	492
Add a unit test project to Jimmy's comic collection app	502
Write your first unit test	503
Write a unit test for the GetReviews method	505
Write unit tests to handle edge cases and weird data	506
Use the => operator to create lambda expressions	508
A lambda test drive	509
Refactor a clown with lambdas	510
Use the ?: operator to make your lambdas make choices	513
Lambda expressions and LINQ	514
LINQ queries can be written as chained LINQ methods	515
Use the => operator to create switch expressions	517
Explore the Enumerable class	521
Create an enumerable sequence by hand	522
Use yield return to create your own sequences	523
Use yield return to refactor ManualSportSequence	524
Downloadable exercise: Go Fish!	528

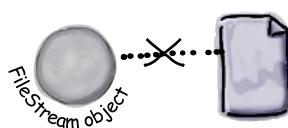
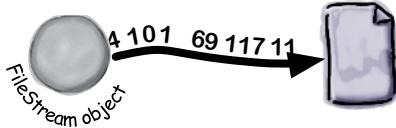
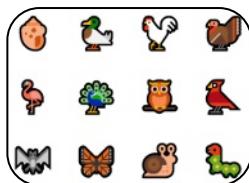
10

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**.



.NET uses streams to read and write data	530
Different streams read and write different things	531
A FileStream reads and writes bytes in a file	532
Write text to a file in three simple steps	533
The Swindler launches another diabolical plan	534
Use a StreamReader to read a file	537
Data can go through more than one stream	538
Use the static File and Directory classes to work with files and directories	542
IDisposable makes sure objects are closed properly	545
Use a MemoryStream to stream data to memory	547
What happens to an object when it's serialized?	553
But what exactly IS an object's state? What needs to be saved?	554
Use JsonSerializer to serialize your objects	556
JSON only includes data, not specific C# types	559
Next up: we'll take a deep dive into our data	561
C# strings are encoded with Unicode	563
Visual Studio works really well with Unicode	565
.NET uses Unicode to store characters and text	566
C# can use byte arrays to move data around	568
Use a BinaryWriter to write binary data	569
Use BinaryReader to read the data back in	570
A hex dump lets you see the bytes in your files	572
Use Stream.Read to read bytes from a stream	574
Modify your hex dumper to use command-line arguments	575
Downloadable exercise: Hide and Seek	576

Eureka! →

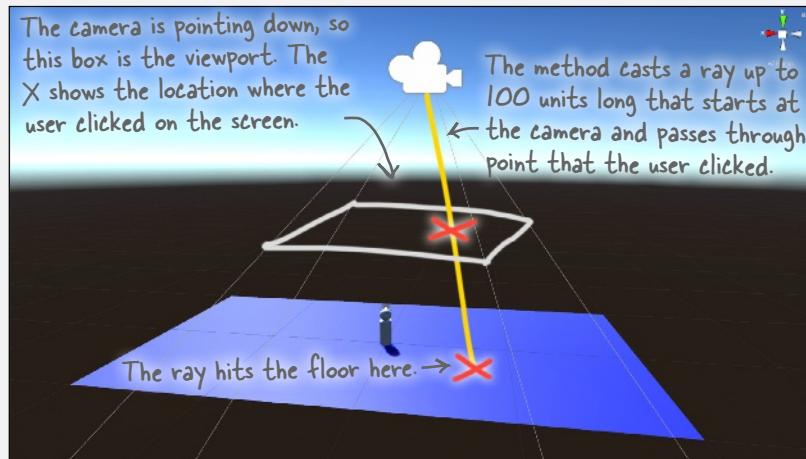


Unity Lab 5

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.

Create a new Unity project and start to set up the scene	578
Set up the camera	579
Create a GameObject for the player	580
Introducing Unity's navigation system	581
Set up the NavMesh	582
Make your player automatically navigate the play area	583



CAPTAIN AMAZING

THE DEATH OF THE OBJECT

Head First C#	
Four bucks	Chapter 11



The life and death of an object	590
Use the GC class (with caution) to force garbage collection	591
Your last chance to DO something...your object's finalizer	592
When EXACTLY does a finalizer run?	593
Finalizers can't depend on other objects	595
A struct looks like an object...	599
Values get copied; references get assigned	600
Structs are value types; objects are reference types	601
The stack vs. the heap: more on memory	603
Use out parameters to make a method return more than one value	606
Pass by reference using the ref modifier	607
Use optional parameters to set default values	608
A null reference doesn't refer to any object	609
Non-nullable reference types help you avoid NREs	610
The null-coalescing operator ?? helps with nulls	611
Nullable value types can be null...and handled safely	612
“Captain” Amazing...not so much	613
Extension methods add new behavior to EXISTING classes	617
Extending a fundamental type: string	619



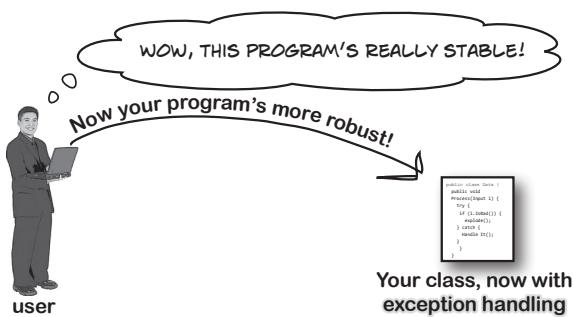
12

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.



Your class, now with exception handling



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



Your hex dumper reads a filename from the command line	624
When your program throws an exception, the CLR generates an Exception object	628
All Exception objects inherit from System.Exception	629
There are some files you just can't dump	632
What happens when a method you want to call is risky?	633
Handle exceptions with try and catch	634
Use the debugger to follow the try/catch flow	635
If you have code that ALWAYS needs to run, use a finally block	636
Catch-all exceptions handle System.Exception	637
Use the right exception for the situation	642
Exception filters help you create precise handlers	646
The worst catch block EVER: catch-all plus comments	648
Temporary solutions are OK (temporarily)	649

Watch 1	
Name	Value
Exception	[System.IndexOutOfRangeException] ("Index was outside the bounds of the array.")
Data	null
HelpLink	-2146233080
HResult	null
Inheritance	"Index was outside the bounds of the array."
Message	"ConsoleApplication1"
Source	" at ConsoleApplication1.Program.Main(String[])"
StackTrace	(Void Main(System.String[]))
TargetSite	
Static members	
Non-Public members	

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#.

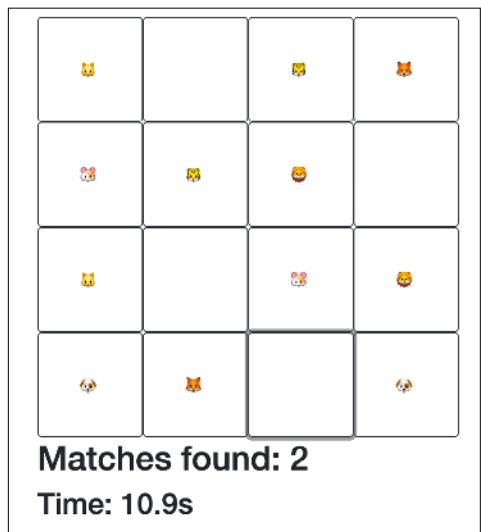
Let's pick up where the last Unity Lab left off	652
Add a platform to your scene	653
Use bake options to make the platform walkable	654
Include the stairs and ramp in your NavMesh	655
Fix height problems in the NavMesh	657
Add a NavMesh Obstacle	658
Add a script to move the obstacle up and down	659
Get creative!	660



appendix i: ASP.NET Core Blazor projects

Visual Studio for Mac Learner's Guide

i



Why you should learn C#	664
Create your first project in Visual Studio for Mac	666
Let's build a game!	670
Create a Blazor WebAssembly App in Visual Studio	672
Run your Blazor web app in a browser	674
Start writing code for your game	676
Finish creating your emoji list and display it in the app	680
Shuffle the animals so they're in a random order	682
You're running your game in the debugger	684
Add your new project to source control	688
Add C# code to handle mouse clicks	689
Add click event handlers to your buttons	690
Test your event handler	692
Use the debugger to troubleshoot the problem	693
Track down the bug that's causing the problem	696
Add code to reset the game when the player wins	698
Add a timer to your game's code	702
Clean up the navigation menu	704
Controls drive the mechanics of your user interfaces	706
Create a new Blazor WebAssembly App project	707
Add a page with a slider control	708
Add text input to your app	710
Add color and date pickers to your app	713
Build a Blazor version of your card picking game	714
The page is laid out with rows and columns	716
The slider uses data binding to update a variable	717
Welcome to Sloppy Joe's Budget House o'Discount Sandwiches!	720

ii

appendix ii: Code Kata

A learning guide for advanced and impatient readers



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.

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.

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.

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.

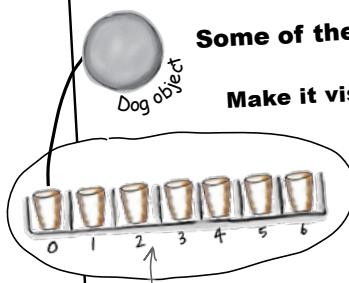
Just one problem. Your brain’s trying to do you a big favor. It’s trying to make sure that this *obviously* unimportant content doesn’t clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted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



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:

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.

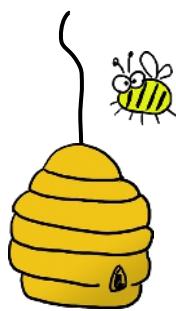
Put the words within or near the graphics they relate to, rather than at the bottom or on another page, and learners will be up to twice as likely to be able to solve problems related to the content.

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.



Even scary emotions
can help ideas stick in
your brain.



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.

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.

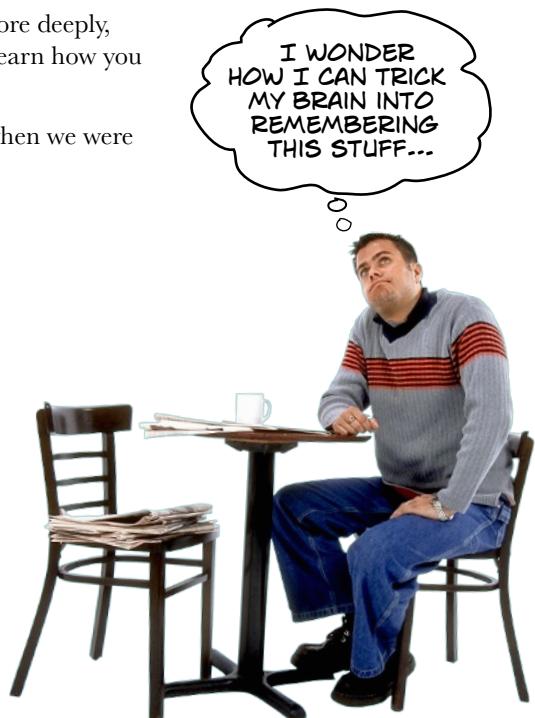
So just how **DO** you get your brain to treat C# like it was a hungry tiger?

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.

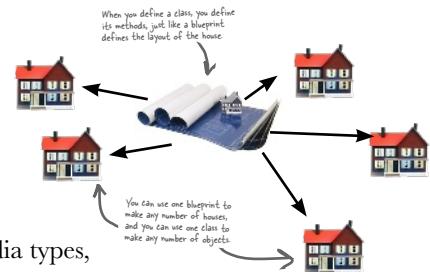
A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the 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.



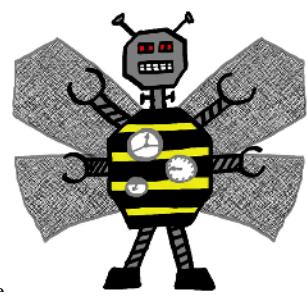
Here's what WE did

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.

BULLET POINTS

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 the same content represented in multiple ways.

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*.

Fireside Chats





Cut this out and stick it
on your refrigerator.

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.

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.

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.

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.

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>.

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



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



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



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.



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



Lisa Kellner



Lindsey Bieda



Tatiana Mac

Ashley Godbold

Not pictured (but just as amazing) are the reviewers from the third and second editions: Rebecca 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 arlindseysmash.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:

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 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 Johansen**, 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.

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!

Jon Galloway



O'Reilly online learning



For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <http://oreilly.com>.

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#

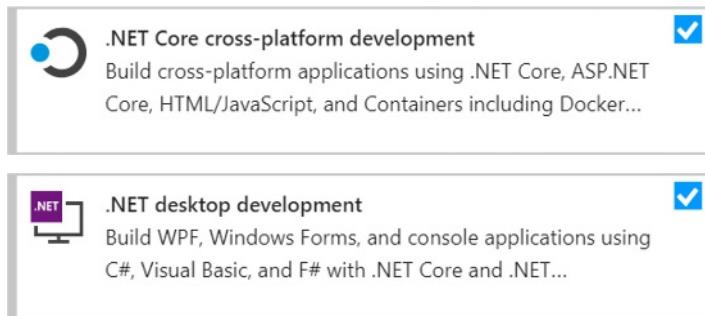
C# is a simple, modern language that lets you do incredible things. When you learn C#, you're learning more than just a language. C# unlocks the whole world of .NET, an incredibly powerful open source platform for building all sorts of applications.

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.)

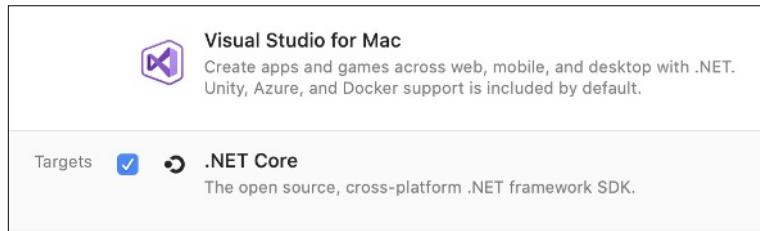
If you're on Windows...

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.



If you're on a Mac...

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



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#.

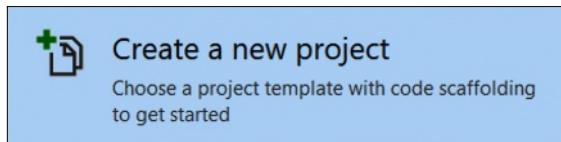
Create your first project in Visual Studio

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!

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.



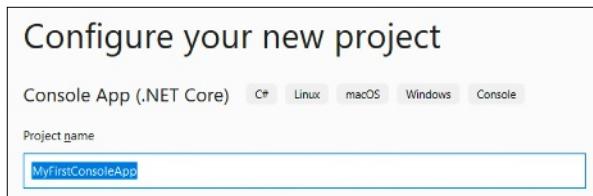
Choose the **Console App (.NET Core)** project type by clicking on it, then press the **Next** button.



← 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.

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



If you’re using **Visual Studio for Mac**, the code for this project—and all .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.

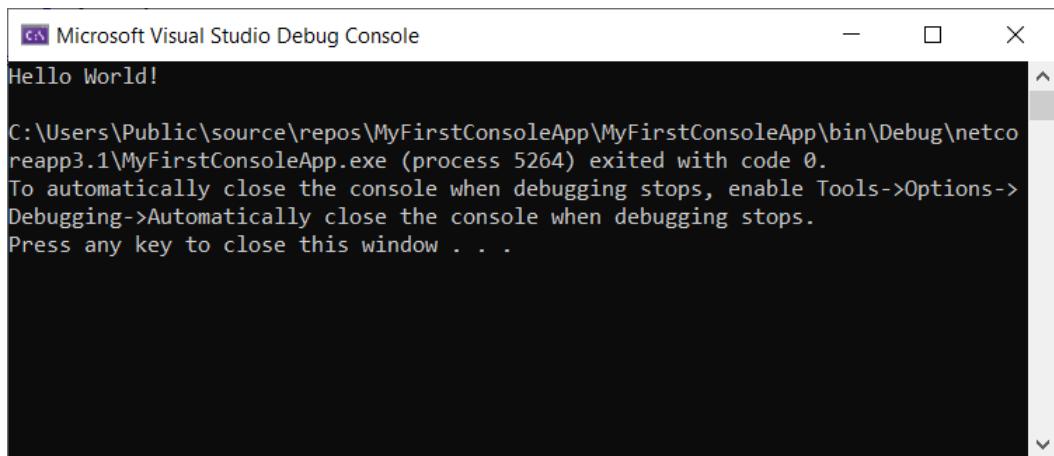
The app Visual Studio created for you is ready to run. At the top of the Visual Studio IDE, find the button with a green triangle and your app's name and click it:



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:

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 of 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:

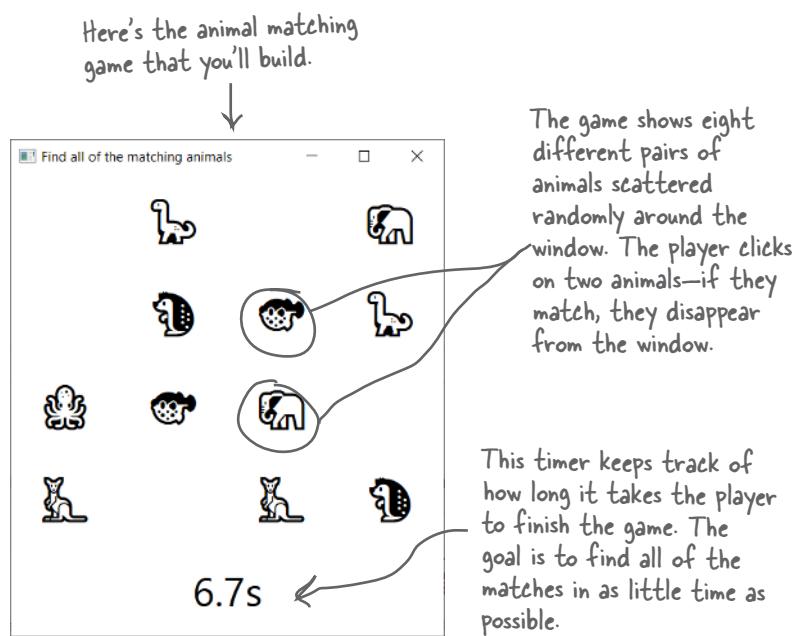
```
C:\path-to-your-project-folder\MyFirstConsoleApp\MyFirstConsoleApp\bin\Debug\netcoreapp3.1\MyFirstConsoleApp.exe (process ####) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press 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.



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.

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.

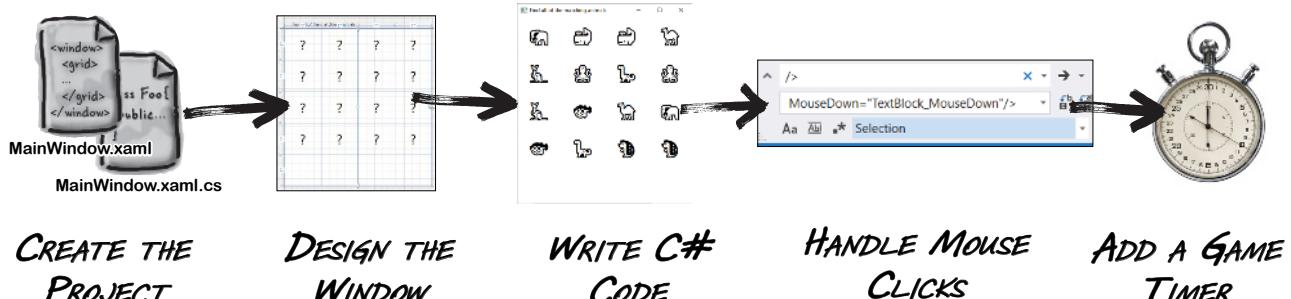
By the time you're done with this project, you'll be a lot more familiar with the tools that you'll rely on throughout this book to learn and explore C#.

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.



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.



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:

Game design... and beyond

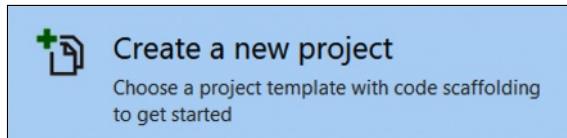
A game is a program that lets you play with it in a way that (hopefully) is at least as entertaining to play as it is to build.



files files so many files

Create a WPF project in Visual Studio

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

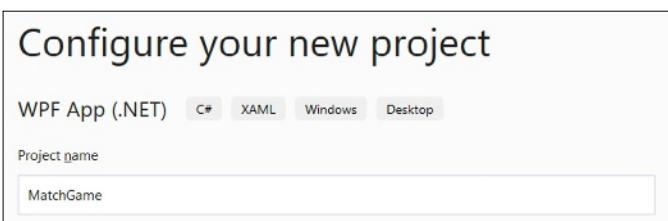


We're done with the Console App project you created in the first part of this chapter, so feel free to close that instance of Visual Studio.

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



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):



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

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



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

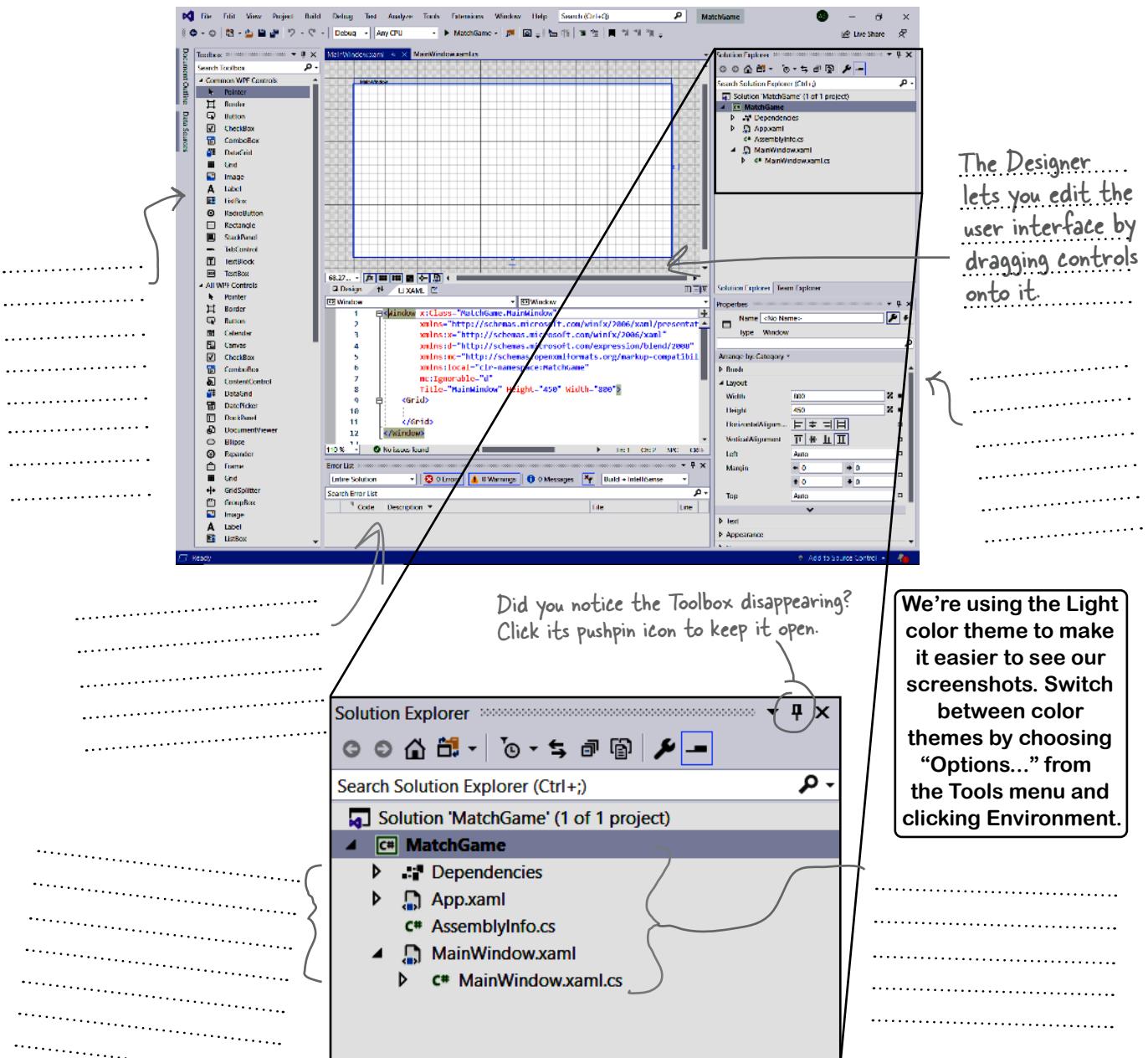
If you run into any trouble with this project, go to our GitHub page and look for a link to a video walkthrough: <https://github.com/head-first-csharp/fourth-edition>.



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.

start building with c#

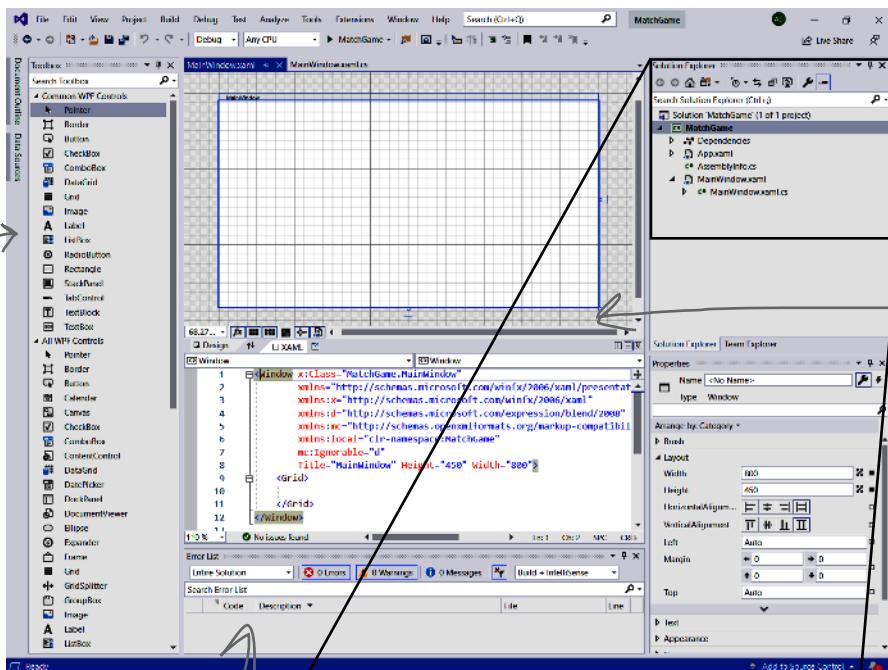
Adjust your IDE to match the screenshot below. First, open **MainWindow.xaml** by double-clicking on it in the Solution Explorer window. Then open the **Toolbox** and **Error List** windows by choosing them from the **View menu**. You can actually figure out the purpose of many of these windows and files 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.



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.



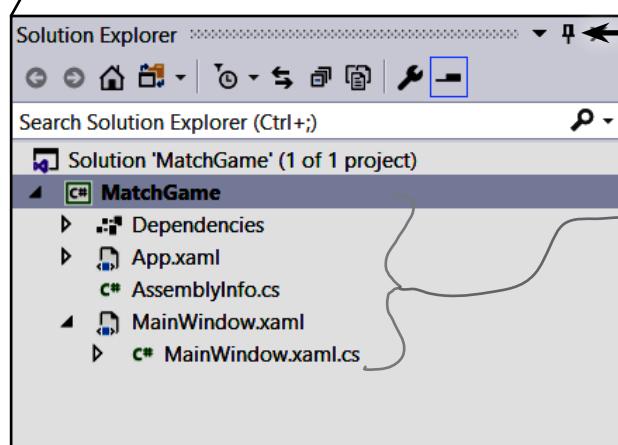
This is the Toolbox. It has a bunch of visual controls that you can drag onto your window.

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

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

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

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.



Clicking this pushpin icon turns auto-hide on or off. The Toolbox window has auto-hide turned on by default.

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

there are no Dumb Questions

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

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

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

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

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

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

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 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.

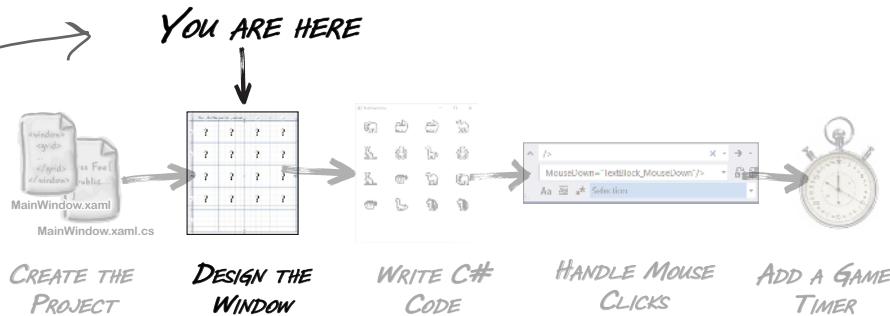
Visual Studio
will generate
code you
can use as
a starting
point for your
applications.

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

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

xaml rhymes with camel

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.

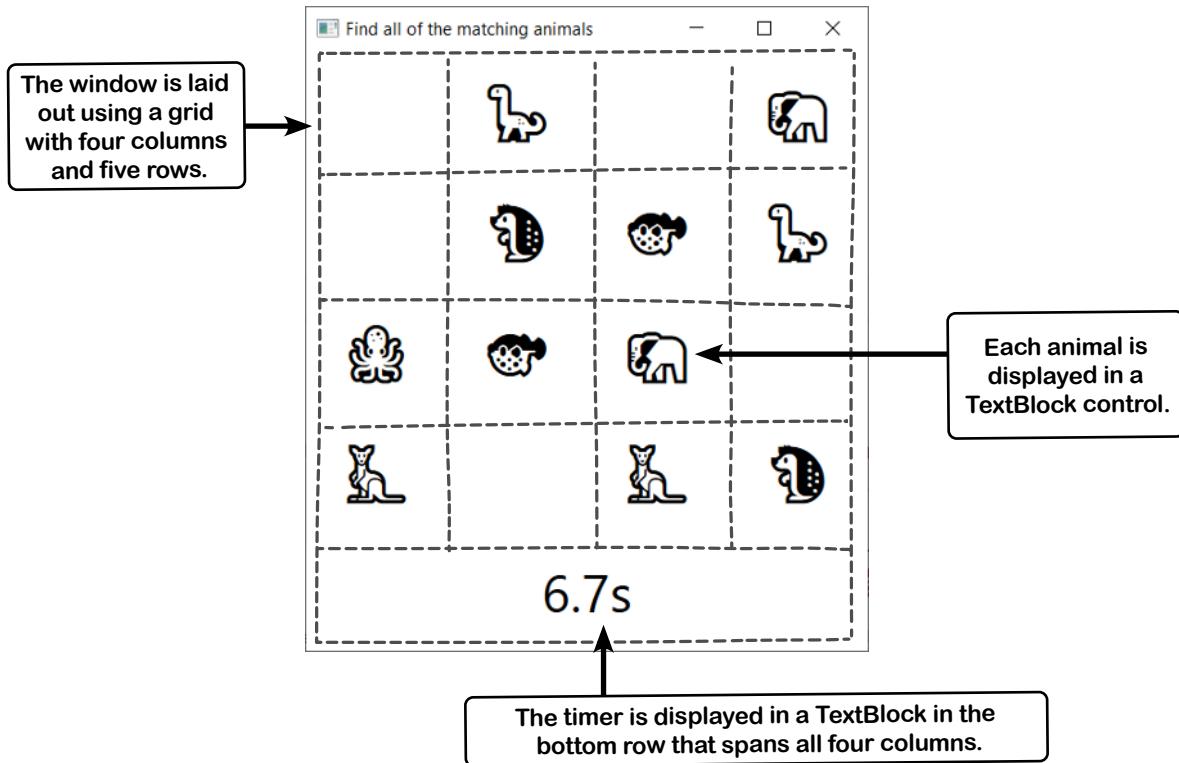


These numbers in the screenshot show the parts of the UI that correspond to the similar numbers we added to the XAML code.

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:



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:***

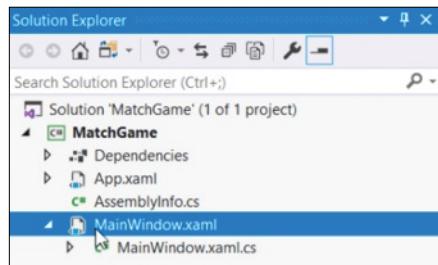
XAML is code that defines the user interface. C# is code that defines the behavior.

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.

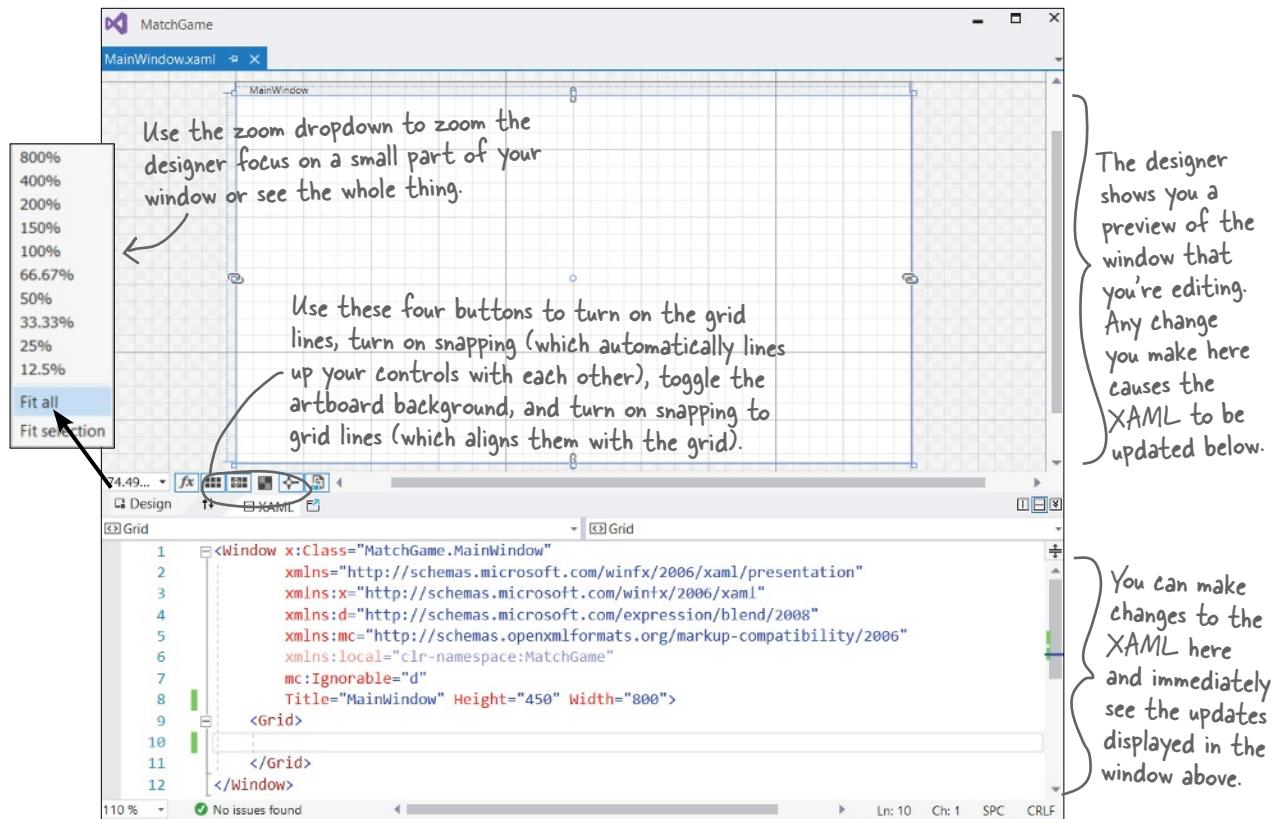
1 Select the main window.

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.



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.

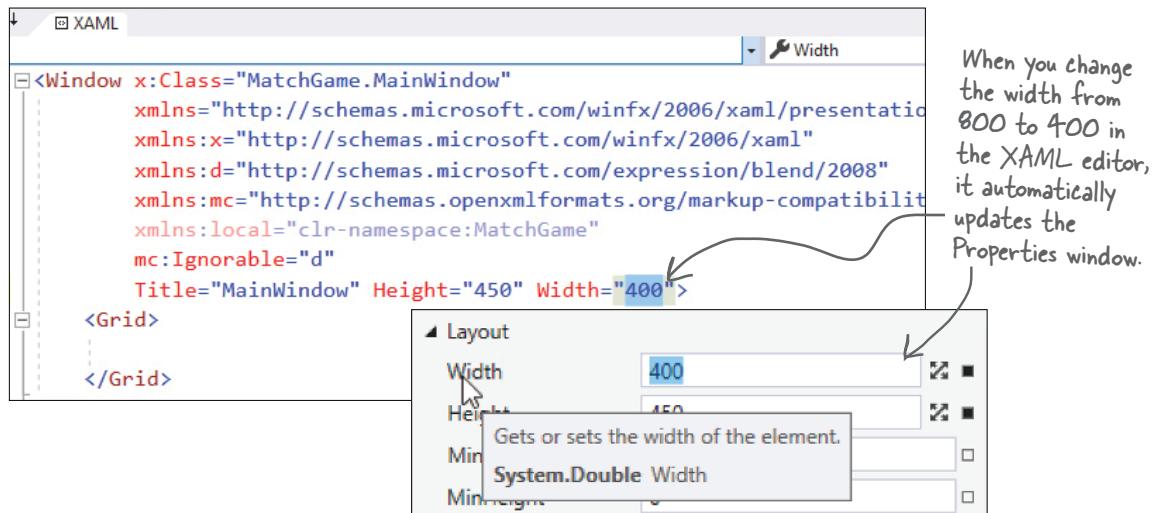
You can make changes to the XAML here and immediately see the updates displayed in the window above.

```
1 <Window x:Class="MatchGame.MainWindow"
2   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6   mc:Ignorable="d"
7   Title="MainWindow" Height="450" Width="800">
8   <Grid>
9     </Grid>
10  </Window>
```

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.



3 Change the window title.

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

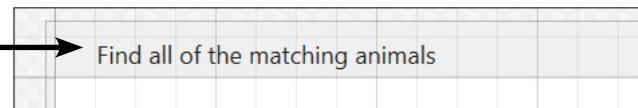
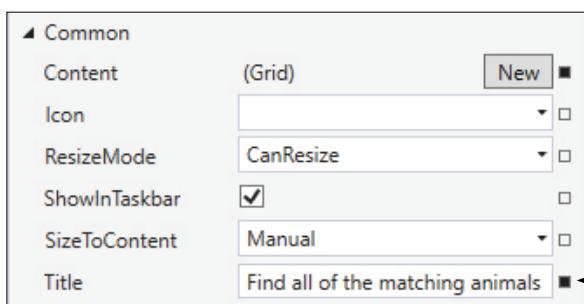
```
Title="MainWindow" Height="450" Width="800">
```

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.

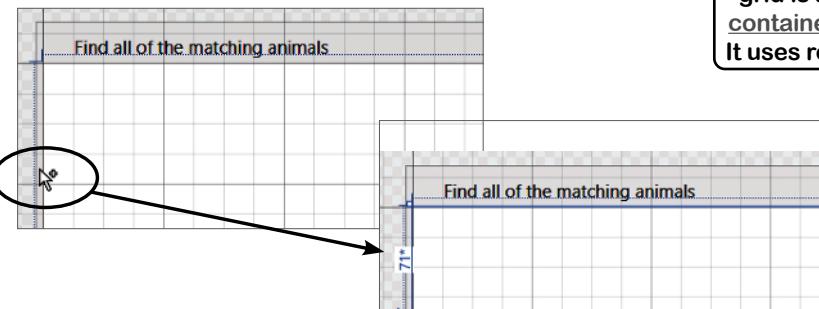
When you modify properties in your XAML tags, the changes immediately show up in the Properties window. When you use the Properties window to modify your UI, the IDE updates the XAML.



Add rows and columns to the XAML grid

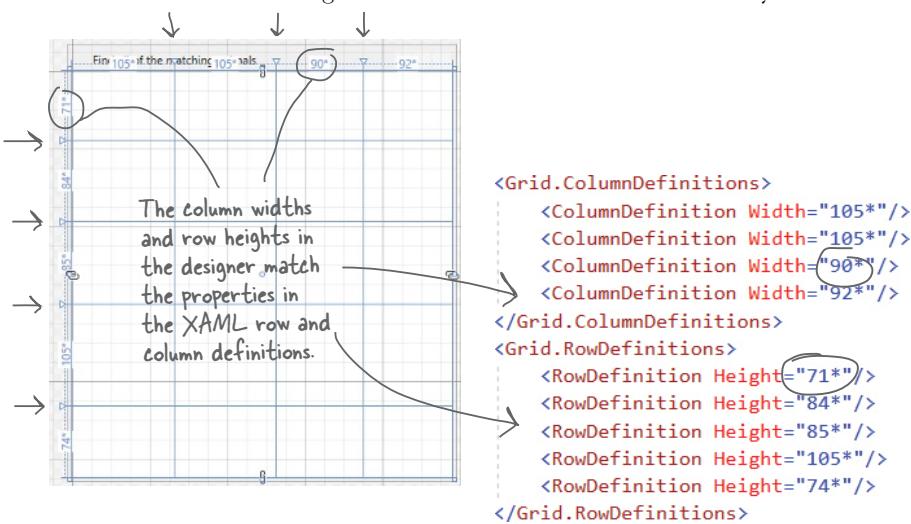
It might look like your main window is empty, but have a closer look at the bottom of the XAML. Notice how there's a line with `<Grid>` followed by one with `</Grid>`? Your window actually has a **grid**—you just don't see anything because it doesn't have any rows or columns. Let's go ahead and add a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



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.



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.

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



Watch it!

Things may look a bit different in your IDE.

All of the screenshots in this book were taken with **Visual Studio 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.

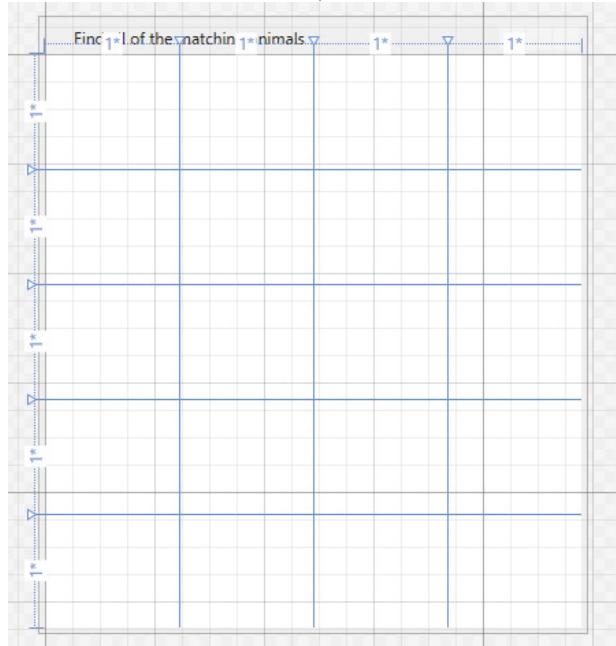
Make the rows and columns equal size

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:



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

Here's what you should see in the designer:



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.

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.

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:

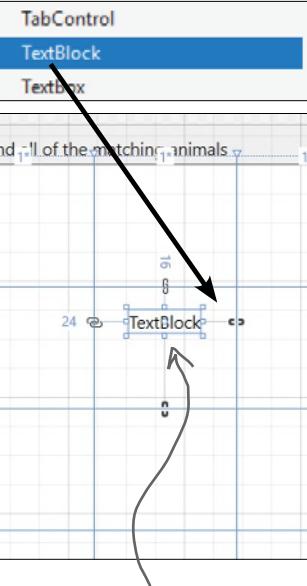
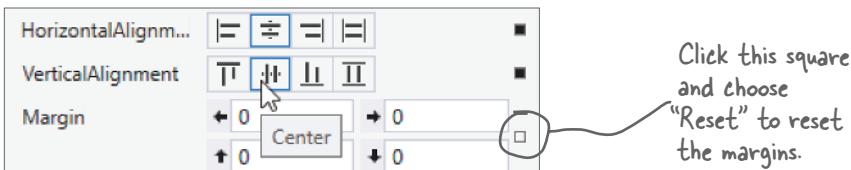
```
<TextBlock Text="TextBlock"  
    HorizontalAlignment="Left" VerticalAlignment="Center"  
    Margin="560,0,0,0" TextWrapping="Wrap" />
```

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.

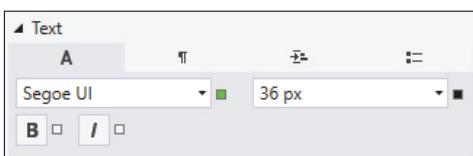
Your properties may be in a different order, and the Margin property will have different numbers because they depend on where in the cell you dragged it. All of these properties can be modified or reset using the IDE's Properties window.

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**.



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 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.



Exercise

You have one `TextBlock`—that's a great start! But we need 16 `TextBlock`s 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!

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 Solution

```
<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.

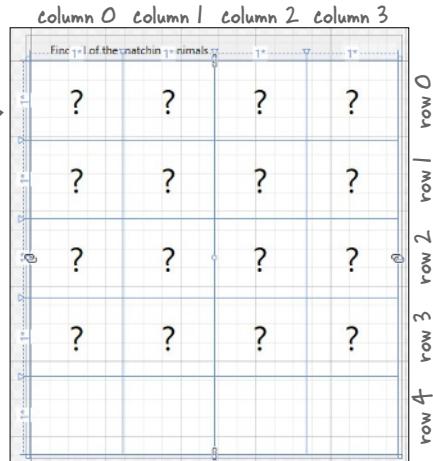
```
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="1"/>
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="2"/>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

<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"/>

<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"/>

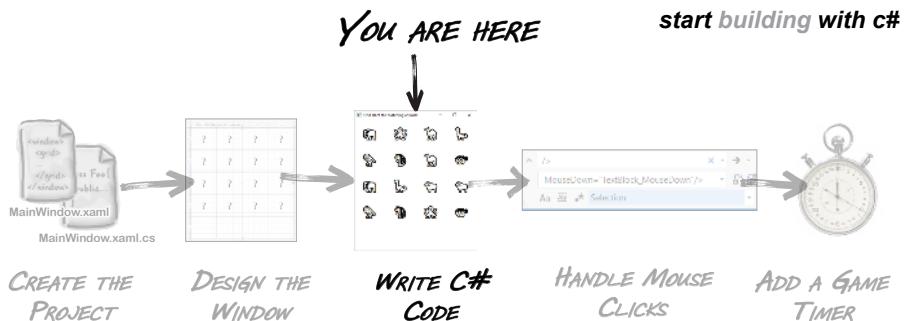
<TextBlock Text="?" FontSize="36" Grid.Row="4" HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="4" Grid.Column="1" HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="4" Grid.Column="2" HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="4" 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).

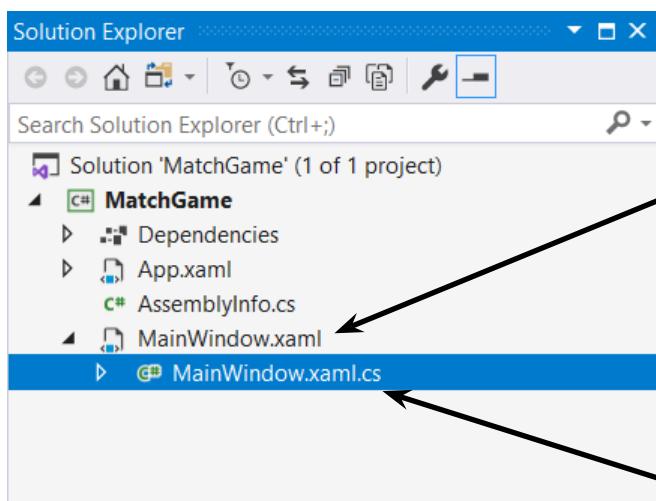
It's OK if you included Grid.Row or Grid.Column properties with a value of 0. We left them out because 0 is the default value.

There's a lot of code here, but it's really just the same line repeated 16 times with small variations. Every line that starts with **<TextBlock** has the same four properties (**Text**, **FontSize**, **HorizontalAlignment**, and **VerticalAlignment**). They just have different **Grid.Row** and **Grid.Column** properties. (The properties can be in any order.)



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

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



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

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



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.

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.

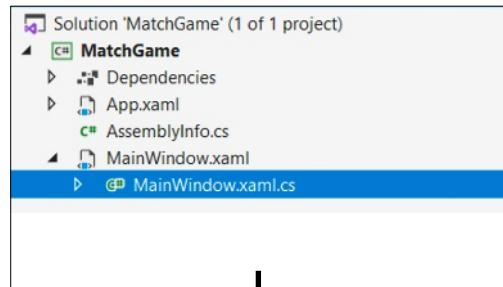
2 Generate a method called SetUpGame.

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.



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

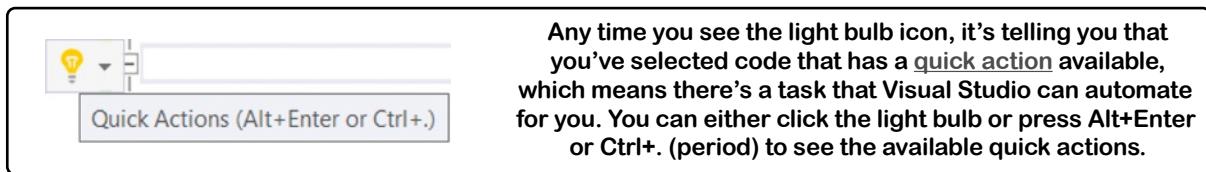


```
21   public partial class MainWindow : Window
22   {
23       0 references
24       public MainWindow()
25       {
26           InitializeComponent();
27           Generate method 'MainWindow.SetUpGame'
28           Introduce local for 'SetUpGame0'
29       }
30   }
```

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.

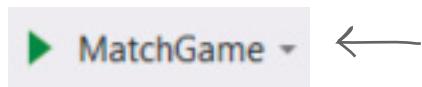
When you click on the Quick Actions icon, it shows you a context menu with actions. If an action generates code, it shows you a preview of the code it will generate. Choose the “Generate method” action to generate a new method called `SetUpGame`.





3 Try running your code.

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



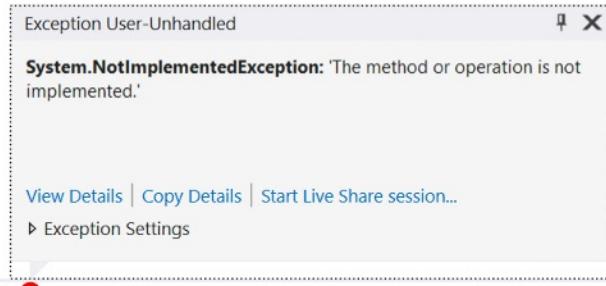
The “Start Debugging” button in the toolbar at the top of the IDE starts your app. You can also use Start Debugging (F5) from the Debug menu to start your app.

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

```

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

```



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.

the plural of emoji is emoji

Finish your SetUpGame method

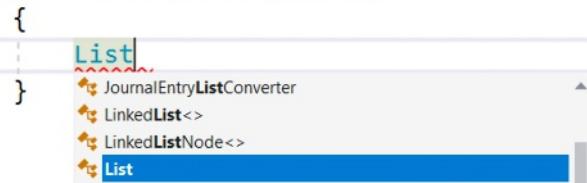
You put your SetUpGame method inside the `public MainWindow()` method because everything inside that method is called as soon as your app starts.

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

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()
```

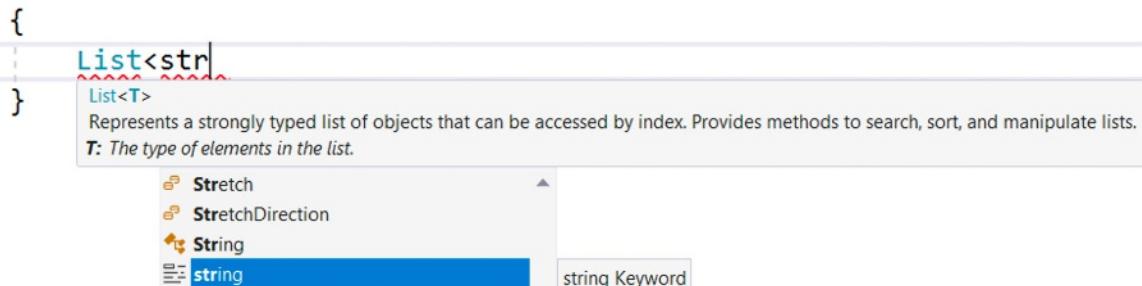


You'll learn a lot more about methods soon.

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.

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

```
private void SetUpGame()
```



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:

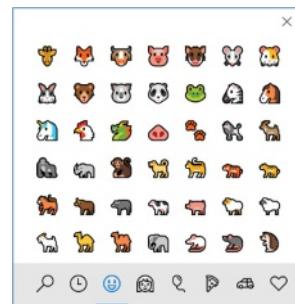
```
List<string> animalEmoji = new List<string>()
{
}
```

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 pairs of animal emoji between the brackets**. Add a ; after the closing curly bracket:

```
List<string> animalEmoji = new List<string>()
{
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹";
};
```

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



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

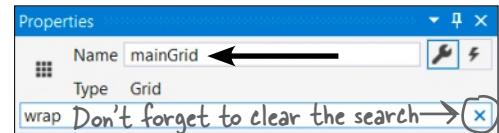
③

Finish your method.

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

`Random random = new Random();` ← This line goes right after the closing bracket and semicolon.

```
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 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.

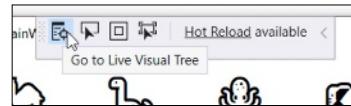
you built it and it works—excellent job

Run your program

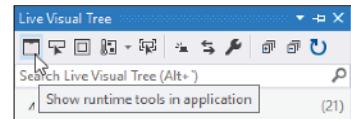
Click the  MatchGame  button in the IDE's toolbar to start your program running. A window will pop up with your eight pairs of animals in random positions:



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:



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

What it does

```
List<string> animalEmoji = new List<string>()
{
    "🦁", "🦁",
    "🐯", "🐯",
    "🐘", "🐘",
    "🐵", "🐵",
    "🐺", "🐺",
    "🐆", "🐆",
    "🐱", "🐱",
    "🐹", "🐹"
};
```

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

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

- Create a list of eight pairs of emoji

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

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

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

Create a new random number generator

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

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

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

* WHO DOES WHAT? *

solution

C# statement

What it does

```
List<string> animalEmoji = new List<string>()
{
    "🐶", "🐱",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
    "🐹", "🐹",
};
```

Update the TextBlock with the random emoji from the list

```
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);
}
```

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 mistakes (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

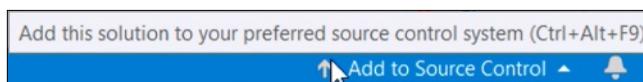
- Visual Studio is **Microsoft's IDE**—or **integrated development environment**—that simplifies and assists in editing and managing your C# code files.
- **.NET Core console apps** are cross-platform apps that use text for input and output.
- The IDE's **AI-assisted IntelliSense** helps you enter code 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.

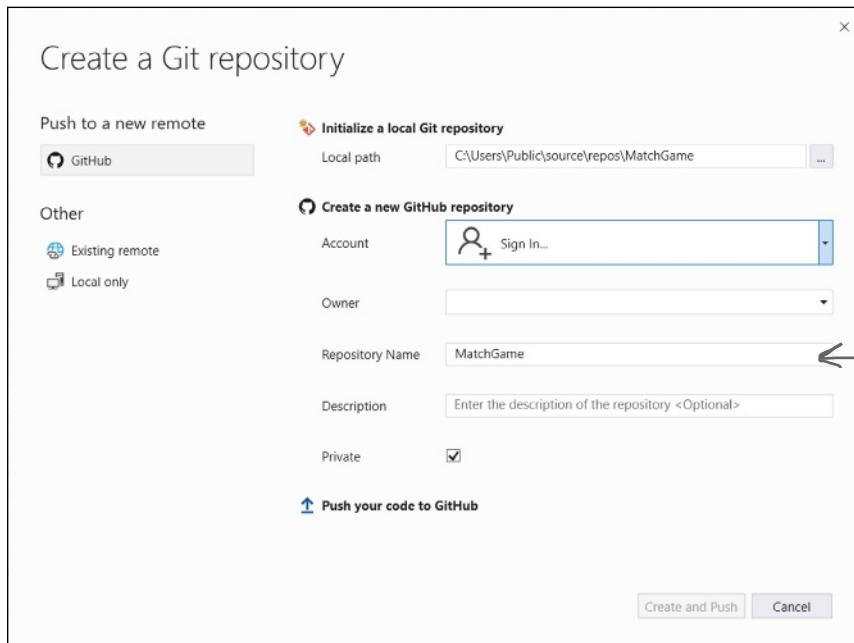
Find **Add to Source Control** in the status bar at the bottom of the IDE:



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



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



Adding your project to source control is optional.

Maybe you're working on a computer on an office network that doesn't allow access to GitHub, 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.

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.

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  [Sign In...](#). 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.

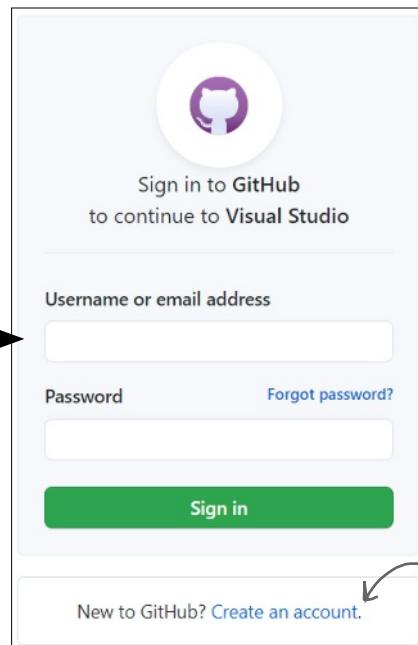
Once you're logged into GitHub, you'll be returned to the “Create a Git Repository” window in Visual Studio. If you want other people to see your code, **unchecked the Private checkbox** to make your new repository public.

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.

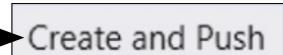
Keep an eye on the status bar. If you see a number like  2 it's telling you that there are unpushed commits that you can push to the GitHub repo.

Once you've published your code to GitHub, you can use the commands in the Git menu to work with your Git repo.

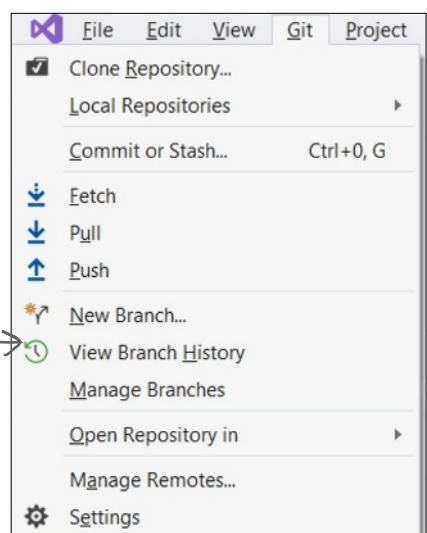
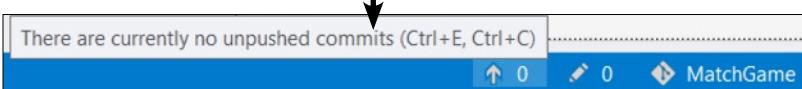
Go to <https://github.com/<your-github-username>/MatchGame> to see the code that you just pushed. When you sync your project to the remote repo, you'll see updates in the Commits section.



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



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



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:

Error List					
Entire Solution		Code	Description	Project	File
				Line	Suppression State
		CS1001	Identifier expected	MatchGame	MainWindow.xaml.cs
		CS1002	; expected	MatchGame	MainWindow.xaml.cs
		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

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.

YOU ARE HERE



CREATE THE PROJECT



DESIGN THE WINDOW



WRITE C# CODE



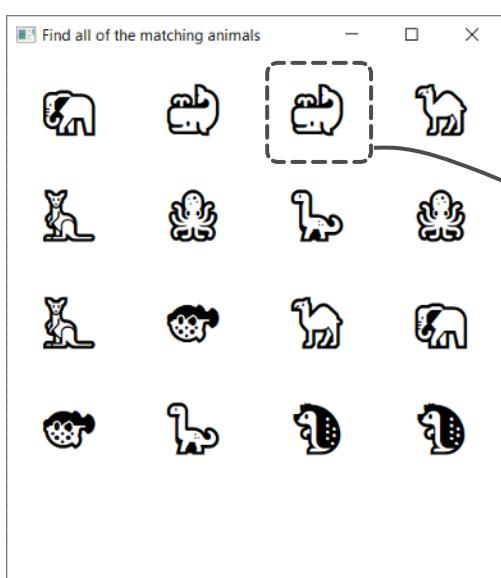
HANDLE MOUSE CLICKS



ADD A GAME TIMER

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).
 */
```

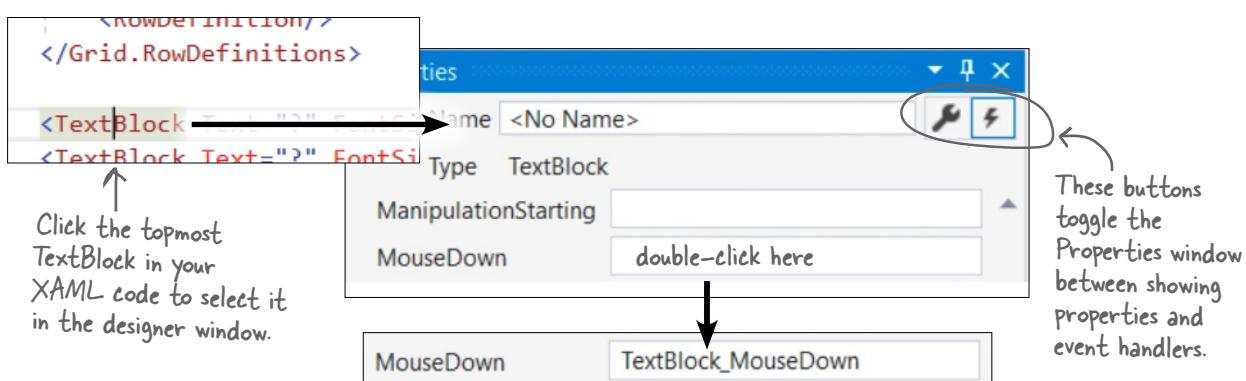
This is a comment. Everything between `/*` and `*/` is ignored by C#. We added this comment to tell you what your `TextBlock_MouseDown` method will do—and also to show you what a comment looks like.

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.



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



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 does the 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:

```

MatchGame - MainWindow.xaml.cs
MainWindow.xaml.cs

MatchGame.MainWindow

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 **fields**. 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.

you've got these mouse clicks handled

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

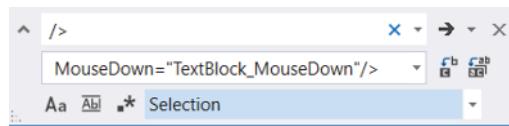
Right now only the first TextBlock has an event handler hooked up to its MouseDown event. Let's hook up the other 15 TextBlocks to it, too. You *could* do it by selecting each one in the designer and entering `TextBlock_MouseDown` into the box next to MouseDown. We already know that just adds a property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



← 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.



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?



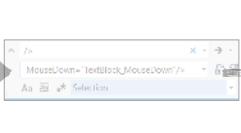
CREATE THE PROJECT



DESIGN THE WINDOW



WRITE C# CODE



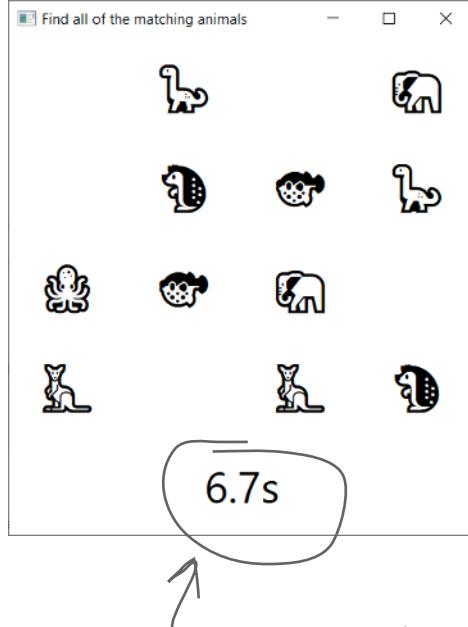
HANDLE MOUSE CLICKS

ADD A GAME TIMER



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();
```

Next, add these two lines of code. Start typing the second line: "timer.Tick +≡"
As soon as you add the equals sign, the IDE will display this "Press TAB to insert" message.

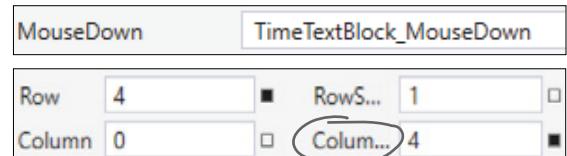
- ④ Press the Tab key. The IDE will finish the line of code and add a `Timer_Tick` method:

```
0 references  
public MainWindow()  
{  
    InitializeComponent();  
  
    timer.Interval = TimeSpan.FromSeconds(.1);  
    timer.Tick += Timer_Tick;  
    SetUpGame();  
}  
  
1 reference  
private void Timer_Tick(object sender, EventArgs e)  
{  
    throw new NotImplementedException();  
}
```

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

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

- ★ Drag a **TextBlock** into the lower-left square.
- ★ 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.



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

Here's what the XAML will look like—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) } This resets the game if all 8 matched
    { pairs have been found (otherwise it does
        SetUpGame(); 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?



Use the debugger to troubleshoot the exception

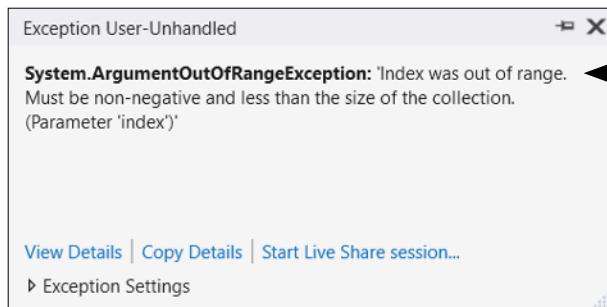
You might have heard the word “bug” before. You might have even said something like this to your friends at some point in the past: “That game is really buggy, it has so many glitches.” Every bug has an explanation—everything in your program happens for a reason—but not every bug is easy to track down.

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!)



1 Restart your game a few times.

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



An exception is C#'s way of telling you that something went wrong when your code was running. 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:

```
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;
```

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

Exception User-Unhandled

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

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

16 references

```
private void TextBlock_MouseDown(object sender,
```

Exception Settings

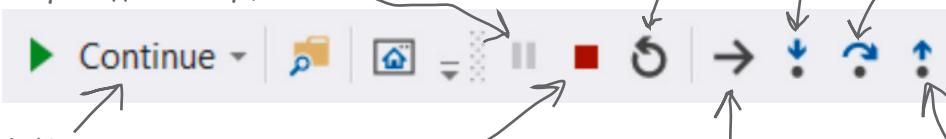
This exception is **reproducible**: you can reliably get your program to throw the exact same exception, and you have a really good idea of where the problem is.



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.

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



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

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

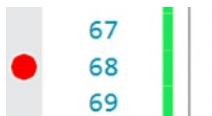
The Step Into button executes the next statement. If that statement is a method, it only executes the first statement inside the method.

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

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

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

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



```

67
68 int index = random.Next(animalEmoji.Count);
69 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?

foreach is a kind of loop 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.

Behind the Scenes



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 `ArgumentOutOfRangeException` exception. What caused it to run out of emoji to add?

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 code
to fix the bug.

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

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

you've done great job

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?

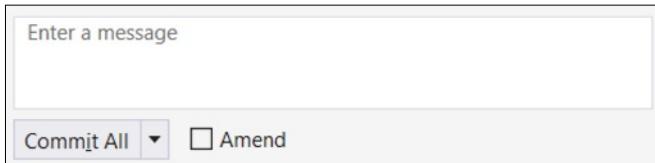


Go to <https://github.com/head-first-csharp/fourth-edition> to view and download the complete code for this project and all of the other projects in this book.

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.

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



- 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).

- 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!

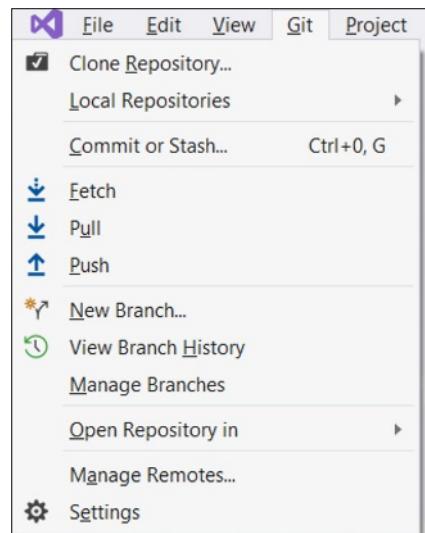


IT WAS REALLY USEFUL TO BREAK THE GAME UP INTO SMALLER PIECES THAT I COULD TACKLE ONE AT A TIME.

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.



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.

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?



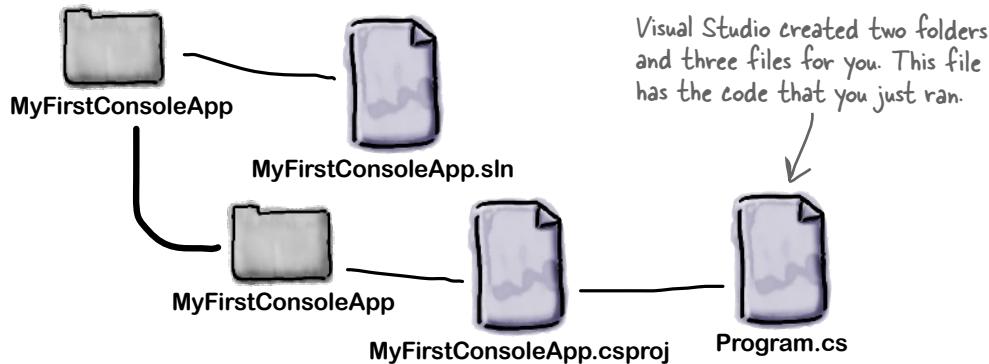
You're not just an IDE user. You're a developer.

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.



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

A screenshot of Visual Studio showing the code for Program.cs. The code is as follows:

```
1  using System;
2
3  namespace MyFirstConsoleApp
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
13 }
```

A callout bubble points to the Main method with the text: "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."

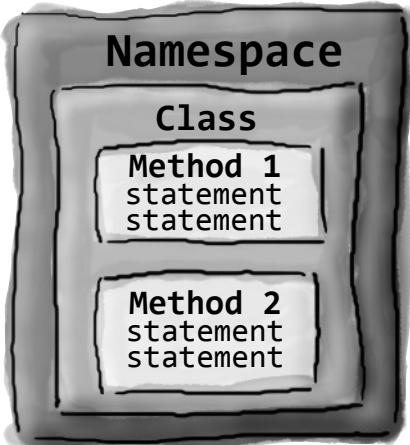
This is a screenshot of Visual Studio for Windows. If you're using macOS the screen will look a little different, but the code will be the same.

- ★ 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 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**.

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

MoreClasses.cs

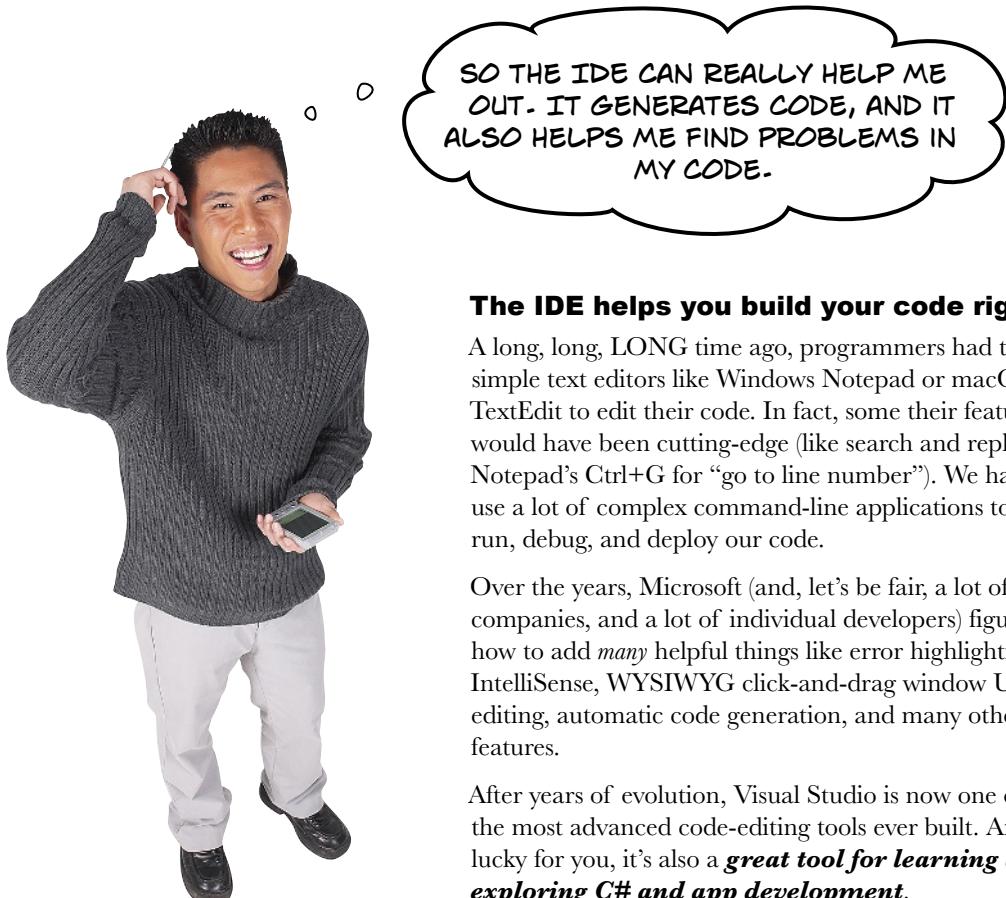
```
namespace PetFiler2 {  
  
    public class Fish {  
        public void Swim() {  
            // statements  
        }  
    }  
  
    public partial class Cat {  
        public void Purr() {  
            // statements  
        }  
    }  
}
```

SomeClasses.cs

```
namespace PetFiler2 {  
  
    public class Dog {  
        public void Bark() {  
            // statements go here  
        }  
    }  
  
    public partial class Cat {  
        public void Meow() {  
            // more statements  
        }  
    }  
}
```

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

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



The IDE helps you build your code right.

A long, long, LONG time ago, programmers had to use simple text editors like Windows Notepad or macOSTextEdit to edit their code. In fact, some of 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.

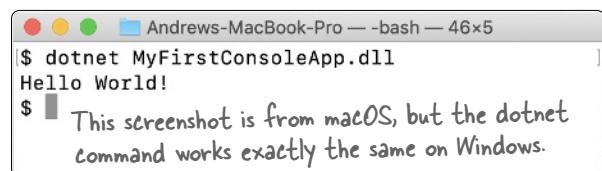
So when you run your program, how does it know which statement to start with? That's where the entry point comes in. Your code won't build unless there is **exactly one method called Main**. It's called the entry point because the program starts running—we say that it *enters* the code—with the first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

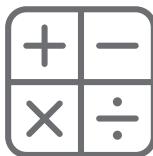
Your app is made up of classes, and those classes contain methods, and those methods contain statements. So if we want to build apps that do a lot of things, we'll need a few **different kinds of statements** to make them work. You've already seen one kind of statement:

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

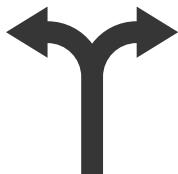
This is a **statement that calls a method**—specifically, the Console.WriteLine method, which prints a line of text to the console. We'll also use a few other kinds of statements in this chapter and throughout the book. For example:



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.



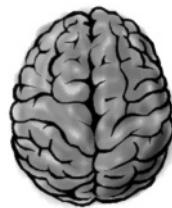
Conditionals let our code choose between options, either executing one block of code or another.



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;
```

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.

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.

Variables vary

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

```
int myHeight = 63;
```

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

```
myHeight = 12;
```

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

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

You need to assign values to variables before you use them

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

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

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

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

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

Do this!

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.



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();



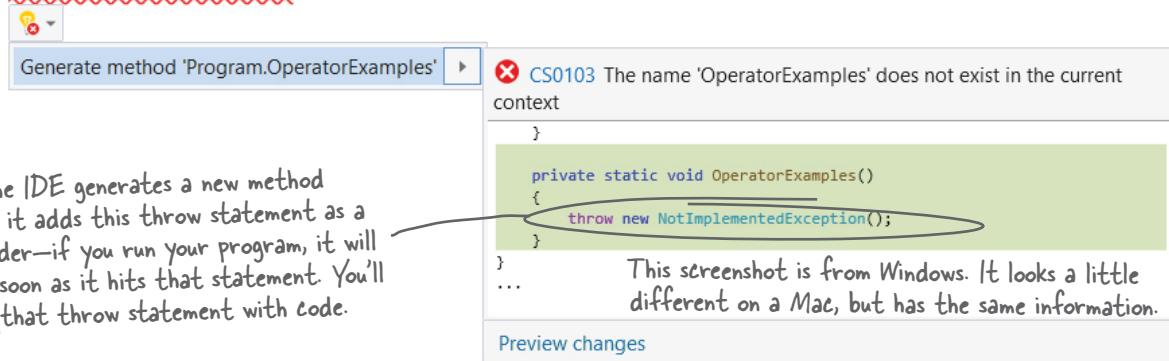
On a Mac, click the link or press Option+Return to show the potential fixes.

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();



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

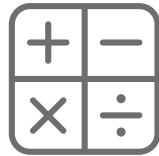
The statements you just added to your code will write three lines to the console: each `Console.WriteLine` statement prints a separate line. **Before you run your code**, figure out what they'll be and write them down. And don't bother looking for a solution, because we didn't include one! Just run the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

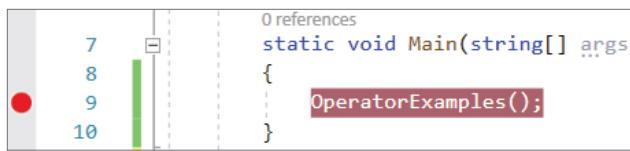


Debug
this!

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:



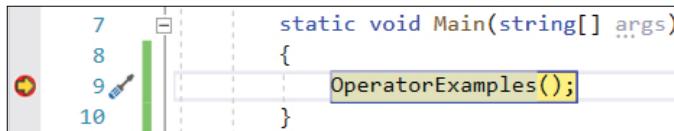
```
0 references
static void Main(string[] args)
{
    OperatorExamples();
}
```

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.

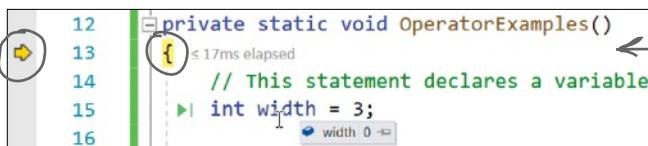


```
static void Main(string[] args)
{
    OperatorExamples();
}
```

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()
{
    // This statement declares a variable
    int width = 3;
    width 0
}
```

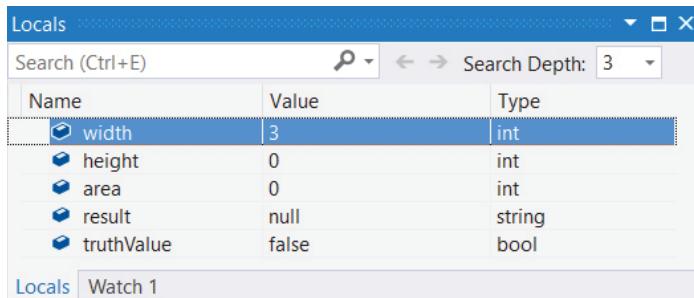
The highlighted bracket and arrow in the left margin mean the code is paused just before the first statement of the method.

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.

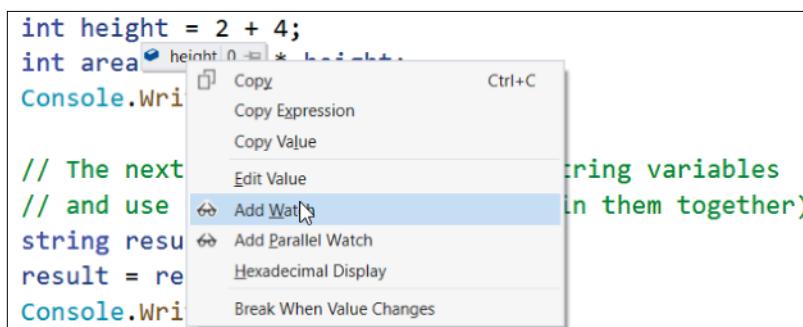


The Locals and Watch windows in Visual Studio for Mac look a little different than they do on Windows, but they contain the same information. You add watches the same way in both Windows and Mac versions of Visual Studio..

5

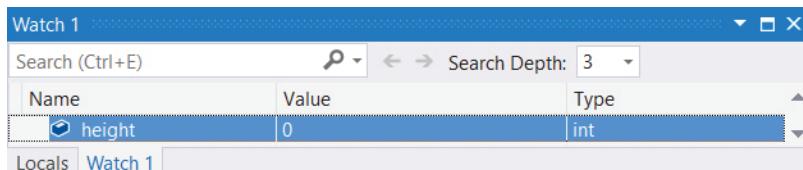
Add a watch for the height variable.

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.



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

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



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)`



Watch it!
Don't
confuse
the two
equals
sign operators!

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

Use operators to compare two int variables

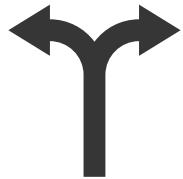
You can do simple tests by checking the value of a variable using a comparison operator. Here's how you compare two ints, `x` and `y`:

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

These are the ones you'll use most often.

"if" statements make decisions

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



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

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

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

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

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

if/else statements are just what they sound like: if a condition is true 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) REMEMBER - always use two equals signs to
{                               check if two things are equal to each other.
    // You can have as many statements
    // as you want inside the brackets
    message = "The value was 24.";
}
else
{
    message = "The value wasn't 24.";
}
```

do for loops study for conditional tests?

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 (`i = 0`), the conditional test (`i < 8`), and the iterator (`i = i + 2`). Each time through a for loop (or any loop) is called an iteration.

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

for Loops Up Close



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

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

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

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

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

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

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

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



Sharpen your pencil

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

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

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

// Loop #4
int i = 0;
int count = 2;
while (i == 0) {
```

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

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

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

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

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



Sharpen your pencil Solution



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

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

Loop #1 executes once.

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 #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.

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

Loop #3 executes 8 times.

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

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



Use code snippets to help write loops

Do this!

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

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

```
while (true)
{
}
```

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

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

IDE Tip: Brackets

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

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

```
do
{
}
```

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.



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



Mechanics

Game design... and beyond

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*.

Enter a number

4

- ★ 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.

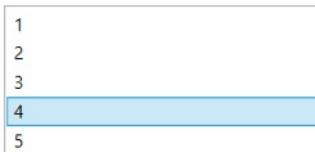
Enter a number

- 1
 2
 3
 4
 5

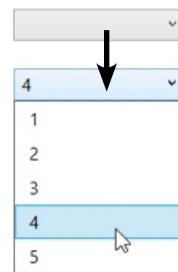
- ★ **Radio buttons** let you restrict the user's choice. You can use them for numbers if you want, and you can choose how you want to lay them out.



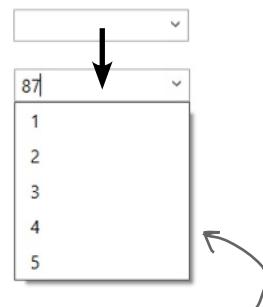
- ★ The other controls on this page can be used for other types of data, but **sliders** are used exclusively to choose a number. Phone numbers are just numbers, too. So *technically* you could use a slider to choose a phone number. Do you think that's a good choice?



- ★ 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.



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.

7,183,876,962

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

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

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.

Do this!

This TextBox lets you type text. You'll add code to make it only accept numeric input.

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 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.

Relax

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

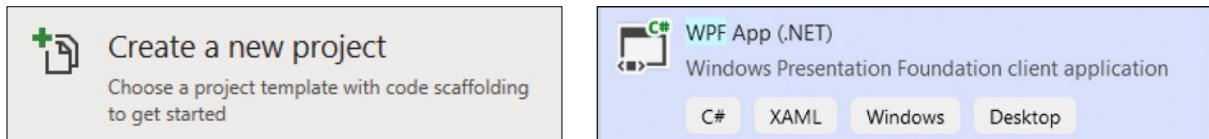
This Do this! and these exercises are all about getting some practice using XAML to build a UI with controls. You can always refer back to it when we use these controls in projects later in the book.



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).



Name your project **ExperimentWithControls**.

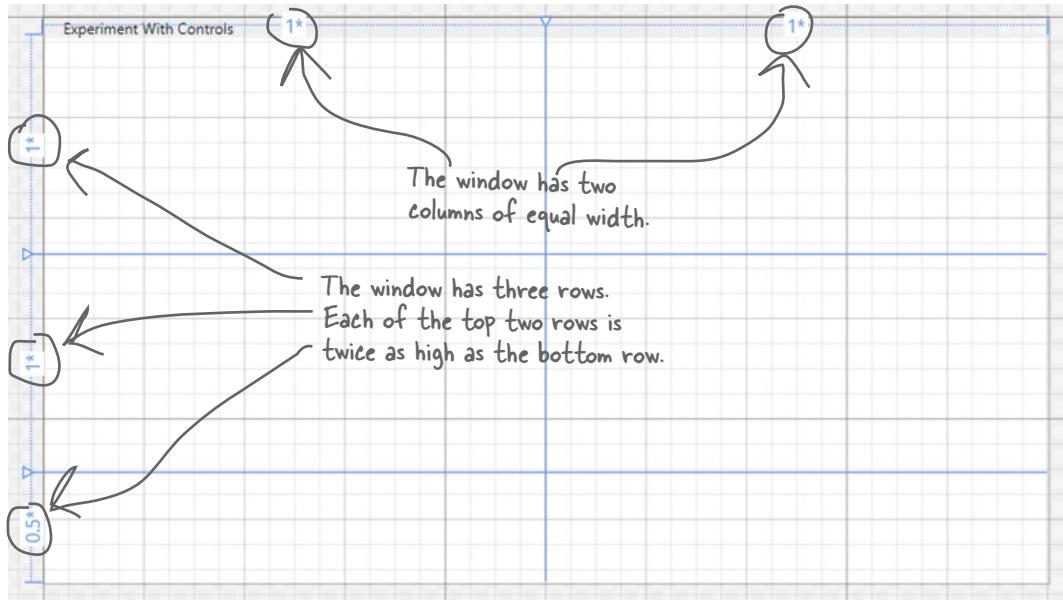
Set the window title

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:





Exercise Solution

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.ColumnDefinitions>` 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.

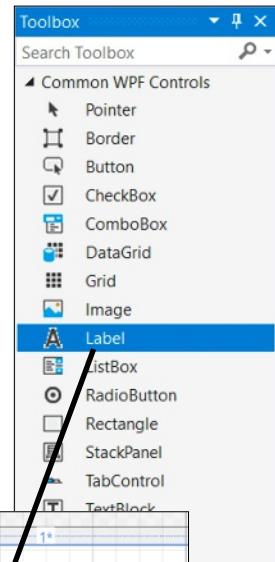


Add a TextBox control to your app

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).

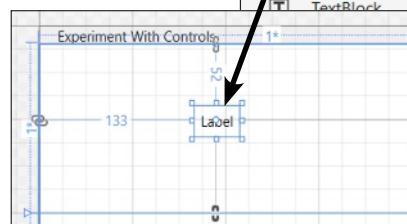
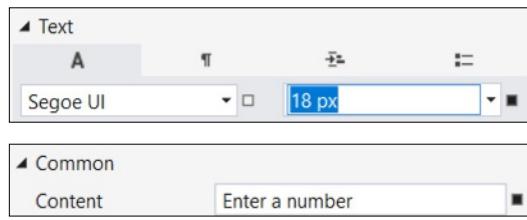
1 Drag a Label out of the Toolbox into the top-left cell of the grid.

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**.

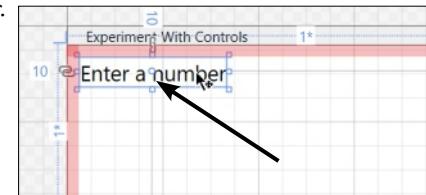


3 Drag the Label to the upper-left corner of the cell.

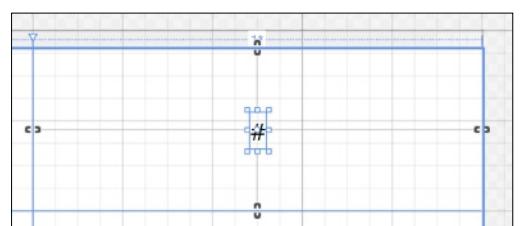
Click on the Label in the designer and drag it to the upper-left corner. When it's 10 pixels away from the left or top cell wall, you'll see gray bars appear and it will snap to a 10px margin.

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 TextBox 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**.





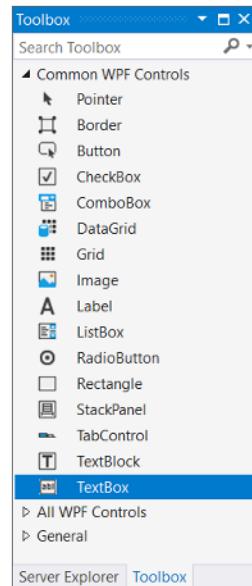
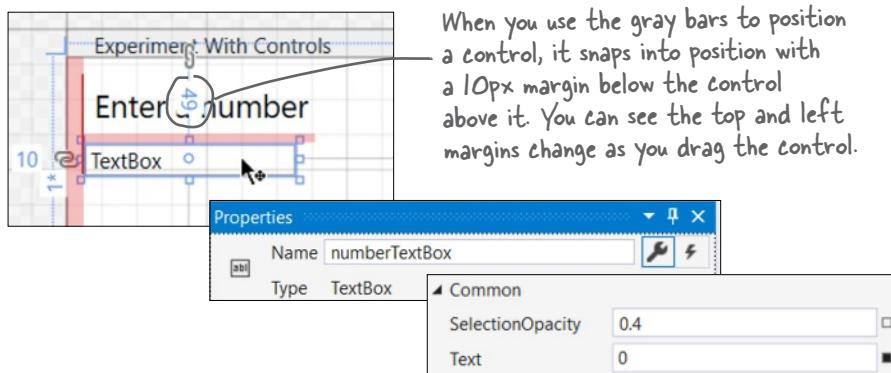
Here's the XAML for the TextBlock that goes in the upper right cell of the grid. You can use the visual designer or type in the XAML by hand. Just make sure your TextBlock has exactly the same properties as this solution—but like earlier, it's OK if your properties are in a different order.

```
<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**.



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:



```
<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" />
```

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

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

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

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

Exception User-Unhandled
System.NullReferenceException: 'Object reference not set to an instance of an object.'
number was null.
[View Details](#) | [Copy Details](#) | [Start Live Share session...](#)
[Exception Settings](#)

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

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

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

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

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



Sleuth it out

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

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

The problem is that the XAML for the TextBlock includes `Text="0"`, so when the app starts running it initializes the TextBlock 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.

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.
```

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

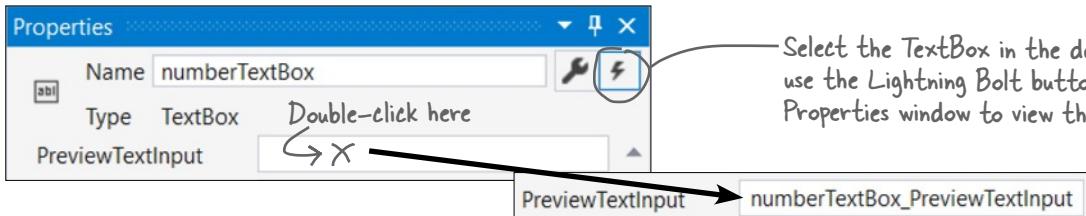
When you added the `MouseDown` event to your `TextBlock` in Chapter 1, you used the buttons in the upper-right corner of the Properties window to switch between properties and events. Now you'll do the same thing, except this time you'll use the **PreviewTextInput** event to only accept input that's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

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.

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.



Do this!



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

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

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

Here's how this event handler works:

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

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

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

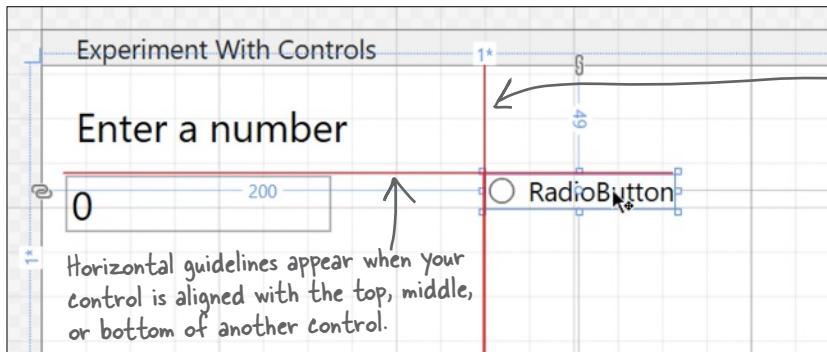
Now it's hooked up to two event handlers: the `TextChange` event is hooked up to an event handler method called `numberTextBox_TextChanged`, and right below it the `PreviewTextInput` event is hooked up to a method called `numberTextBox_PreviewTextInput`.



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

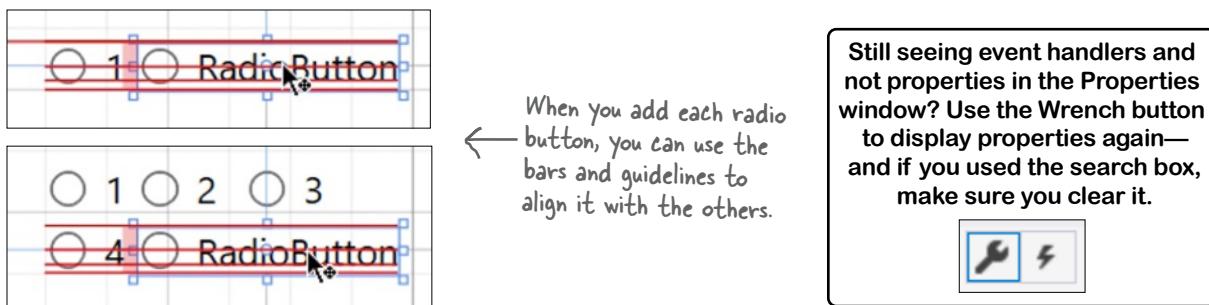
Add radio buttons to the upper-left cell next to the TextBox

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



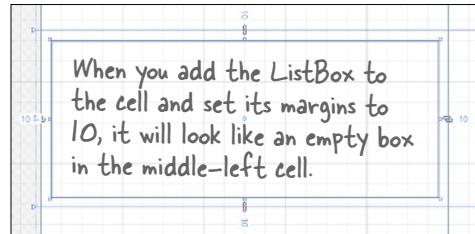
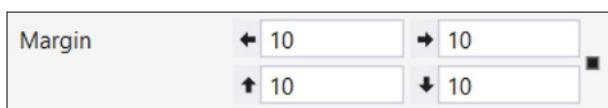
Expand the Common section of the Properties window and set the Content property of the RadioButton control to 1.

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



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.





Name your ListBox myListBox and add ListBoxItems to it

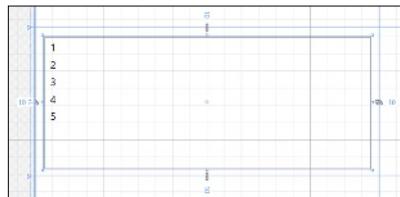
The purpose of the ListBox is to let the user choose a number. We'll do that by adding items to the list. Select the ListBox, expand Common in the Properties window, and click the Edit Items button next to Items (…). Add five ListBoxItem items and set their Content values to numbers 1 to 5.

Click the Edit Items button to open the Collection Editor window.

Add five ListBoxItems. Use the Common section to set the Content for each of them to a number (1, 2, 3, 4, or 5).

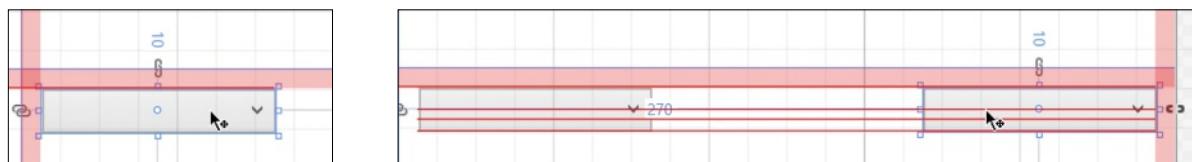
Add each item by choosing ListBoxItem from the dropdown and clicking Add.

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.

Common

IsDropDownOpen

IsEditable ■

IsReadOnly

The editable ComboBox looks different to let users know they can either type in their own value or choose one from the list.



Exercise Solution

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"/>
← The IDE added the
margin and alignment
properties to each
RadioButton control
when you dragged it
into place.
<RadioButton Content="4" Margin="200,69,0,0"
             HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="5" Margin="230,69,0,0"
             HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="6" Margin="265,69,0,0"
             HorizontalAlignment="Left" VerticalAlignment="Top"/>

<ListBox x:Name="myListBox" Grid.Row="1" Margin="10,10,10,10">
    <ListBoxItem Content="1"/>
    <ListBoxItem Content="2"/>
    <ListBoxItem Content="3"/>
    <ListBoxItem Content="4"/>
    <ListBoxItem Content="5"/>
    } 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.
</ListBox>

<ComboBox x:Name="readOnlyComboBox" Grid.Column="1" Margin="10,10,0,0" Grid.Row="1"
          HorizontalAlignment="Left" VerticalAlignment="Top" Width="120">
    <ListBoxItem Content="1"/>
    <ListBoxItem Content="2"/>
    <ListBoxItem Content="3"/>
    <ListBoxItem Content="4"/>
    <ListBoxItem Content="5"/>
    Make sure you gave
        your ListBox and
        two ComboBoxes the
        right names. You'll use
        them in the C# code.
    </ComboBox>
    } 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.

1 Add a slider to your app.

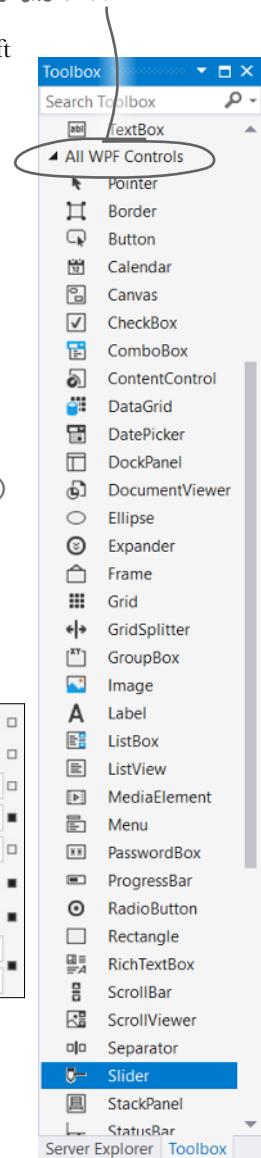
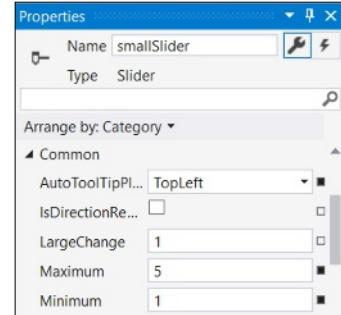
Drag a Slider out of the Toolbox and into the lower-right cell. Drag it to the upper-left corner of the cell and use the gray bars to give it left and top margins of 10.



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, RoutedEventArgs<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.

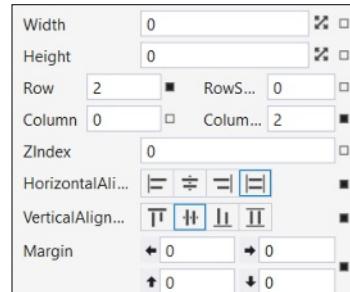


2 Add a ridiculous slider to choose phone numbers.

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.

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, RoutedEventArgs<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_CheckedChanged (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_CheckedChanged(  
    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_CheckedChanged"**—this is exactly like how the other event handlers were hooked up. **Copy this property to the other RadioButton tags** so they all have identical Checked properties—and **now they're all connected to the same Checked event handler**. You can use the Events view in the Properties window to check that each RadioButton is hooked up correctly.



If you switch the Properties window to the Events view, you can select any of the RadioButton controls and make sure they all have the Checked event hooked up to the RadioButton_CheckedChanged 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 ListViewItem 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();  
}
```

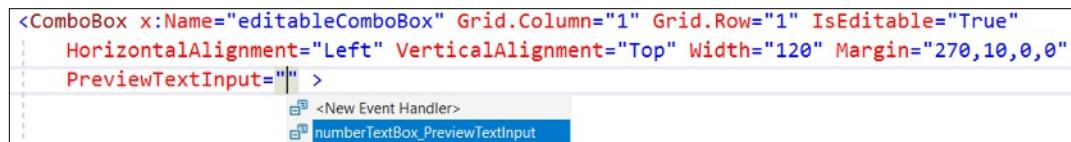
You can also use the Properties window to add a SelectionChanged event. If you accidentally do this, you can hit "undo" (but make sure you do it in both files).

5

Make the editable combo box update the TextBlock.

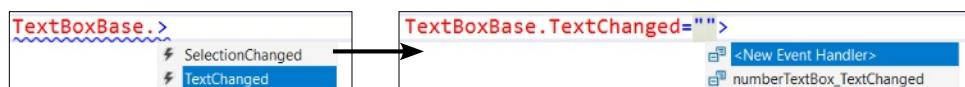
An editable combo box is like a cross between a ComboBox and a TextBox. You can choose items from a list, but you can also type in your own text. Since it works like a TextBox, we can add a PreviewTextInput event handler to make sure the user can only type numbers, just like we did with the TextBox. 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.



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.



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**.

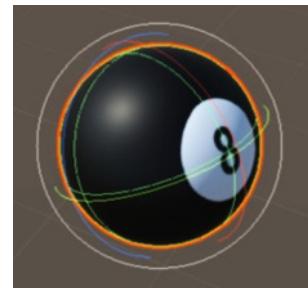
In this first lab, you'll get up and running with Unity. You'll get oriented with the Unity editor, and you'll start creating and manipulating 3D shapes.

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.

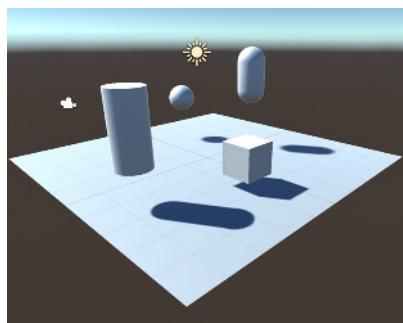


A powerful 2D and 3D scene editor

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**.

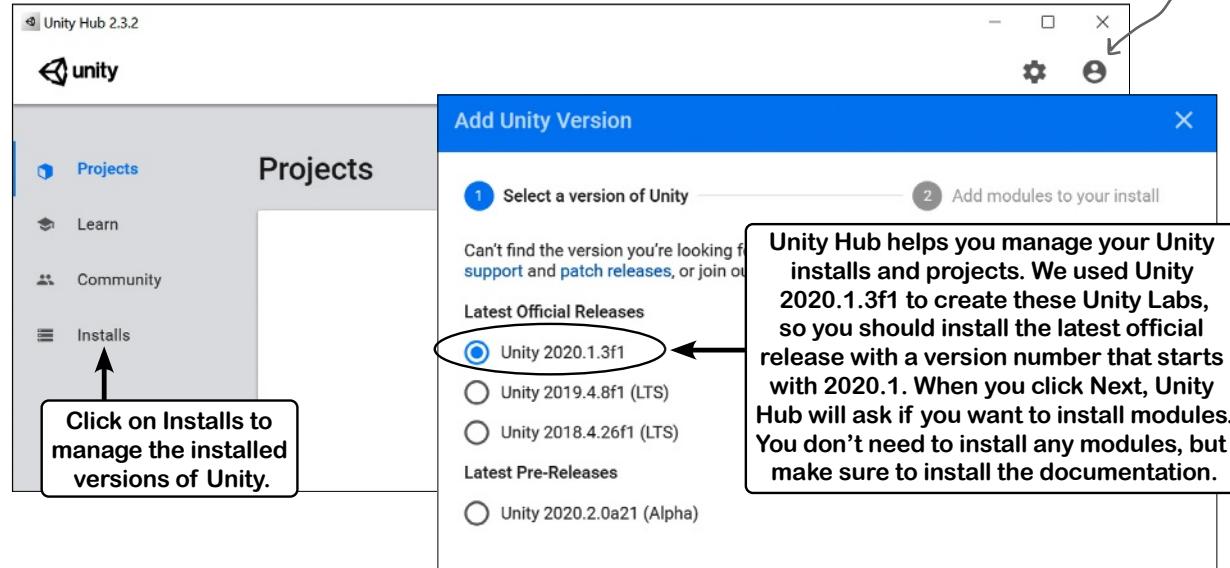
Unity Lab #1

Explore C# with Unity

Download Unity Hub

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.

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 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.



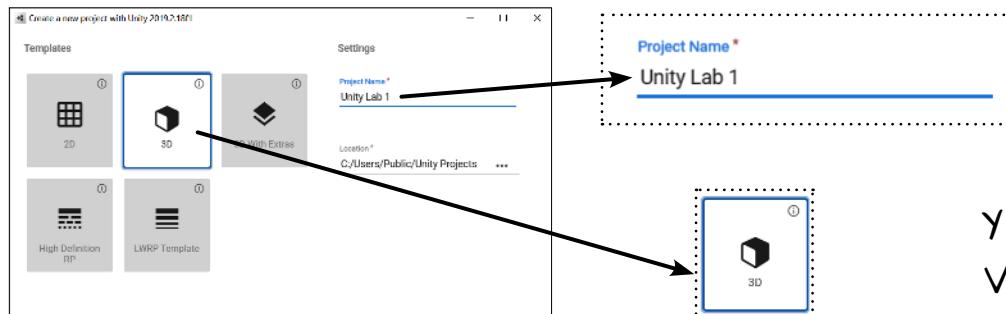
Watch it!

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).



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

Make Visual Studio your Unity script editor

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

*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).*



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

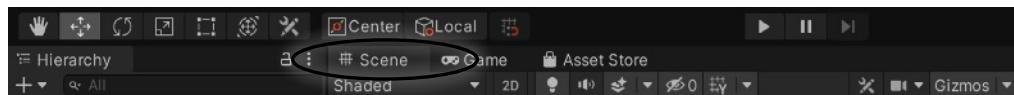
Unity Lab #1

Explore C# with Unity

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.

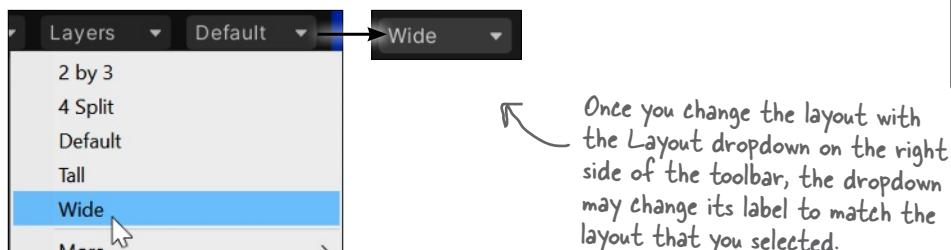
Find the Scene tab near the top of the window. Click on the tab and drag it to detach the window:



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

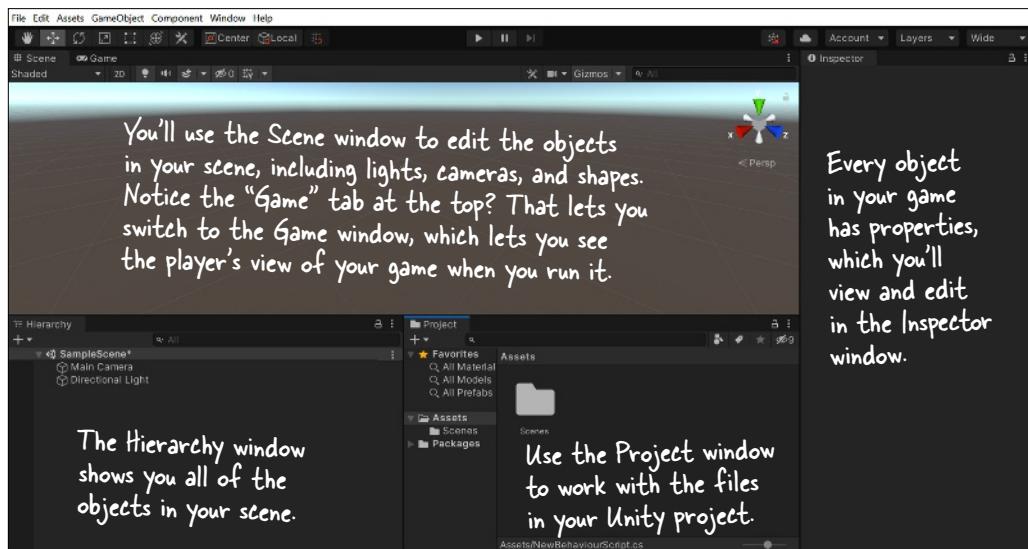
We've chosen the Wide layout because it works well for the screenshots in these labs. Find the Layout dropdown and choose Wide so your Unity editor looks like ours.



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.

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.

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



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.

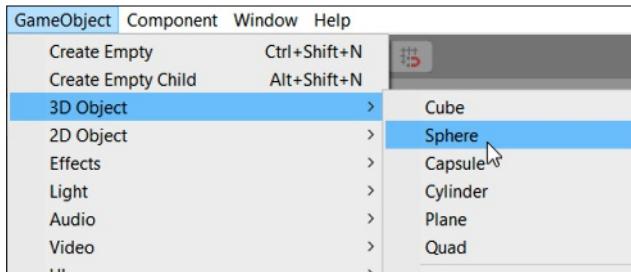
The Hierarchy window shows you all of the objects in your scene.

Use the Project window to work with the files in your Unity project.

Your scene is a 3D environment

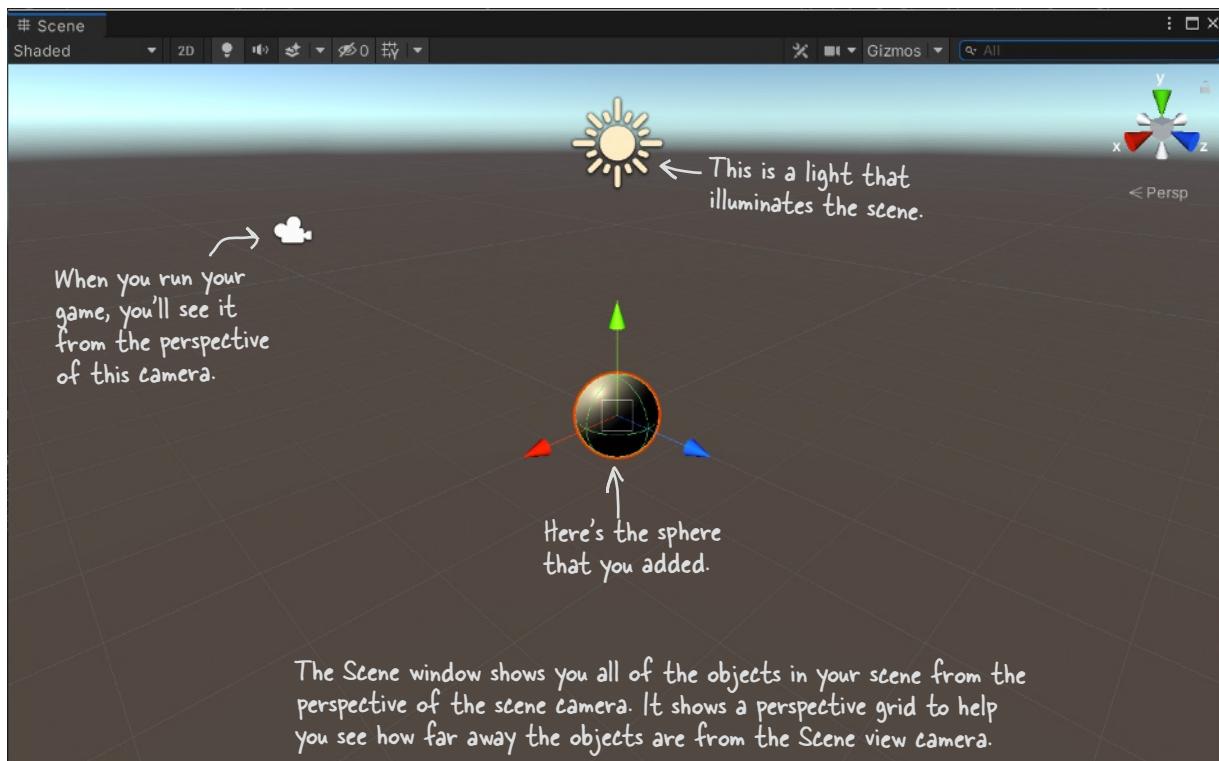
As soon as you start the editor, you're editing a **scene**. You can think of scenes as levels in your Unity games. Every game in Unity is made up of one or more scenes. Each scene contains a separate 3D environment, with its own set of lights, shapes, and other 3D objects. When you created your project,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

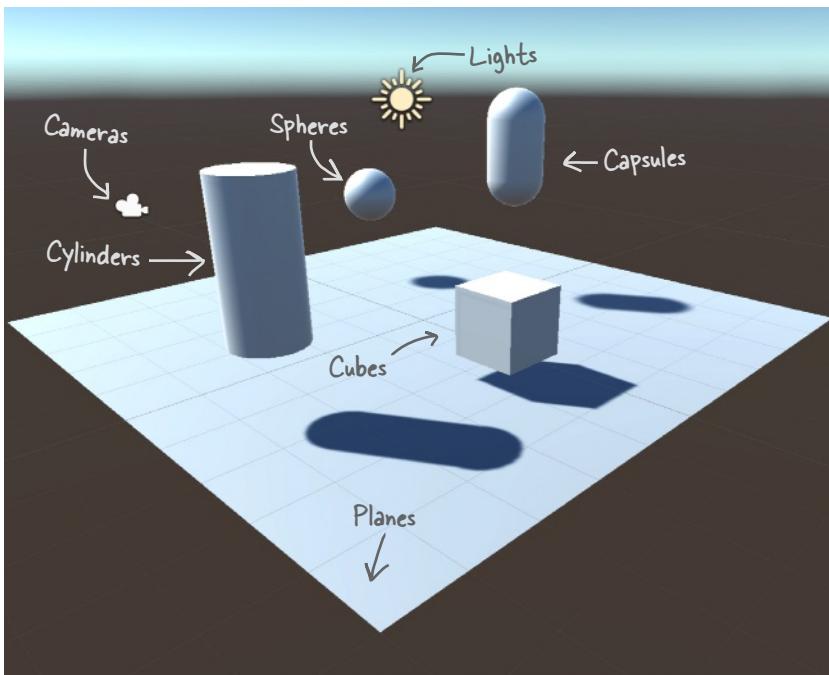
A sphere will appear in your Scene window. Everything you see in the Scene window is shown from the perspective of the **Scene view camera**, which "looks" at the scene and captures what it sees.



Unity games are made with GameObjects

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

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



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

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

ren-der, verb.

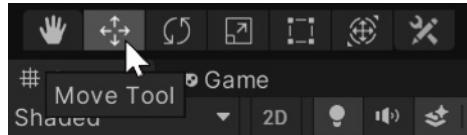
to represent or depict artistically.

*Michelangelo **rendered** his favorite model with more detail than he used in any of his other drawings.*

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

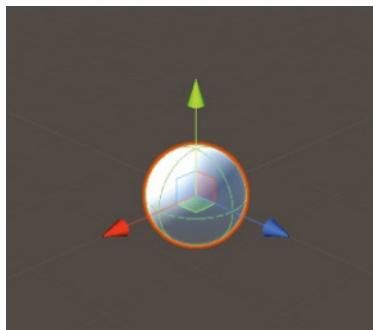
Use the Move Gizmo to move your GameObjects

The 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.



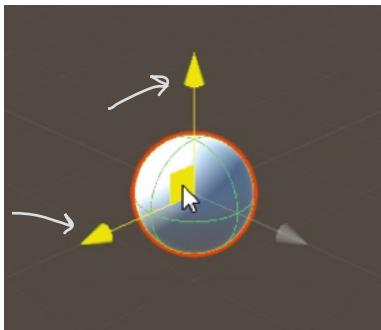
The buttons on the left side of the toolbar let you choose Transform Tools like the Move tool, which displays the Move Gizmo as arrows and a cube on top of the GameObject that's currently selected.

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.



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

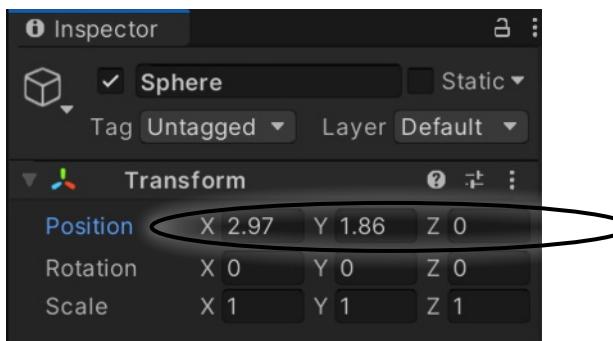
Move your sphere around the scene to get a feel for how the Move Gizmo works. Click and drag each of the three arrows to drag it along each plane individually. Try clicking on each of the faces of the cube in the Scene Gizmo to drag it around all three planes. Notice how the sphere gets smaller as it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

The Inspector shows your GameObject's components

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

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

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

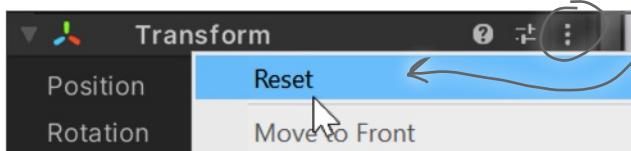


Did you notice the grid in your 3D space? As you're dragging the sphere around, hold down the Control key. That causes the GameObject that you're moving to snap to the grid. You'll see the numbers in the Transform component move by whole numbers instead of small decimal increments.

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.

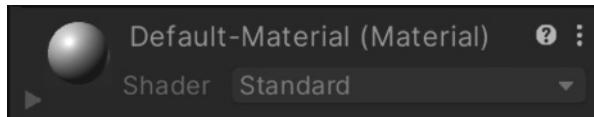
Save your scene often! Use File >> Save or Ctrl+S / ⌘S to save the scene right now.

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:



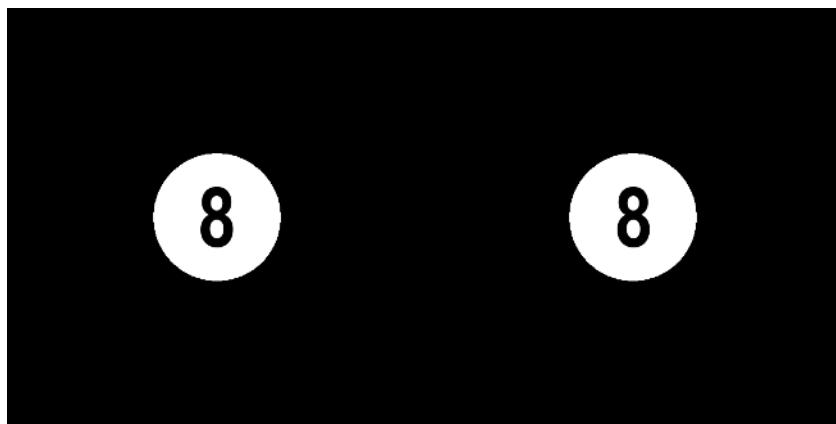
We'll make your sphere more interesting by adding a **texture**—that's just a simple image file that's wrapped around a 3D shape, almost like you printed the picture on a rubber sheet and stretched it around your object.

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.



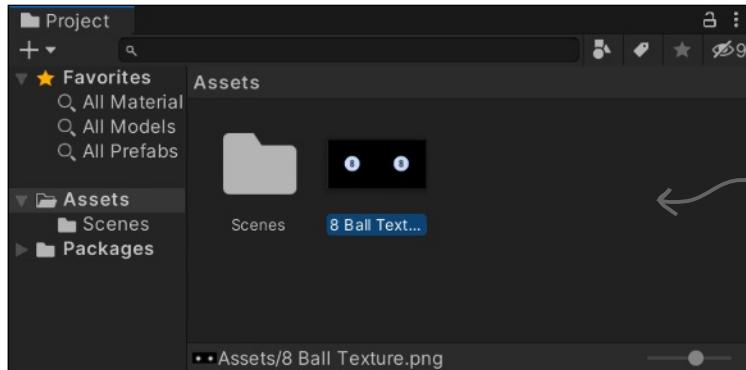
Download the file into a folder on your computer.

(*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.

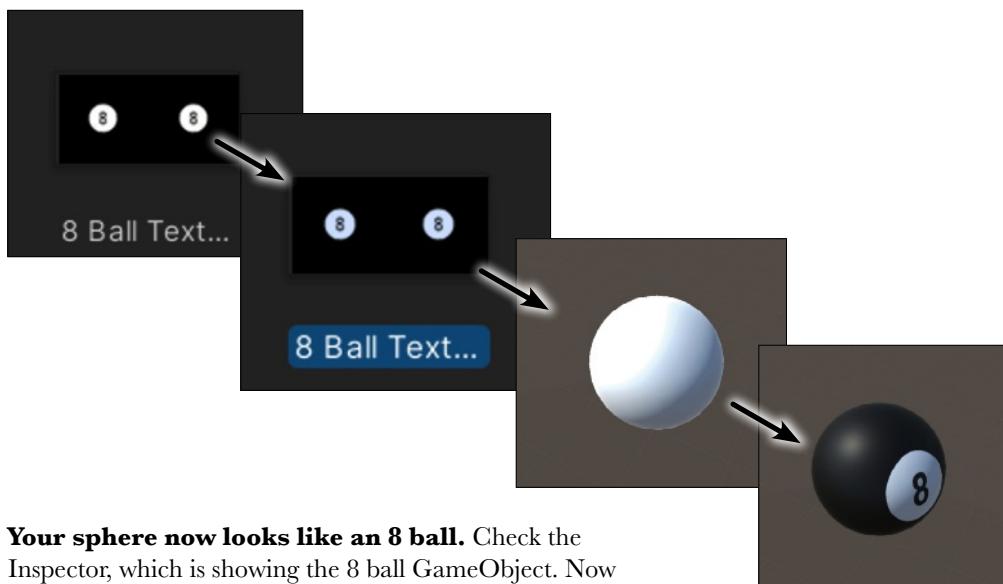


You right-clicked inside the Assets folder in the Project window to import the new asset, so Unity imported the texture into that folder.

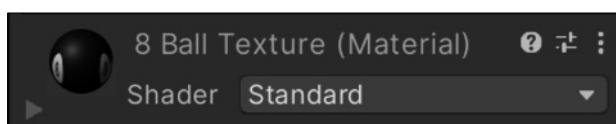
⑤

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.

Unity Lab #1

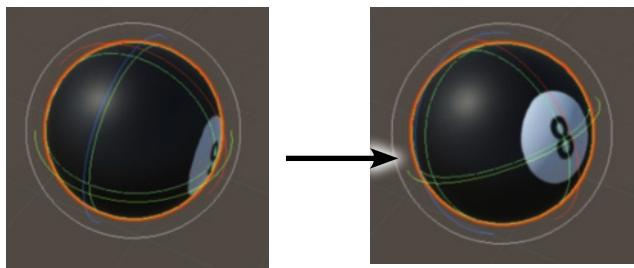
Explore C# with Unity

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.

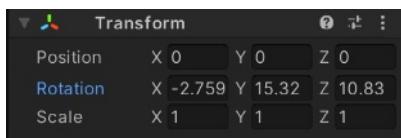


- 1 Click on the sphere. Unity will display a wireframe sphere Rotate Gizmo with red, blue, and green circles. Click the red circle and drag it to rotate the sphere around the X axis.



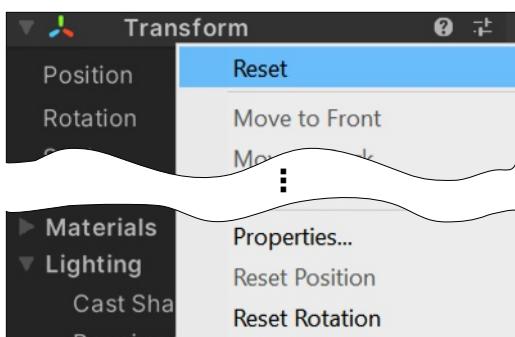
- 2 Click and drag the green and blue circles to rotate around the Y and Z axes.

The outer white circle rotates the sphere along the axis coming out of the Scene view camera.
Watch the Rotation numbers change in the Inspector window.



- 3 Open the context menu of the Transform panel in the Inspector window. Click

Reset, just like you did before. It will reset everything in the Transform component back to default values—in this case, it will change your sphere's rotation back to [0, 0, 0].



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

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

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



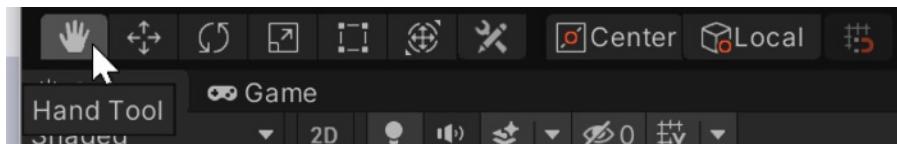
It's easy to reset your windows and scene camera.

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.

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

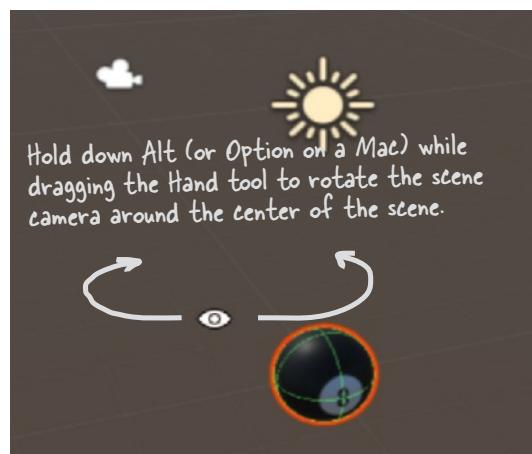
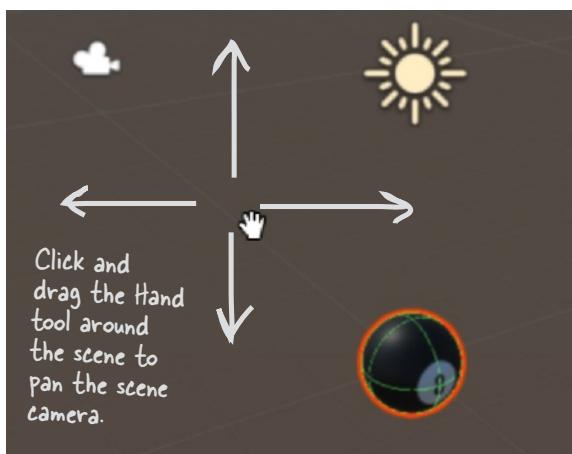
Use the mouse scroll wheel or scroll feature on your trackpad to zoom in and out, and toggle between the Move and Rotate Gizmos. Notice that the sphere changes size, but the Gizmos don't. The Scene window in the editor shows you the view from a virtual **camera**, and the scroll feature zooms that camera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

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



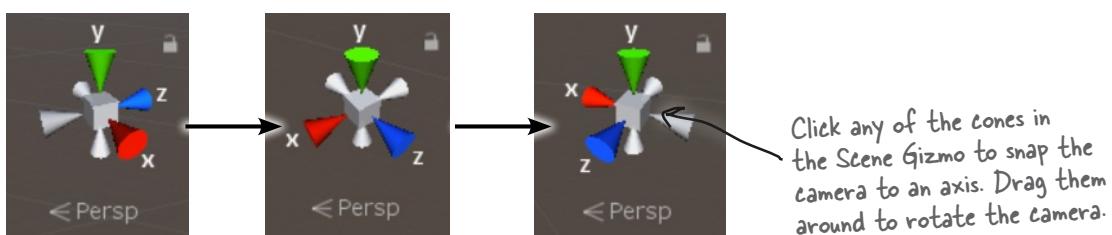
Hold down Alt (or Option on a Mac) while dragging and the Hand tool turns into an eye and rotates the view around the center of the window

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.



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.



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

Unity Lab #1

Explore C# with Unity

there are no
Dumb Questions

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

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

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

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

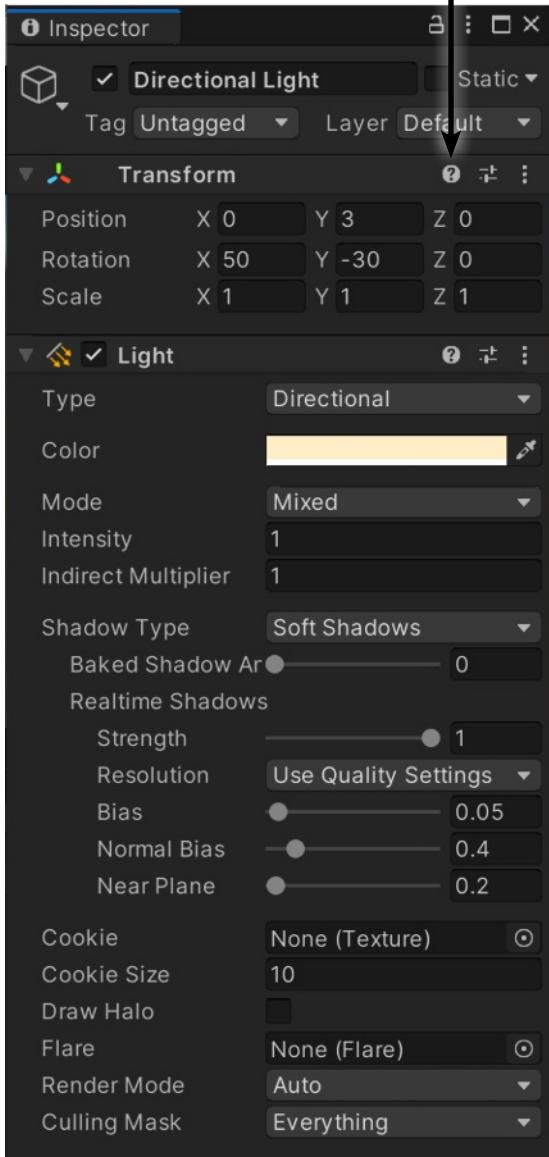
Q: If I add a Light component to any GameObject, does it become a light?

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

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

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

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



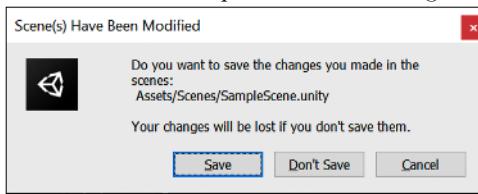
When you click on the Directional Light GameObject in the Hierarchy window, the Inspector shows you its components. It just has two: a Transform component that provides its position and rotation and a Light component that actually casts the light.

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?

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



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

BULLET POINTS

- The **Scene view** is your main interactive view of the world that you're creating.
- 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!

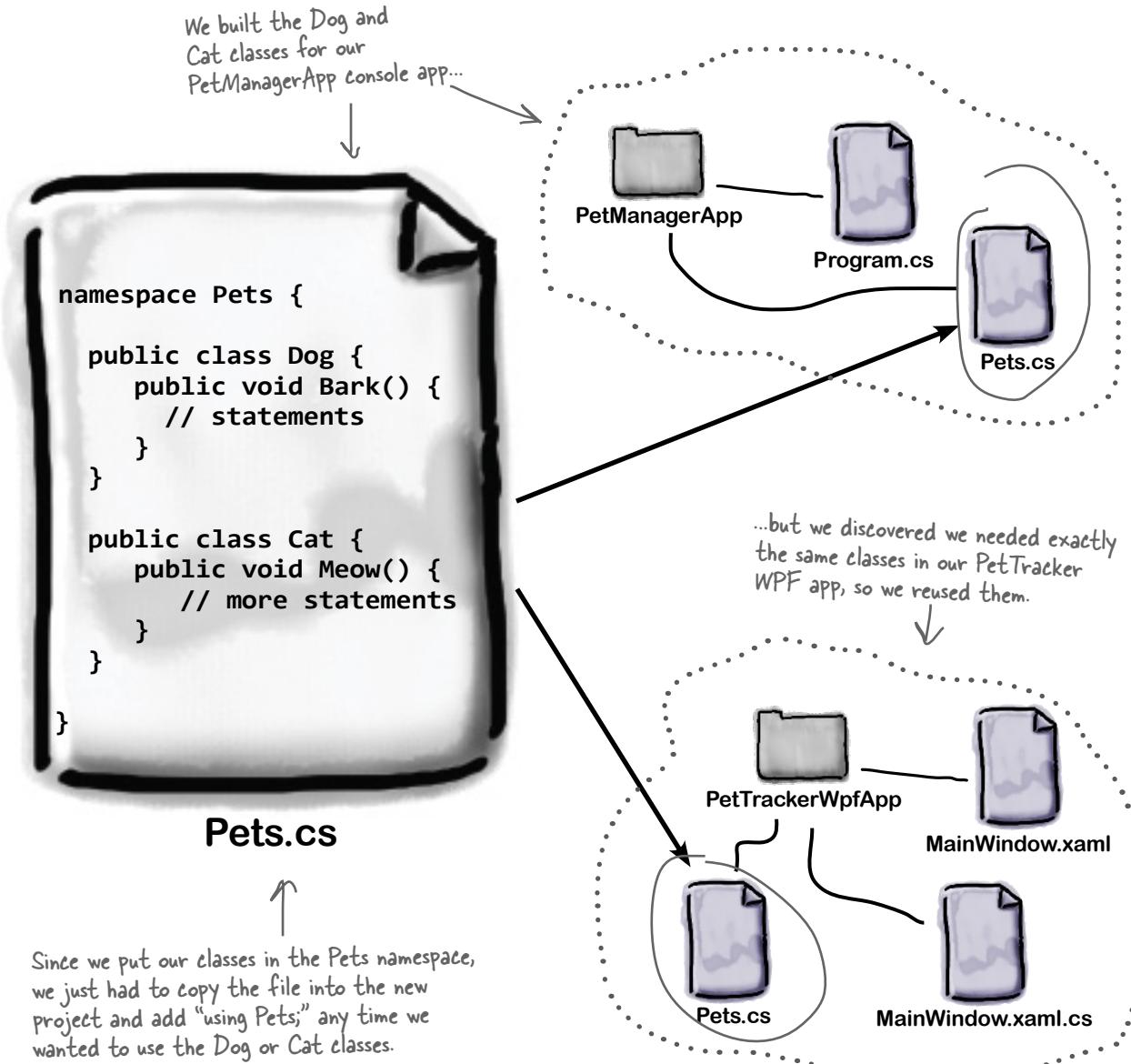


Every program you write solves a problem.

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.



Some methods take parameters and return a value

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



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

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

```

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

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! →

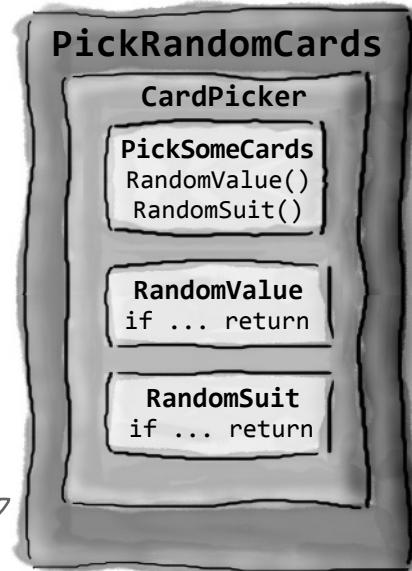
Since you're about to create methods that return values, right now is a perfect time to write some code and use the debugger to *really dig into how the **return** statement works*.

- ★ 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.

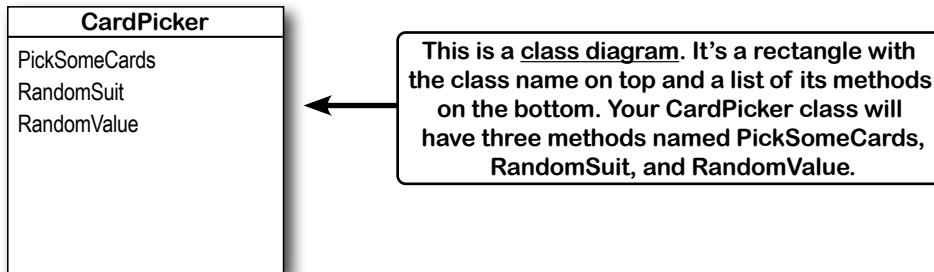


After your array is generated, you'll use a foreach loop to write it to the console.

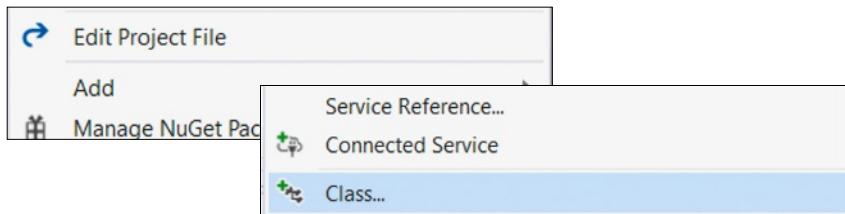
Create your PickRandomCards console app

Do this!

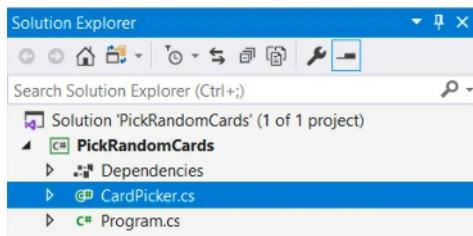
Let's use what you've learned so far to create a program that picks a number of random cards. Open Visual Studio and **create a new Console App project called PickRandomCards**. Your program will include a class called CardPicker. Here's a class diagram that shows its name and methods:



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:



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.
    } 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?

`return` returns immediately

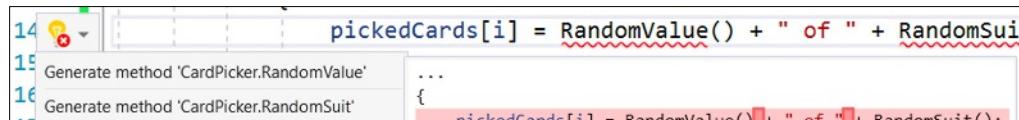
Finish your `PickSomeCards` method

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:



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";
}
```

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 immediately and go back to the statement that called it.

your class is done, now finish the app

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

    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";
    }
}
```



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.



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.

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.WriteLine

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.WriteLine 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?



Exercise Solution

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.WriteLine("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.

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:

The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar says "Microsoft Visual Studio Debug Console". The console output is as follows:

```
Microsoft Visual Studio Debug Console
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 really understand how this program works—this is a great opportunity to use the Visual Studio debugger to help you explore your code. Place a breakpoint on the first line of the Main method, then use Step Into (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.

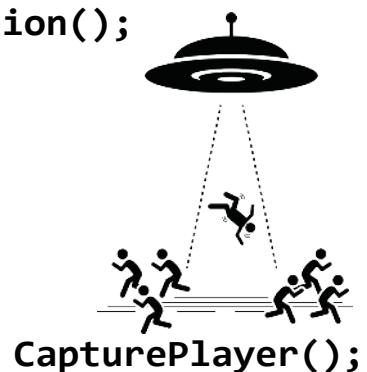


Ana's started working on some **prototypes**. She's been working on the code for the alien enemies that the player has to avoid in one exciting part of the game, where the player needs to escape from their hideout while the aliens search for them. Ana's written several methods that define the enemy behavior: 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();  
}
```



different games can act the same

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.



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



Prototypes

Game design... and beyond

A **prototype** is an early version of your game that you can play, test, learn from, and improve. A prototype can be a really valuable tool to help you make changes early. Prototypes are especially useful because they let you rapidly experiment with a lot of different ideas before you've made permanent decisions.

- The first prototype is often a **paper prototype**, where you lay out the core elements of the game on paper. For example, you can learn a lot about your game by using sticky notes or index cards for the different elements of the game, and drawing out levels or play areas on large pieces of paper to move them around.
- One good thing about building prototypes is that they help you **get from an idea to a working, playable game** very quickly. You learn the most about a game (or any kind of program) when you get working software into the hands of your players (or users).
- Most games will go through **many prototypes**. This is your chance to try out lots of different things and learn from them. If something doesn't go well, think of it as an experiment, not a mistake.
- Prototyping is a **skill**, and just like any other skill, **you get better at it with practice**. Luckily, building prototypes is also fun, and a great way to get better at writing C# code.

Prototypes aren't just used for games! When you need to build any kind of program, it's often a great idea to build a prototype first to experiment with different ideas.



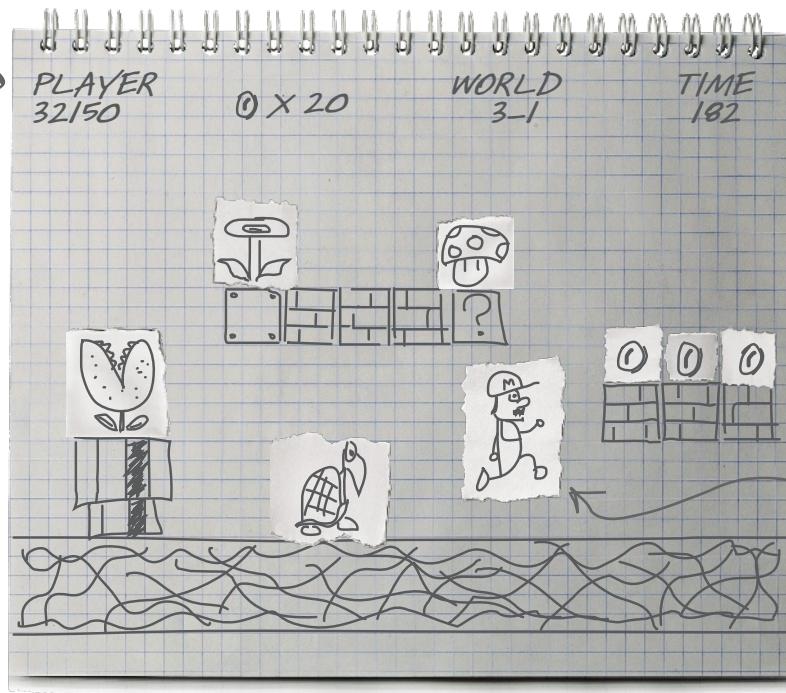
Build a paper prototype for a classic game

Paper prototypes are really useful for helping you figure out how a game will work before you start building it, which can save you a lot of time. There's a fast way to get started building them—all you need is some paper and a pen or pencil. Start by choosing your favorite classic game. Platform games work especially well, so we chose one of the **most popular, most recognizable** classic video games ever made... but you can choose any game you'd like! Here's what to do next.



- 1 Draw the background on a piece of paper.** Start your prototype by creating the background. In our prototype, the ground, bricks, and pipe don't move, so we drew them on the paper. We also added the score, time, and other text at the top.
- 2 Tear small scraps of paper and draw the moving parts.** In our prototype, we drew the characters, the piranha plant, the mushroom, the fire flower, and the coins on separate scraps. If you're not an artist, that's absolutely fine! Just draw stick figures and rough shapes. Nobody else ever has to see this!
- 3 "Play" the game.** This is the fun part! Try to simulate player movement. Drag the player around the page. Make the non-player characters move too. It helps to spend a few minutes playing the game, then go back to your prototype and see if you can really reproduce the motion as closely as possible. (It will feel a little weird at first, but that's OK!)

The text at the top of the screen is called the **HUD**, or head-up display. It's usually drawn on the background in a paper prototype.



When the player catches a mushroom he grows to double his size, so we also drew a small character on a separate scrap of paper.

The ground, bricks, and pipe don't move, so we drew them on the background paper. There's no rule about what goes on the background and what moves around.

The mechanics of how the player jumps were really carefully designed. Simulating them in a paper prototype is a valuable learning exercise.



PAPER PROTOTYPES LOOK LIKE THEY'D BE USEFUL FOR MORE THAN JUST GAMES. I BET I CAN USE THEM IN MY OTHER PROJECTS, TOO.

All of the tools and ideas in “Game design... and beyond” sections are important 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.



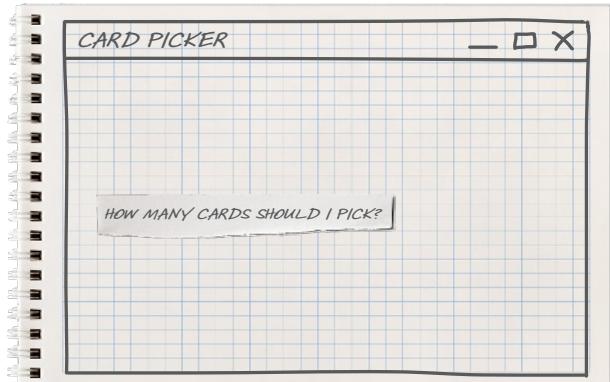
Yes! A paper prototype is a great first step for any project.

If you’re building a desktop app, a mobile app, or any other project that has a user interface, building a paper prototype is a great way to get started. Sometimes you need to create a few paper prototypes before you get the hang of it. That’s why we started with a paper prototype for a classic game...because that’s a great way to learn how to build paper prototypes. **Prototyping is a really valuable skill for any kind of developer**, not just a game developer.

Sharpen your pencil

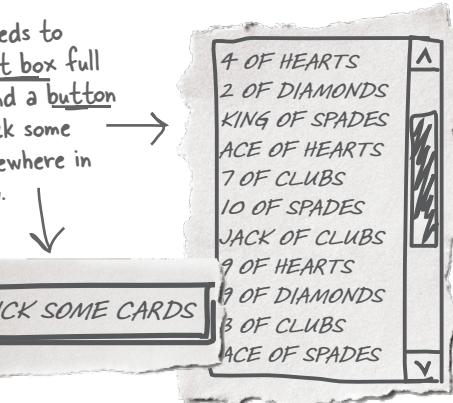
In the next project, you’ll create a WPF app that uses your CardPicker class to generate a set of random cards. In this paper-and-pencil exercise, you’ll build a paper prototype of your app to try out various design options.

Start by drawing the window frame on a large piece of paper and a label on a smaller scrap of paper.



Your app needs to include a list box full of cards and a button labeled “Pick some cards” somewhere in the window.

PICK SOME CARDS

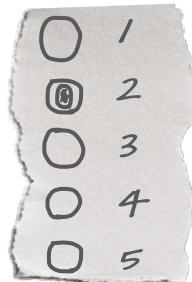


Next, draw a bunch of different types of controls on more small scraps of paper. Drag them around the window and experiment with ways to fit them together. What design do you think works best? There’s no single right answer—there are lots of ways to design any app.

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.

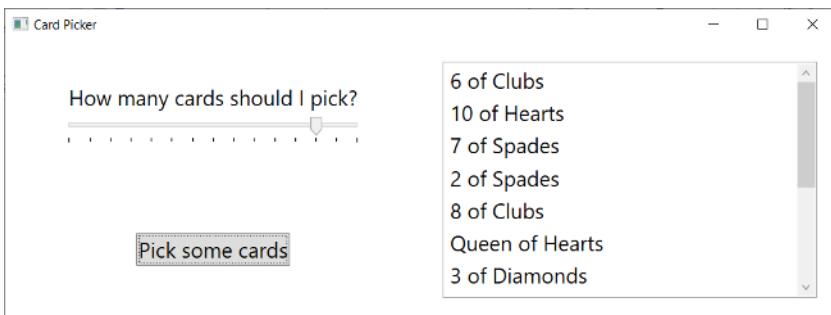


Try a slider and radio buttons, too. Can you think of other controls that you’ve used to input numbers into apps before? Maybe a dropdown box? Get creative!



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:



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 diagram illustrates the layout of the "Card Picker" window. It is a 2x2 grid of cells. The top-left cell contains a label and a slider. The bottom-left cell contains a button. The top-right cell is a large ListBox spanning two rows and centered with a margin of 20. The bottom-right cell is a callout box containing text about the ListBox. A callout box at the bottom also describes the button's function. A note on the left explains the combined control in the top-left cell. A note on the right discusses design choices. A callout at the bottom points to a guide for Mac users.

This cell has two controls, a Label and a Slider. We'll take a closer look at how that works.

The window has two rows and two columns. The ListBox in the right column spans both rows.

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.

There are ASP.NET Core versions of all of the WPF projects in this book that feature screenshots from Visual Studio for Mac.

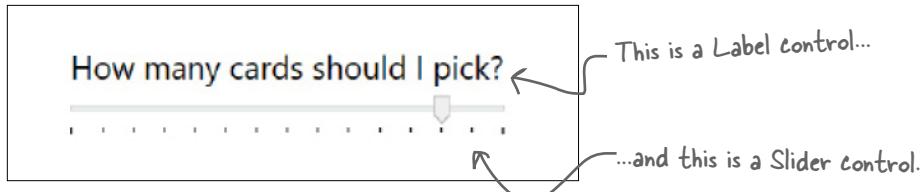
We won't keep reminding you to add your projects to source control—but we still think it's a really good idea to create a GitHub account and publish all of your projects to it!

Go to the Visual Studio for Mac Learner's Guide for the Mac version of this project.

Add to Source Control

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:

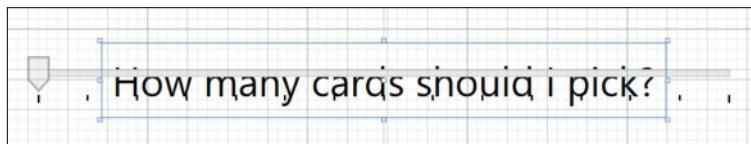


So how do we stack them on top of each other like that? We **could** try putting them in the same cell in the grid:

```
<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:

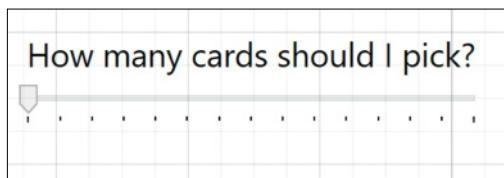


That's where a **StackPanel control** comes in handy. A StackPanel is a container control—like a Grid, its job is to contain other controls and make sure they go in the right place in the window. While the Grid lets you arrange controls in rows and columns, a StackPanel lets you arrange controls **in 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!

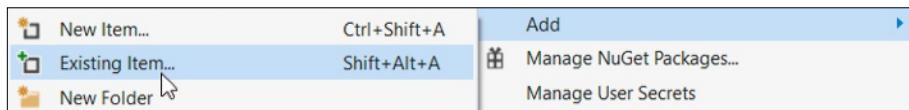
① Create a new WPF app called PickACardUI.

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.

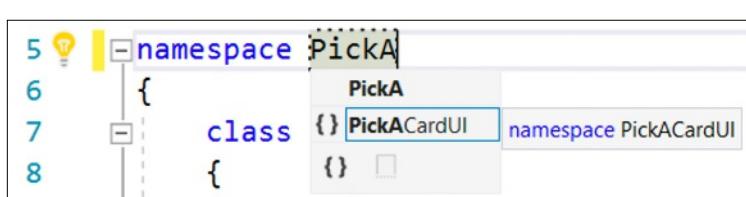
Right-click on the project name and choose **Add >> Existing Item...** from the menu.



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**:



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>
```

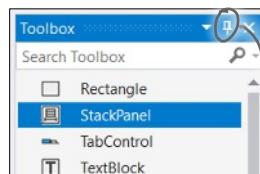
You can use the Visual Studio designer to add two equal rows and two equal columns. If you run into trouble, you can just type the XAML directly into the editor.

- 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**.



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.**



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 .

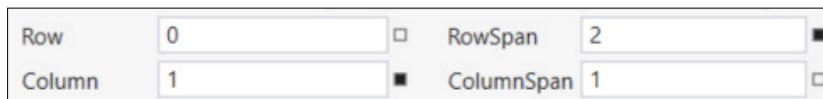
The XAML for your Button control should look like this:

```
<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:



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:

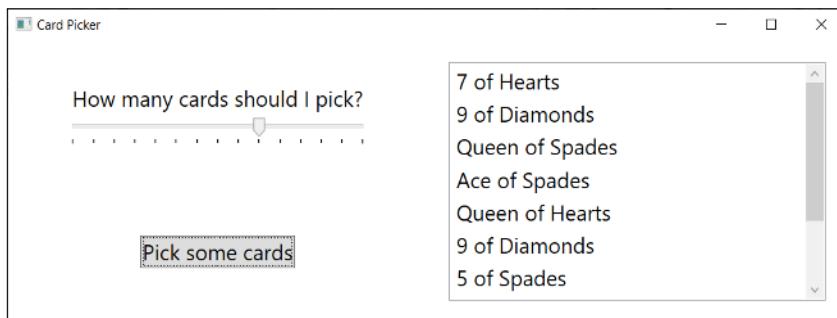
```
Title="Card Picker" Height="300" Width="800"
```

6 Add a Click event handler to your Button control.

The **code-behind**—the 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);
    }
}
```

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!**



The C# code joined to your XAML window that contains the event handlers is called the code-behind.

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.

Enemy
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer

...but what if she wants more than one enemy?

And that's great...until Ana wants more than one enemy, which is all there was in each of her early prototypes. What should she do to add a second or third enemy to her game?

Ana *could* copy the Enemy class code and paste it into two more class files. Then her program could use methods to control three different enemies at once. Technically, we're reusing the code...right?

Hey Ana, what do you think of that idea?

She has a point. What if she wants a level with, say, dozens of zombies? Creating dozens of identical classes just isn't practical.

Enemy1
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer

Enemy2
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer

Enemy3
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer



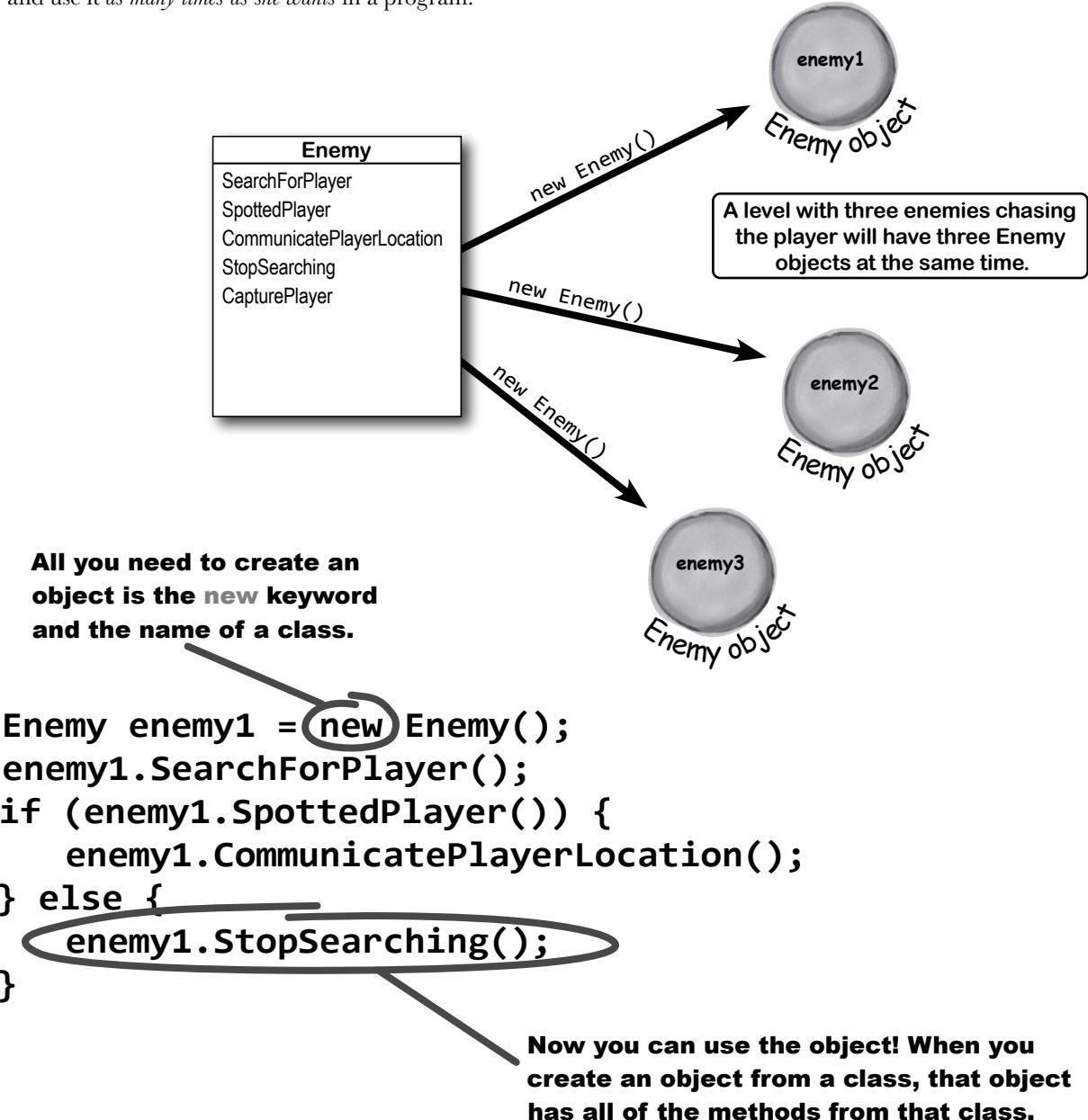
ARE YOU JOOKING?
USING SEPARATE IDENTICAL
CLASSES FOR EACH ENEMY IS A
TERRIBLE IDEA. WHAT IF I WANT
MORE THAN THREE ENEMIES AT
ONCE?

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.



You use a class to build an object

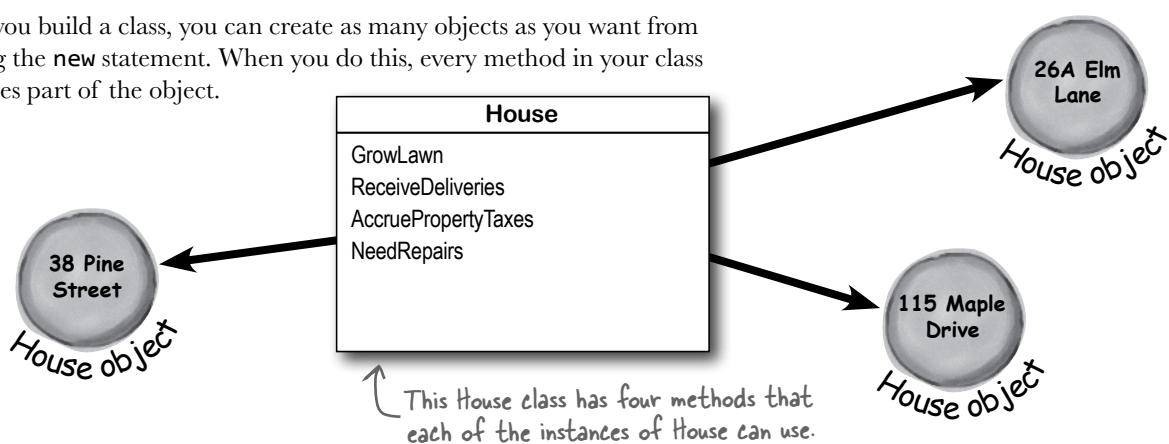
A class is like a blueprint for an object. If you wanted to build five identical houses in a suburban housing development, you wouldn't ask an architect to draw up five identical sets of blueprints. You'd just use one blueprint to build five houses.



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.

Before: here's a picture of your computer's memory when your program starts.



Your program executes a new statement.

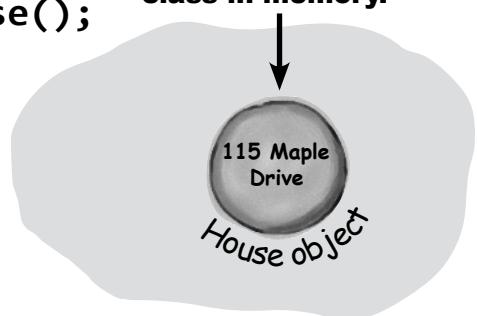
```
House mapleDrive115 = new House();
```

This new statement creates a new House object and assigns it to a variable called mapleDrive115.



THAT NEW KEYWORD LOOKS FAMILIAR. I'VE SEEN THIS SOMEWHERE BEFORE, HAVEN'T I?

After: now it has an instance of the House class in memory.



Yes! You've already created instances in your own code.

Go back to your animal matching program and look for this line of code:

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

You created an instance of the Random class, and then you called its Next method. Now look at your CardPicker class and find the **new** statement. You've been using objects this whole time!

A better solution for Ana...brought to you by objects

Ana used objects to reuse the code in the **Enemy** class without all that messy copying that would've left duplicate code all over her project. Here's how she did it.

- 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];
```

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.

Hmm, this array is inside the class, but outside of the methods. What do you think is going on?

- 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.

Enemy
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer

```
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.

- 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 }
```

The **foreach** loop iterates through the array of **Enemy** objects.

```
foreach (Enemy enemy in enemyArray)
{
    // code that calls the Enemy methods
}
```

When you create a new instance of a class, it's called instantiating that class.



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();`

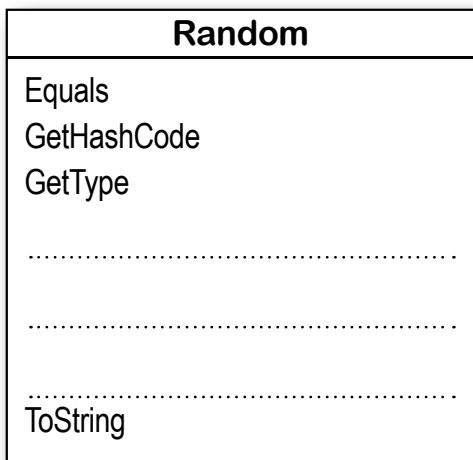
When we’re introducing a new concept (like objects), keep your eyes open for pictures and code excerpts like these.

A circular graphic with a grey gradient background. Inside the circle, the text "115 Maple Drive" is written vertically, and "House object" is written diagonally across the bottom right.



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.



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 =
```

```
}
```

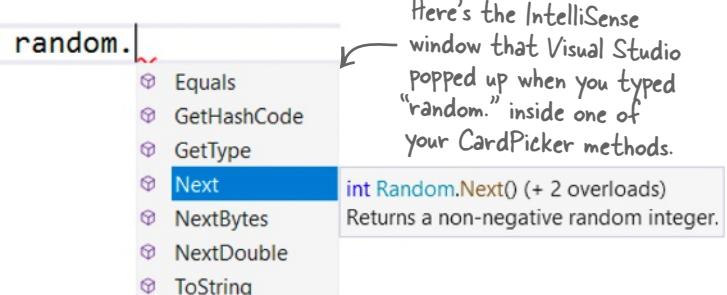
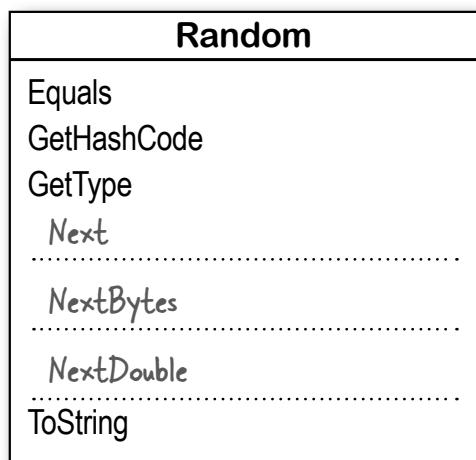
We filled in part of the code, including the curly braces. Your job is to finish those statements and then write the rest of the code.



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 (cube). We filled in some of the methods. Finish filling in the class diagram for the Random class.



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

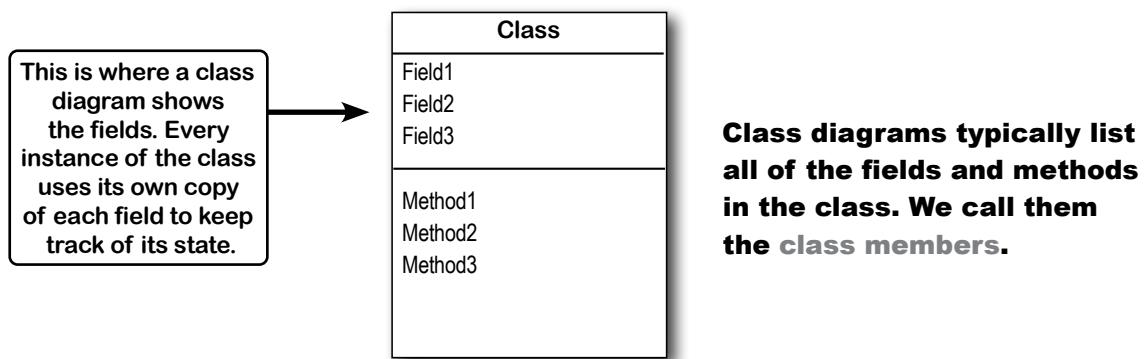
You've seen how classes can contain fields as well as methods. We just saw how you used the `static` keyword to declare a field in your CardPicker class:

```
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.

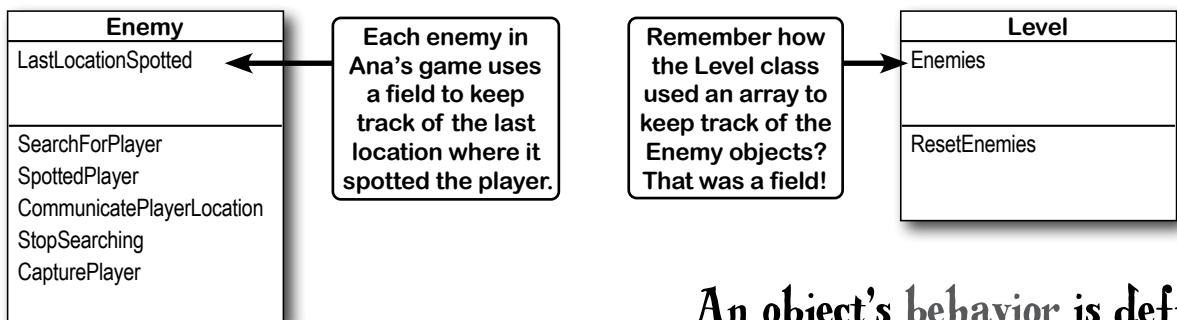
When we want to include fields a class diagram, we'll draw a horizontal line in the box. The fields go above the line, and methods go below the line.

Sometimes people think the word "instantiate" sounds a little weird, but it makes sense when you think about what it means.



Methods are what an object does. Fields are what the object knows.

When Ana's prototype created three instances of her Enemy class, each of those objects was used to keep track of a different enemy in the game. Every instance keeps separate copies of the same data: setting a field on the enemy2 instance won't have any effect on the enemy1 or enemy3 instances.



An object's behavior is defined by its methods, and it uses fields to keep track of its state.

static means a single shared copy

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?

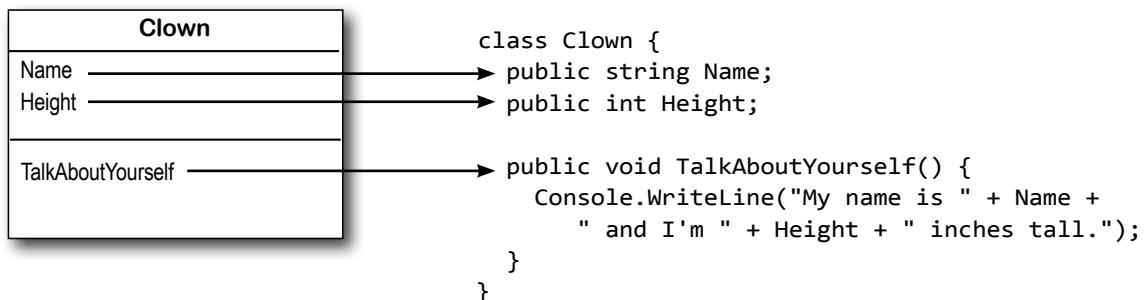
A: Because if you have an object that's keeping track of certain data—like Ana's instances of her Enemy class that each kept track of different enemies in her game—then you can use each instance's methods to work with that data. So when Ana's game calls the StopSearching method on the enemy2 instance, 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.**



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:



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.

you are here ▶

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;           ← 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.
    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; ← This line uses the Height field of the
                                         old oneClown instance to set the Height
                                         field of the new clown3 instance.
    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.
}
```

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:

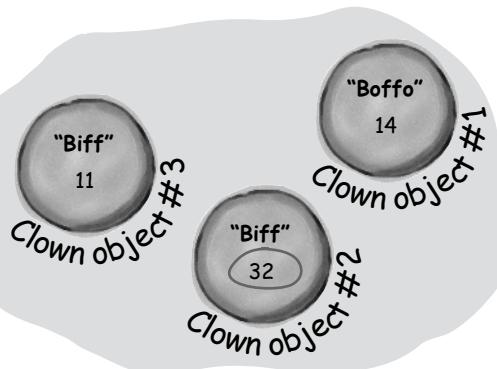
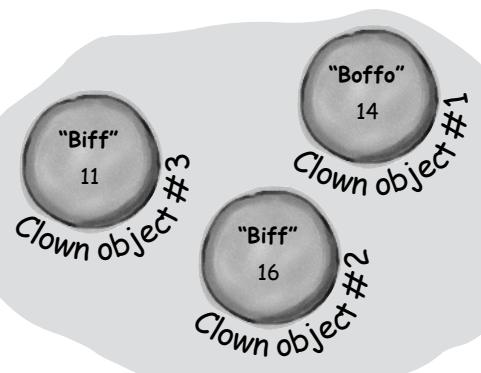
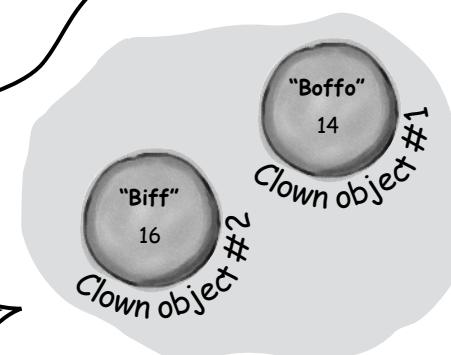
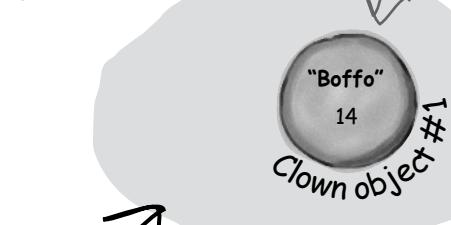
```
// These statements create an instance of
// Clown and then set its fields
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.TalkAboutYourself();
```

```
// 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();
```

This object is an instance of the Clown class.



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 everywhere.

Have you ever played with a Raspberry PI? It's a low-cost computer on a single board, and computers like it can be found inside all sorts of machinery. Thanks to Windows IoT (or Internet of Things), your C# code can run on them. 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.



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;  
    }  
}
```

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.

When we rename the class "CandyBarMaker" and the method "CheckNougatTemperature" it starts to make the code easier to understand.

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**.

TemperatureChecker

ThreeMinuteCheck
DoCICSVentProcedure
IsNougatTooHot

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.

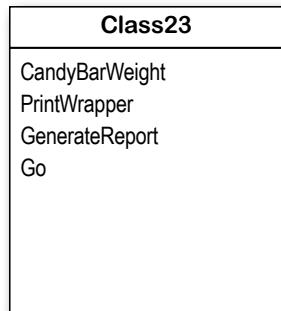
You can make your code easier to read and write by thinking about the problem your code was built to solve. If you choose names for your methods that make sense to someone who understands that problem, then your code will be a lot easier to decipher and develop. No matter how well we plan our code, we 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!).

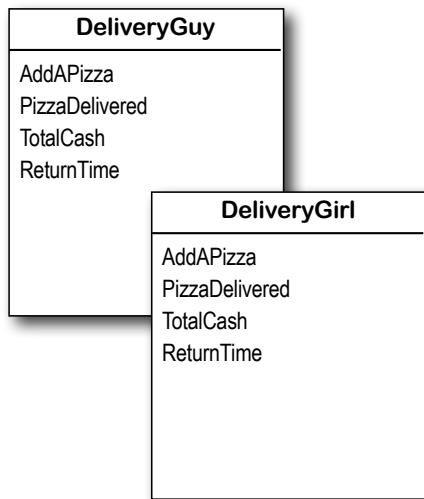


Each of these classes has a serious design flaw. Write down what you think is wrong with each class, and how you'd fix it.



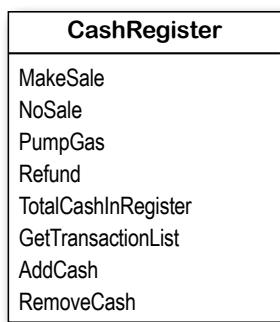
This class is part of the candy manufacturing system from earlier.

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



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.

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



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.

CandyMaker

CandyBarWeight
PrintWrapper
GenerateReport
MakeTheCandy

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.

DeliveryPerson

~~Gender~~
AddAPizza
PizzaDelivered
TotalCash
ReturnTime

We decided NOT to add a Gender field because there's actually no reason for this pizza delivery class to keep track of the gender of the people delivering pizza—and we should respect their privacy! Always look out for ways that bias can sneak into your code.

The CashRegister class is part of a program that's used by an automated convenience store checkout system.

All of the methods in the class do stuff that has to do with a cash register—making a sale, getting a list of transactions, adding cash...except for one: pumping gas. It's a good idea to pull that method out and stick it in another class.

CashRegister

MakeSale
NoSale
Refund
TotalCashInRegister
GetTransactionList
AddCash
RemoveCash

Code Tip: A few ideas for designing intuitive classes

We're about to jump back into writing code. You'll be writing code for the rest of this chapter, and a LOT of code throughout the book. That means you'll be **creating a lot of classes**. Here are a few things to keep in mind when you make choices about how to design them:

- ★ **You're building your program to solve a problem.**

Spend some time thinking about that problem. Does it break down into pieces easily? How would you explain that problem to someone else? These are good things to think about when designing your classes.

- ★ **What real-world things will your program use?**

A program to help a zookeeper track her animals' feeding schedules might have classes for different kinds of food and types of animals.

- ★ **Use descriptive names for classes and methods.**

Someone should be able to figure out what your classes and methods do just by looking at their names.

- ★ **Look for similarities between classes.**

Sometimes two classes can be combined into one if they're really similar. The candy manufacturing system might have three or four turbines, but there's only one method for closing the trip valve that takes the turbine number as a parameter.



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>.

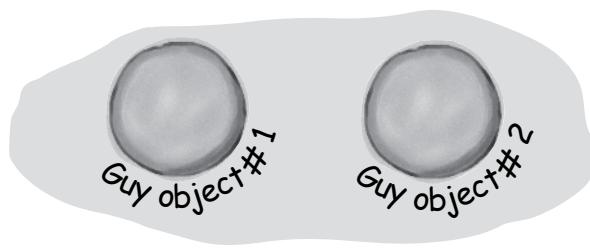
You can learn a lot from reading code. So if you run into a problem with a coding exercise, don't be afraid to peek at the solution. It's not cheating!

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.

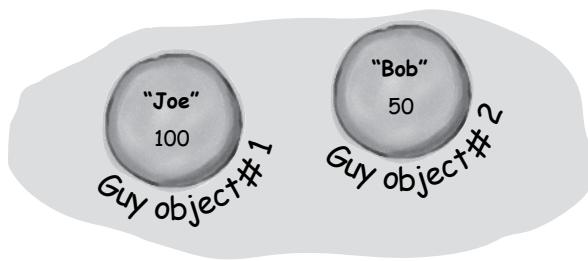
We'll use two Guy variables called `joe` and `bob` to keep track of each of our instances. Here's what the heap will look like after they're created:



Guy
Name Cash
WriteMyInfo GiveCash ReceiveCash

2 We'll set each Guy object's Cash and Name fields.

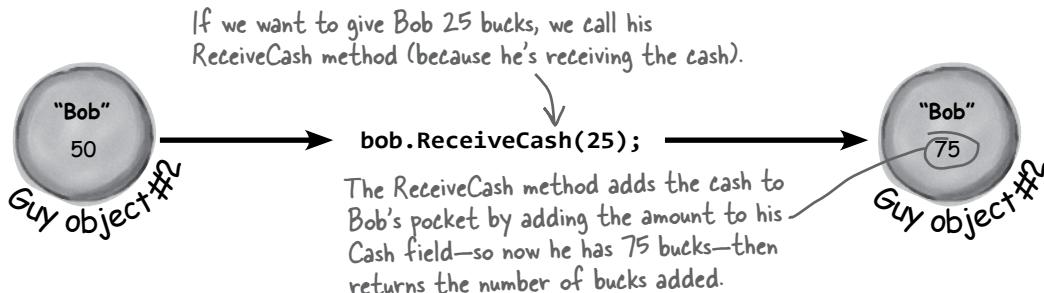
The two objects represent different guys, each with his own name and a different amount of cash in his pocket. Each guy has a Name field that keeps track of his name, and a Cash field that has the number of bucks in his pocket.



We chose names for the methods that make sense. You call a Guy object's GiveCash method to make him give up some of his cash, and his ReceiveCash method when you want to give cash to him (so he receives it).

3 We'll add methods to give and receive cash.

We'll make a guy give cash from his pocket (and reduce his Cash field) by calling his GiveCash method, which will return the amount of cash he gave. We'll make him receive cash and add it to his pocket (increasing his Cash field) by calling his ReceiveCash method, which returns the amount of cash he received.



```

class Guy
{
    public string Name;
    public int Cash; } The Name and Cash fields keep track of the
guy's name and how much cash he has in his pocket.

/// <summary>
/// Writes my name and the amount of cash I have to the console.
/// </summary>
public void WriteMyInfo()
{
    Console.WriteLine(Name + " has " + Cash + " bucks."); ← Sometimes you want to ask
an object to perform a task,
like printing a description
of itself to the console.

/// <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 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.

start your instance off right

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() { }  
Cash (field) int Guy.Cash  
Name
```

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, }  
Name (field) string Guy.Name
```

Set the Name field and add the semicolon. You now have a single statement that initializes your object:

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
```

This new declaration does the same thing as
the three lines of code at the top of the
page, but it's shorter and easier to read.

Now you have all of the pieces to build your console app that uses two instances of the Guy class. Here's 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.

```
C:\Program Files... — X
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 really get it.



Exercise (Part 2)

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
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.
```

These are the odds to beat.

The player makes a double-or-nothing bet each round.

The program picks a random double from 0 to 1. If the number is greater than the odds, the player wins twice their bet, otherwise the player loses.

Create a new console app 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

```
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.");
    }
}
```



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.

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.

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!



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.

The screenshot shows the C# Interactive (64-bit) window in Visual Studio. On the left, there's a code editor with some C# code. On the right, a terminal window is open with the following output:

```
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
>
```

Annotations on the right side of the terminal window explain the code execution:

- A brace groups the first three class definitions with the text: "Paste in each class. You'll see periods for each pasted line."
- An arrow points to the line `> Program.Main(new string[] {})` with the text: "Run the Main method to see the output. Press Ctrl+D to exit."
- A callout box contains the text: "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."
- An annotation on the right says: "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.



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

<u>ELLIWYNN</u>	
Character Name	7
Level	
<u>LAWFUL GOOD</u>	
Alignment	
<u>WIZARD</u>	
Character Class	
<input type="text" value="911"/>	Strength
<input type="text"/>	Dexterity
<input type="text" value="17"/>	Intelligence
<input type="text" value="15"/>	Wisdom
<input type="text" value="10"/>	Charisma

Picture

Players create characters by rolling dice for each of their ability scores, which they write in these boxes.

CharacterSheet

```
CharacterName
Level
PictureFilename
Alignment
CharacterClass
Strength
Dexterity
Intelligence
Wisdom
Charisma
SpellSavingThrow
PoisonSavingThrow
MagicWandSavingThrow
ArrowSavingThrow
```

```
ClearSheet
GenerateRandomScores
```

This box is for a picture of the character. If you were building a C# class for a character sheet, you could save that picture in an image file.

In the RPG that Owen plays, saving throws give players a chance to roll dice and avoid certain types of attacks. This character has a magic wand saving throw, so the player filled in this circle.



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.



**Better a witty fool,
than a foolish wit.**

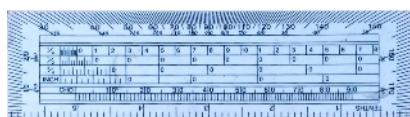


- ★ **string** can hold text of any length (including the empty string "").
- ★ **bool** is a Boolean value—it's either true or false. You'll use it to represent anything that only has two options; it can either be one thing or another, but nothing else.

- ★ **int** can store any **integer** from $-2,147,483,648$ to $2,147,483,647$. Integers don't have decimal points.



- ★ **double** can store **real** numbers from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ 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 $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ with up to 8 significant digits.



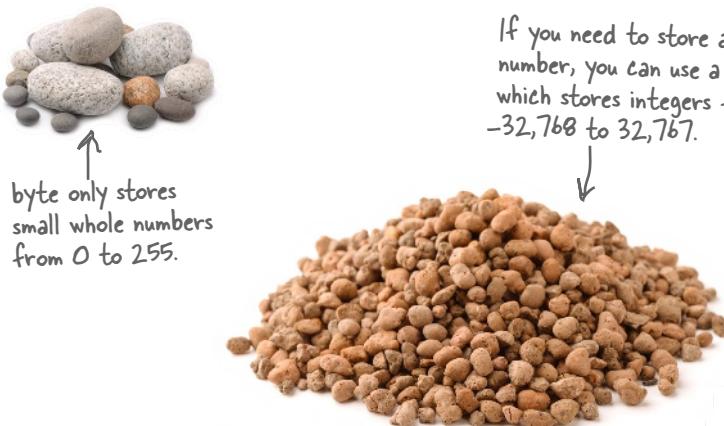
Why do you think C# has more than one type for storing numbers that have a decimal point?

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.



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 $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ with 6–9 significant digits.
- ★ **double** can store any number from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ with 15–17 significant digits.
- ★ **decimal** can store any number from $\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$ 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.

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);
}
```

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 $-0.00\dots$ (40 more zeros)...0015.

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**.



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;
```

```
C# Interactive (64-bit)
Type "#help" for more information.
> int i;
> i
0
> |
```

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.

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.WriteLine("Enter the number of cards to pick: ");  
if (value == 1) return "Ace";
```

So when you type `int i = 5;`, the 5 is a literal.

Can you spot all of the literals in these statements from code you've written in previous chapters? The last statement has two literals.

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.



```
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

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

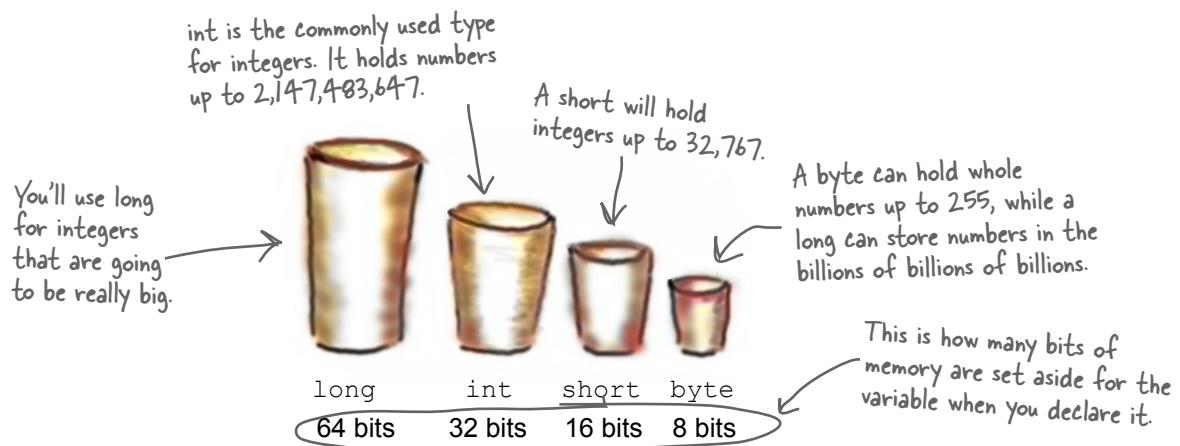
while loops are loops that keep on going as long as the condition at the beginning of the loop is true.

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.

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.

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.



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 11111111111111 (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("11111111111111", 2); // returns 32767
Convert.ToInt32("1111111111111111111111111111", 2); // returns 2147483647
```

bigger values take more memory

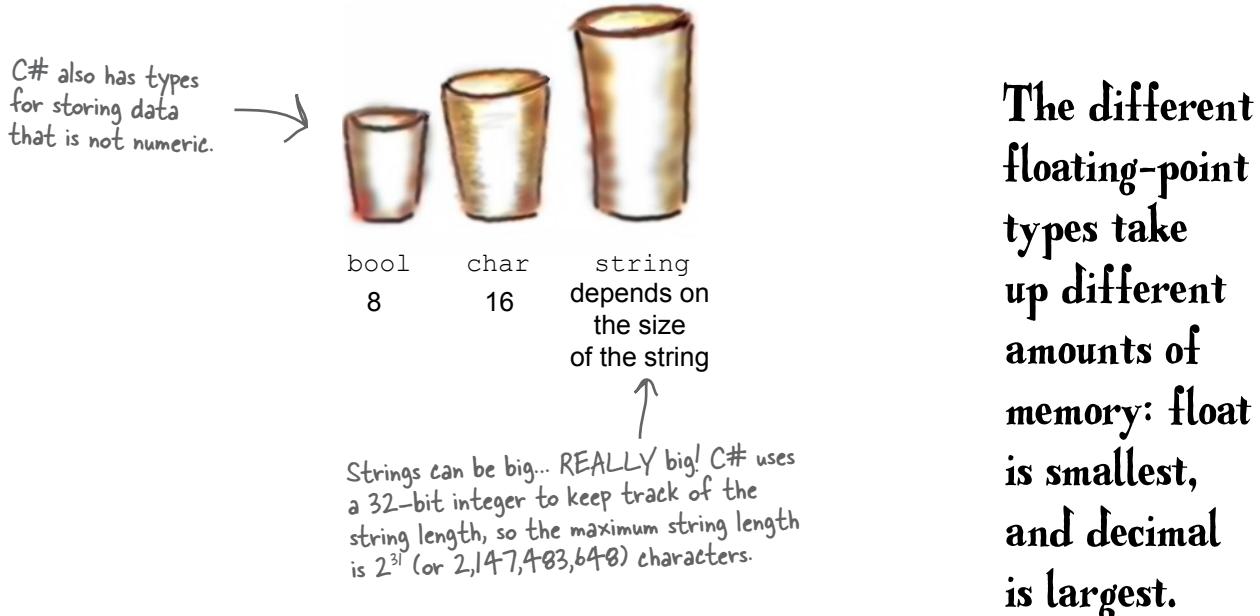
Other types come in different sizes, too

Numbers that have decimal places are stored differently than integers, and the different floating-point types take up different amounts of memory. You can handle most of your numbers that have decimal places using **float**, the smallest data type that stores decimals. If you need to be more precise, use a **double**. If you’re writing a financial application where you’ll be storing currency values, you’ll always want to use the **decimal** type.

Oh, and one more thing: **don’t use double for money or currency, only use decimal.**



We’ve talked about strings, so you know that the C# compiler also can handle **characters** and **non-numeric types**. The **char** type holds one character, and **string** is used for lots of characters “strung” together. There’s no set size for a string object—it expands to hold as much data as you need to store in it. The **bool** data type is used to store true or false values, like the ones you’ve used for your **if** statements.



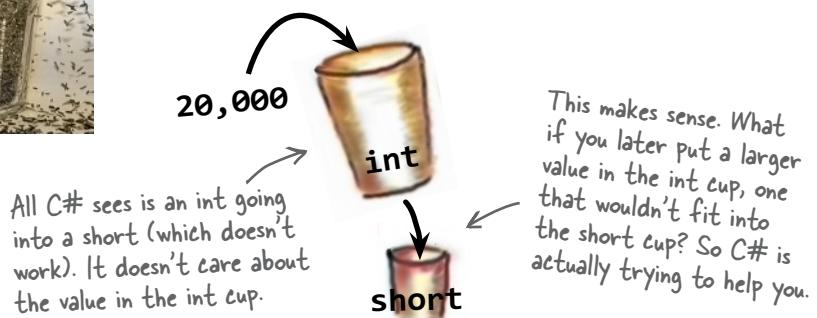
10 pounds of data in a 5-pound bag



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.



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!

bool isDone = yes;

You can only assign a value of “true” or “false” to a bool.

byte days = 365;

A byte can only hold a value between 0 and 255.
You'll need a short for this.

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

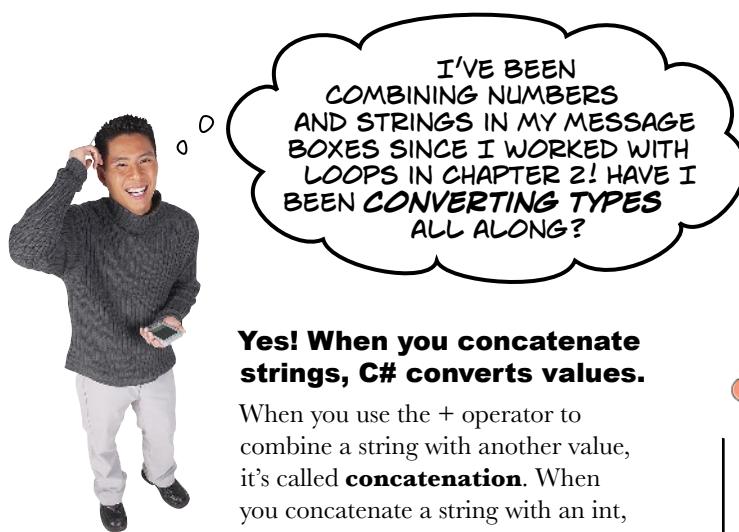
You can't always cast any type to any other type.

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

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;           The - operator subtracted
                           the short from the long, and
                           the = operator converted
                           the result to a double.
d = d / 123.456;
Console.WriteLine("The answer is " + d);
```

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 "1111111", and `Convert.ToString(8675309, 2)` returns the string "1000010001011111101101"—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'

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.

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.

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

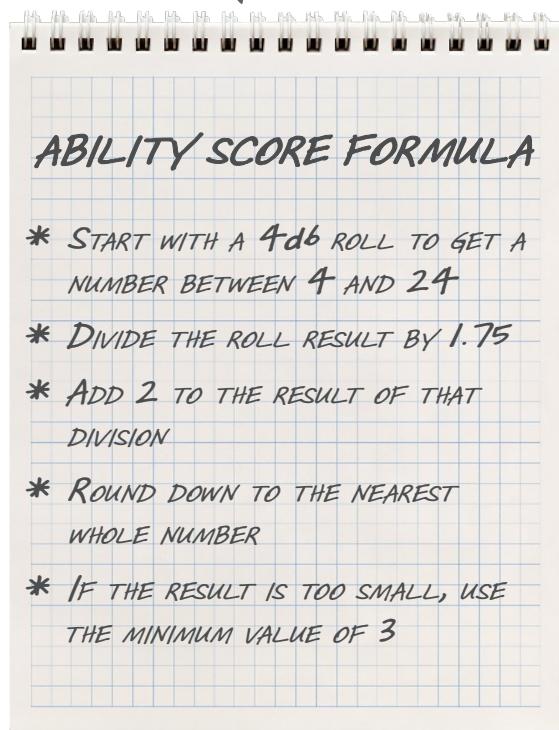
- There are **value types** for variables that hold different sizes of numbers. The biggest numbers should be of type `long` and the smallest ones (up to 255) can be declared as `bytes`.
- Every value type has a **size**, and you can't put a value of a bigger type into a smaller variable, no matter what 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 wants to improve his game

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.



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.



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!



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.

This is what it looks like when Owen runs the app:

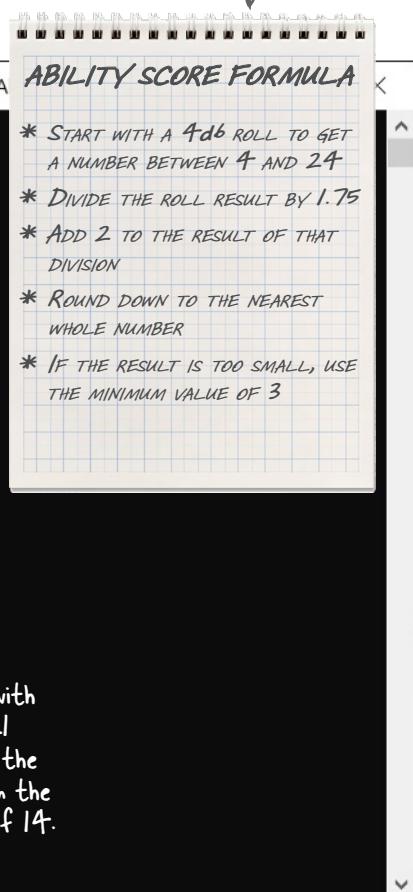
```
C:\Users\public\source\repos\AbilityScoreTester\AbilityScoreTester\bin\Debug\netcoreapp3.1\A
Starting 4d6 roll [14]:
  using default value 14
Divide by [1.75]:
  using default value 1.75
Add amount [2]:
  using default value 2
Minimum [3]:
  using default value 3
Calculated ability score: 10
Press 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.

Here's the page from Owen's game master notebook with the ability score formula.



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.

```
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;
        }
    }
}
```

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?

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.

let's try fixing the problem

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**.

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**.

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.



Exercise Solution

Here are the ReadInt and ReadDouble methods that display a prompt that includes the default value, read a line from the console, try to convert it to an int or a double, and either use the converted value or the default value, writing a message to the console with the value returned.

```
static int ReadInt(int lastUsedValue, string prompt)
{
    Console.WriteLine(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;
    }
}

static double ReadDouble(double lastUsedValue, string prompt)
{
    Console.WriteLine(prompt + " [" + lastUsedValue + "]: ");
    string line = Console.ReadLine();
    if (double.TryParse(line, out double value)) ← Here's the call to double.TryParse,
    {
        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.



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
```

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?

Where did this 9 number come from? Did we see it before? Can that give us a hint about what's causing this bug?



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:

```

39 calculator.DivideBy = ReadDouble(calculator.DivideBy, "Divide by");
40 calculator.AddAmount = ReadInt(calculator.AddAmount, "Add amount");
41 calculator.Minimum = ReadInt(calculator.Minimum, "Minimum");

```

And here's another one in the AbilityScoreCalculator.CalculateAbilityScore method—breakpoint that suspect, too:

```

20 // Add to the result
21 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 calculator.AddAmount and add a watch** (if you just right-click on AddAmount and choose “Add Watch” from the menu, it will only add a watch for AddAmount and not calculator.AddAmount). Does anything look weird there? We're not seeing anything unusual. It seems to read the value and update it just fine. 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 change the statement to use + instead of +=:

```
int added = AddAmount + (int)divided;
```

Change the += to a + to keep this line of code from updating the “added” variable and fix the bug. 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 0.3000000000000004 (with 15 zeros between the 3 and 4). Now it prints `They're equal` again. So apparently $0.1D$ plus $0.2D$ equals $0.3000000000000004D$.



←
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 `0.3000000000000004` 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 0.3000000000000004 ?

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 0.01000000000000002 , 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 $5 \div 2$, or 1. (If you’re trying to remember how long division works, that just means that $5 \div 2$ is equal to $2 \times 2 + 1$, 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 $-397 \text{ mod } 17 = 11$, 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 0.3000000000000004 problem! Check out <https://0.3000000000000004.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>.

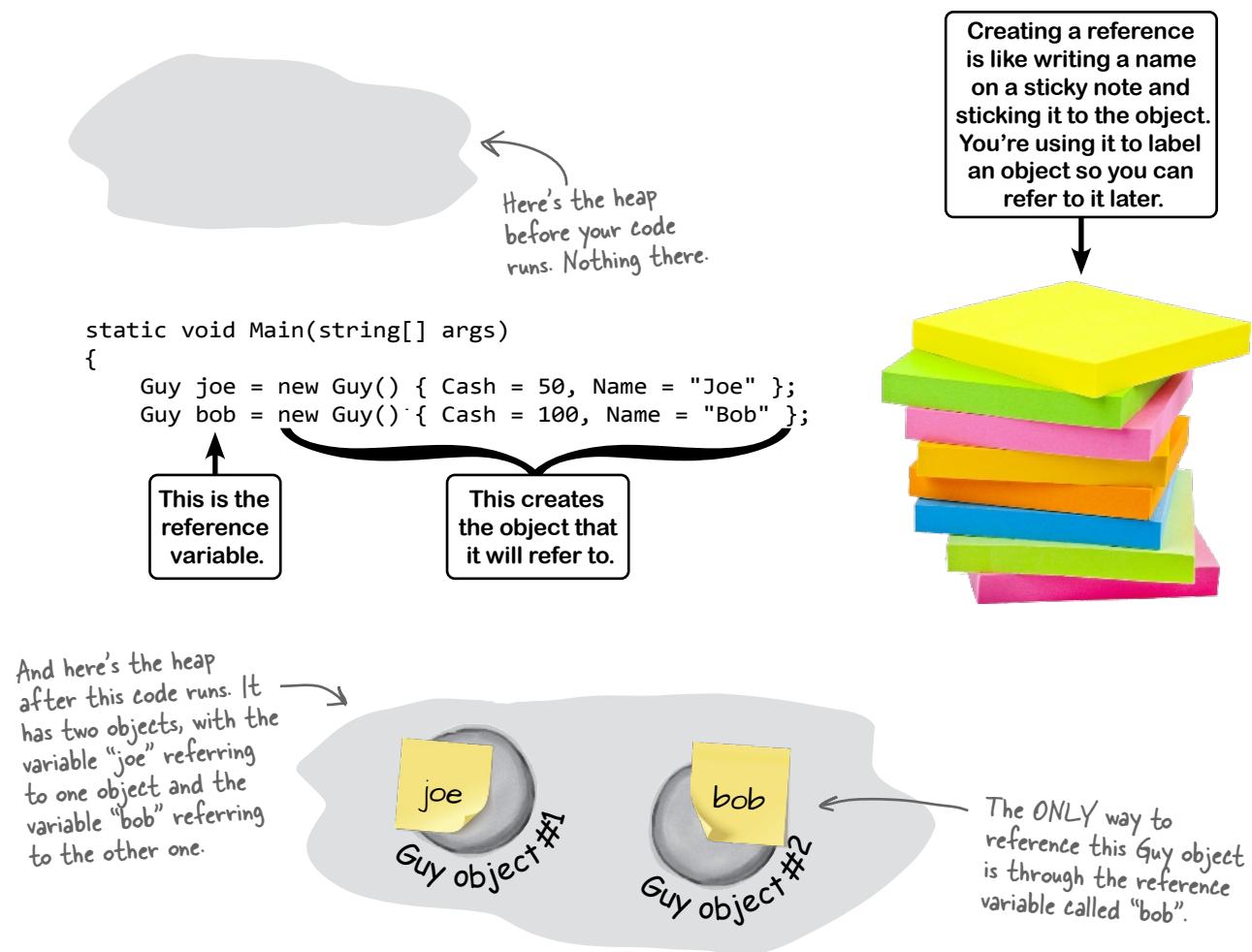
↑
Jon gave us some amazing technical review feedback for the 1st edition of this book that made a huge difference for us. Thanks so much, Jon!

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:

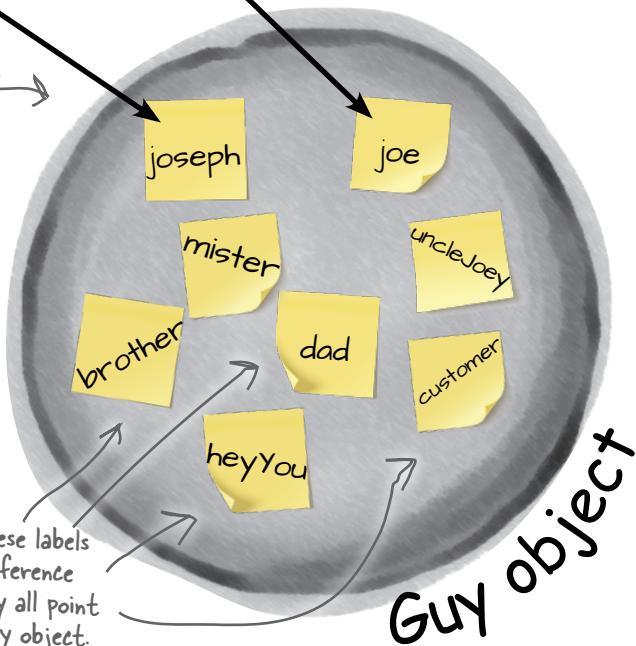


References are like sticky notes for your objects

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.



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.

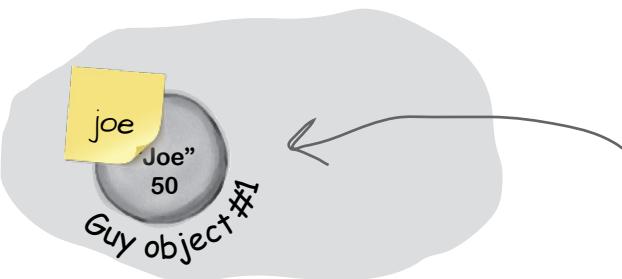
If there aren't any more references, your object gets garbage-collected

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.

➊ 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" };
```

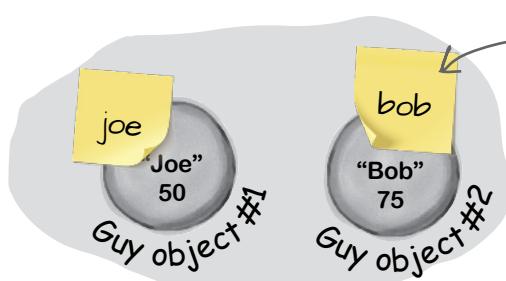


We used an object initializer to create this Guy object. Its Name field has the string "Joe", its Cash field has the int 50, and we put a reference to the object in a variable called "joe".

➋ 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.

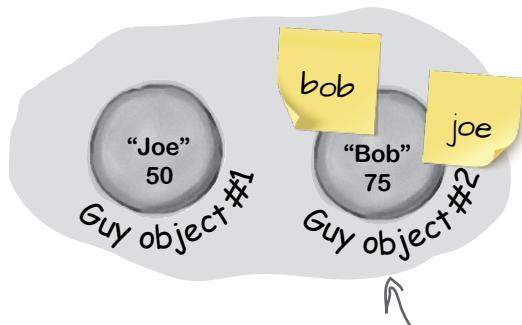
3 Let's take the reference to the first Guy object and change it to point to the second Guy object.

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.**

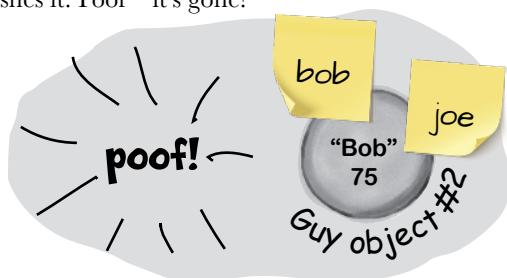


After the CLR (coming up in "Garbage Collection Exposed" interview!) removes the last reference to the object, it marks it for garbage collection.

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.

you can pet the dog in head first c#

```
public partial class Dog {  
    public void GetPet() {  
        Console.WriteLine("Woof!");  
    }  
}
```

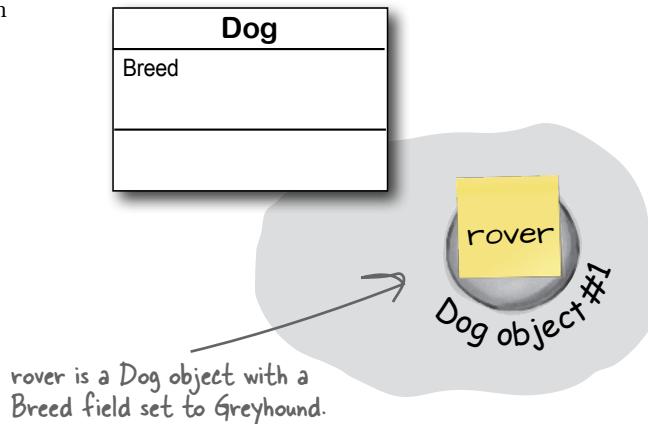
Multiple references and their side effects

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

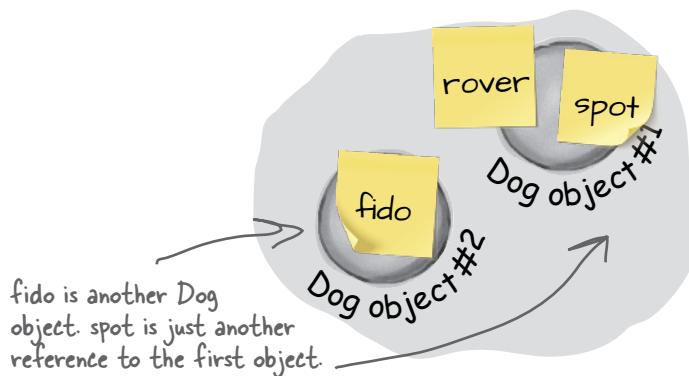
References: 1



2 **Dog fido = new Dog();**
fido.Breed = "Beagle";
Dog spot = rover;

Objects: 2

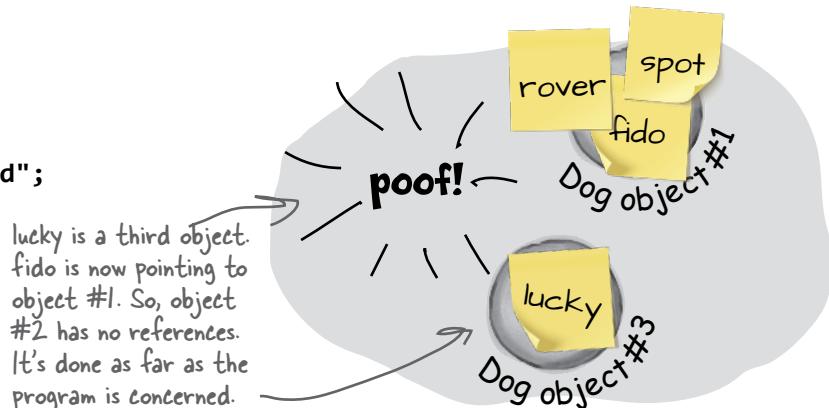
References: 3



3 **Dog lucky = new Dog();**
lucky.Breed = "Dachshund";
fido = rover;

Objects: 2

References: 4





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

One new Dog object is created, but spot is the only reference to it. When spot is set to rover, that object goes away.

References: 4

2 Dog spot = new Dog();
 spot.Breed = "Dachshund";
 spot = rover;

Objects: 3

References: 5

3 Dog lucky = new Dog();
 lucky.Breed = "Beagle";
 Dog charlie = fido;
 fido = rover;

Objects: 4

charlie was set to fido when fido was still on object #3. Then, after that, fido moved to object #1, leaving charlie behind.

References: 7

4 rintintin = lucky;
 Dog laverne = new Dog();
 laverne.Breed = "pug";

Objects: 4

Dog #2 lost its last reference, and it went away.

References: 8

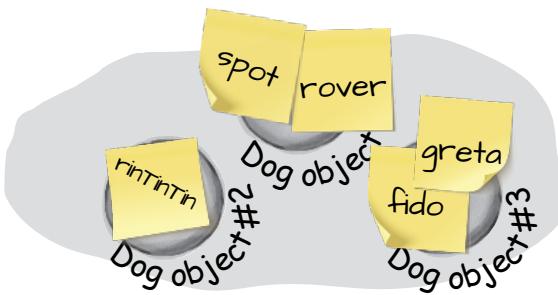
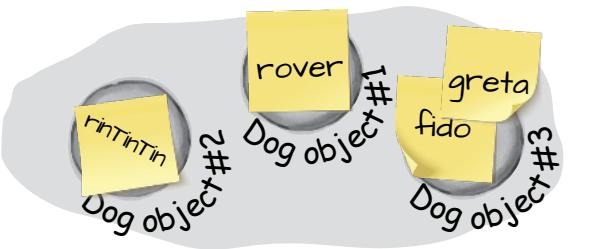
When rintintin moved to lucky's object, the old rintintin object disappeared.

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()

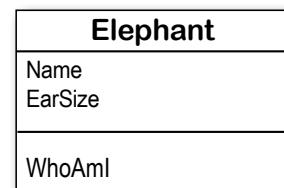
My name is Lucinda.

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.

Here's the class diagram for the Elephant class you'll need to create.



Swapping them again returns things to the way they were when the program started.

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

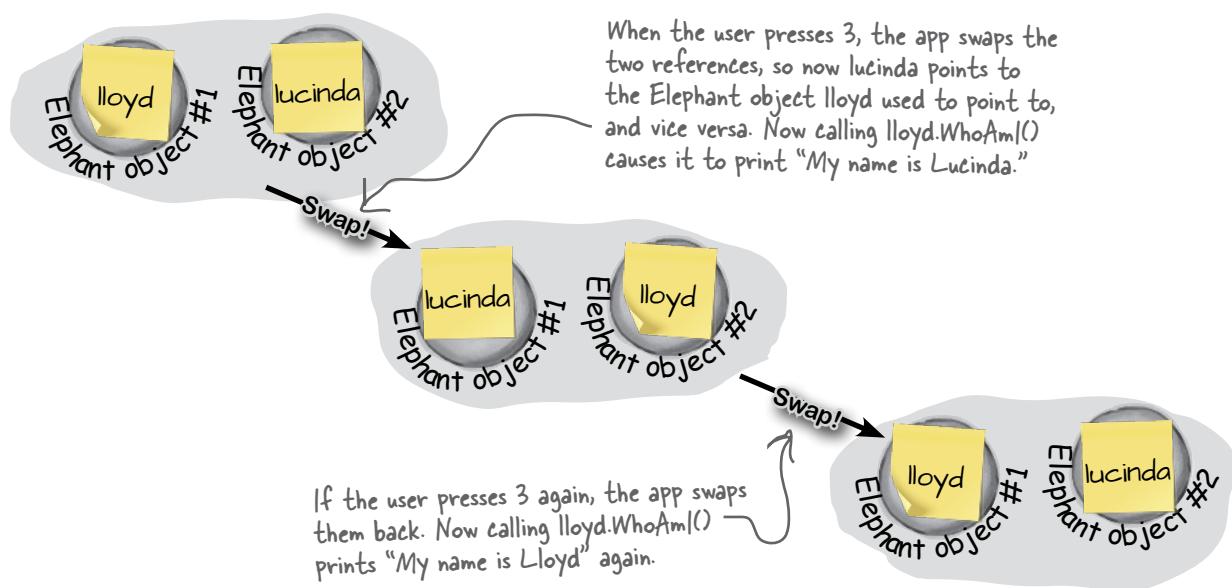
Call their WhoAmI methods.

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.



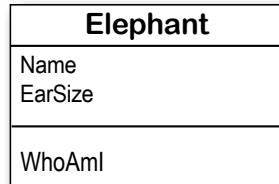


Exercise Solution

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");
        }
    }
}
```

If you just point Lucinda to Lloyd, 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.

Now go ahead and run your program. Here's what you'll see:

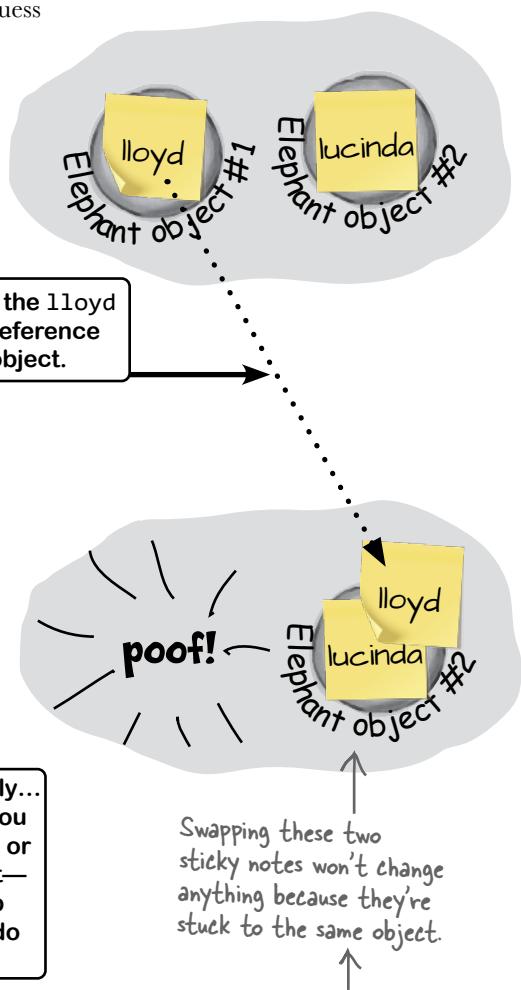
```
You pressed 4
My name is Lucinda
My ears are 4321 inches tall.
```

```
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.
```

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.

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.



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!

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.

Elephant
Name EarSize
WhoAmI HearMessage SpeakTo

1 Add a method that lets an Elephant hear a message.

Let's add a method to the Elephant class. Its first parameter is a message from another Elephant object. Its second parameter is the Elephant object that sent the message:

```
public void HearMessage(string message, Elephant whoSaidIt) {
    Console.WriteLine(Name + " heard a message");
    Console.WriteLine(whoSaidIt.Name + " said this: " + message);
}
```

Do this

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 whoTalkTo, string message) {
    whoTalkTo.HearMessage(message, this);
}
```

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:

```
whoTalkTo.HearMessage("Hi, Lloyd!", this);
```

An Elephant's SpeakTo method uses the "this" keyword to send a reference to itself to another Elephant.

[a reference to Lloyd].HearMessage("Hi, Lloyd!", [a reference to Lucinda]);

Lucinda uses **whoTalkTo** (which has a reference to Lloyd) to call HearMessage.

this is replaced with a reference to Lucinda's object.

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;
}
```

The “`this`” keyword lets an object get a reference to itself.

Now run your program and press 5. You should see this output:

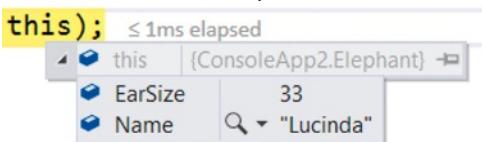
```
You pressed 5
Lloyd heard a message
Lucinda said this: Hi, Lloyd!
```

4 Use the debugger to understand what's going on.

Place a breakpoint on the statement that you just added to the Main method:



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.



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;
```

Use each element in an array like it's a normal variable

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.

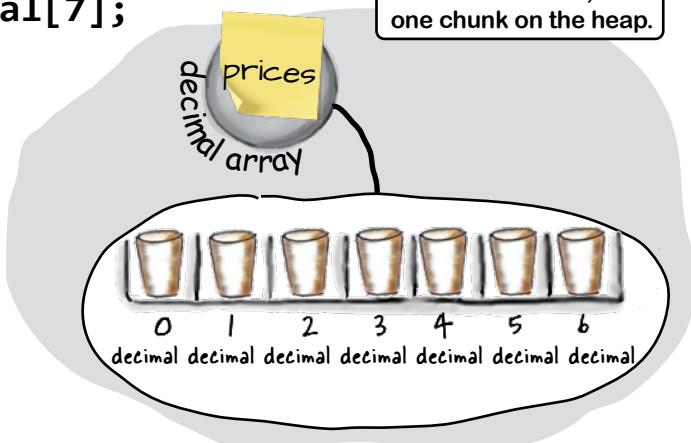
When you use an array, first you need to **declare a reference variable** that points to the array. Then you need to **create the array object** using the **new** statement, specifying how big you want the array to be. Then you can **set the elements** in the array. Here's an example of code that declares 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.

```
// 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;
```

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.



Arrays can contain reference variables

You can create an **array of object references** just like you create an array of numbers or strings. Arrays don't care what type of variable they store; it's up to you. So you can have an array of ints, or an array of Duck objects, with no problem.

Here's code that creates an array of seven Dog variables. The line that initializes the array only creates reference variables. Since there are only two `new Dog()` lines, only two actual instances of the Dog class are created.

```
// Declare a variable that holds an
// array of references to Dog objects
Dog[] dogs = new Dog[7];

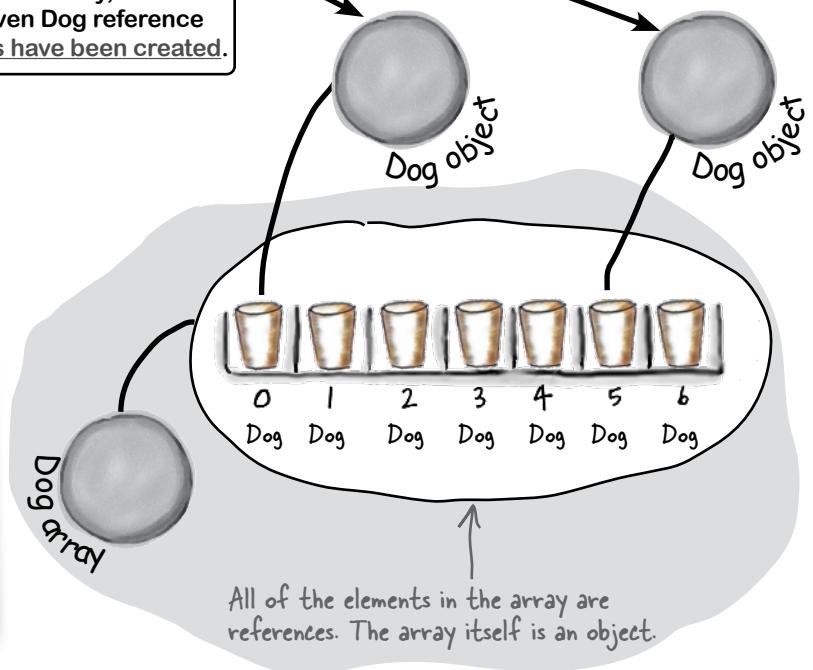
// Create two new instances of Dog
// and put them at indexes 0 and 5
dogs[5] = new Dog();
dogs[0] = new Dog();
```

When you set or retrieve an element from an array, the number inside the brackets is called the index. The first element in the array has an index of 0.

The first line of code only created the array, not the instances. The array is a list of seven Dog reference variables—but only two Dog objects have been created.

An array's length

You can find out how many elements are in an array using its `Length` property. So if you've got an array called "prices", then you can use `prices.Length` to find out how long it is. If there are seven elements in the array, that'll give you 7—which means the array elements are numbered 0 to 6.





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]; ← We're creating an array of
                                                seven Elephant references.

    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];           Iteration #1 biggestEars.EarSize = _____
    for (int i = 1; i < elephants.Length; i++)
    {
        Console.WriteLine("Iteration #" + i);       Iteration #2 biggestEars.EarSize = _____
        if (elephants[i].EarSize > biggestEars.EarSize)
        {
            biggestEars = elephants[i];           Iteration #3 biggestEars.EarSize = _____
            This sets the biggestEars reference to the
            object that elephants[i] points to.
        }
        Console.WriteLine(biggestEars.EarSize.ToString());
    }                                               Iteration #4 biggestEars.EarSize = _____
}                                               Iteration #5 biggestEars.EarSize = _____
```

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 #6 biggestEars.EarSize = _____

Arrays start with index 0, so the first Elephant in the array is `elephants[0]`.

null means a reference points to nothing

There's another important keyword that you'll use with objects. When you create a new reference and don't set it to anything, it has a value. It starts off set to `null`, which means **it's not pointing to any object at all**. Let's have a closer look at this:

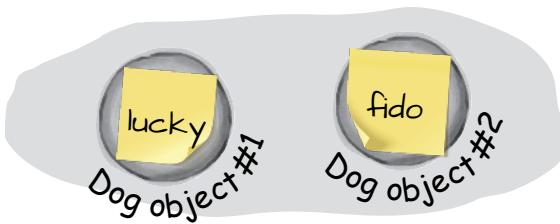
The default value for any reference variable is `null`. Since we haven't assigned a value to `fido`, it's set to `null`.

→ `Dog fido;`
`Dog lucky = new Dog();`



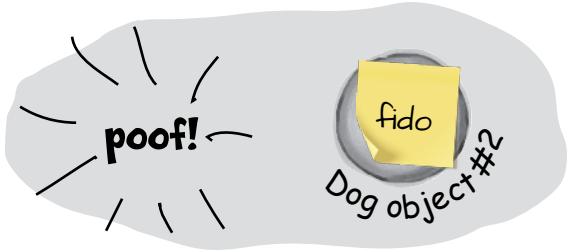
Now `fido` is set to a reference to another object, so it's not equal to `null` anymore.

→ `fido = new Dog();`



Once we set `lucky` to `null` it no longer points to its object, so it gets marked for garbage collection.

→ `lucky = null;`



WOULD I EVER **REALLY** USE
NULL IN A PROGRAM?

Yes. The `null` keyword can be 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

```
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 };
```

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.

Did you remember that the loop starts with the second element of the array? Why do you think that is?

```
    Elephant biggestEars = elephants[0];
```

Iteration #1 biggestEars.EarSize = 40

```
    for (int i = 1; i < elephants.Length; i++)
```

```
{
```

```
    Console.WriteLine("Iteration #" + i);
```

Iteration #2 biggestEars.EarSize = 42

```
    if (elephants[i].EarSize > biggestEars.EarSize)
```

```
{
```

Iteration #3 biggestEars.EarSize = 42

```
        biggestEars = elephants[i];
```

```
}
```

Iteration #4 biggestEars.EarSize = 44

```
    Console.WriteLine(biggestEars.EarSize.ToString());
```

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 #5 biggestEars.EarSize = 44

45

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.



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!



Game design... and beyond



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 minimum 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);
```



How would you use Random to choose a random string from an array of strings?

sloppy joe sez: "that roast beef's not old...it's vintage"

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.



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 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.

```
<Grid Margin="20">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="5*"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
```

The grid has 7 equal-sized rows

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

The bottom `TextBlock` spans both columns

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`.

```
<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;
    }
}

```

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

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.

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".

I EAT ALL MY MEALS AT SLOPPY JOE'S!



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 MenuItem objects 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	
Why aren't the menu items and prices getting randomized?		Add guacamole for \$3.38

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.**

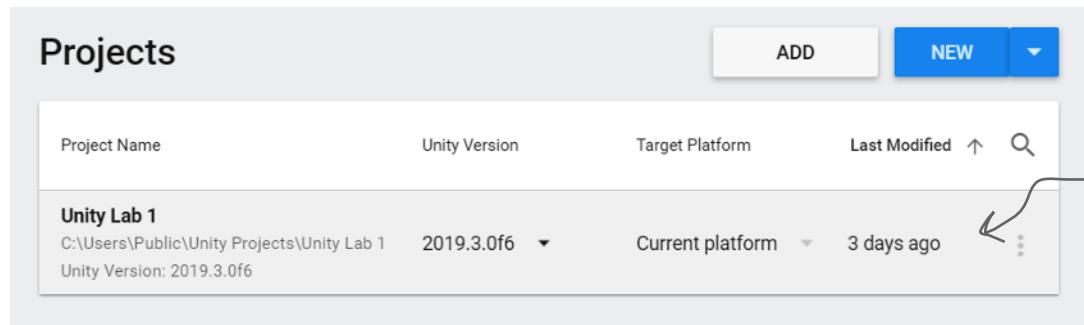
In the last Unity Lab, you learned how to navigate around Unity and your 3D space, and started to create and explore GameObjects. Now it's time to write some code to take control of your GameObjects. The whole goal of that lab was to get you oriented in the Unity editor (and give you an easy way to remind 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

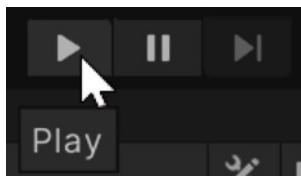
Now that you can add a GameObject to your scene, you need a way to make it, well, do stuff. That's where your C# skills come in. Unity uses **C# scripts** to define the behavior of everything in the game.

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.



Here's what you'll do in this Unity Lab:

- ➊ **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.
- ➋ **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.
- ➌ **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.

- ➍ **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.

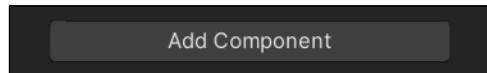
Unity Lab #2

Write C# Code for Unity

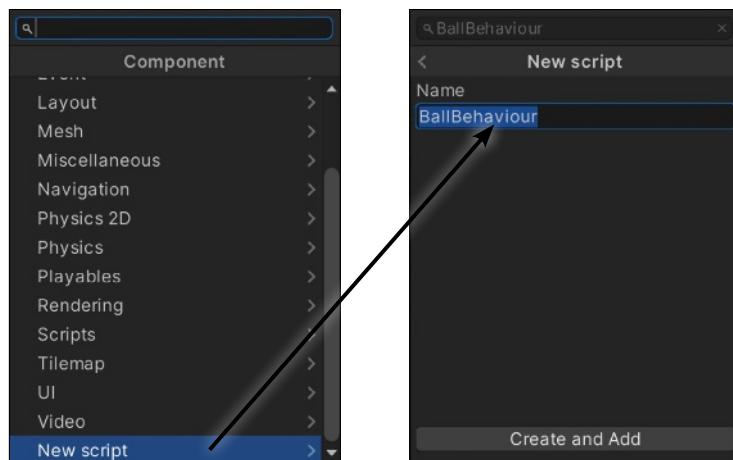
Add a C# script to your GameObject

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#**.

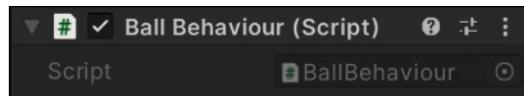
Let's start using C# and Unity right now. Make sure the Sphere GameObject is selected, then **click the Add Component button** at the bottom of the Inspector window.



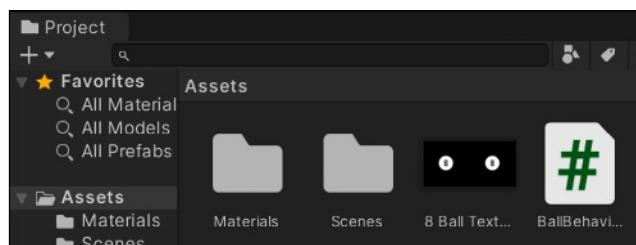
When you click it, Unity pops up a window with all of the different kinds of components that you can add—and there are **a lot** of them. **Choose “New script”** to add a new C# script to your Sphere GameObject. You'll be prompted for a name. **Name your script BallBehaviour**.



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.



Unity code uses British spelling.

Watch it!

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**.

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()
    {
    }
}
```

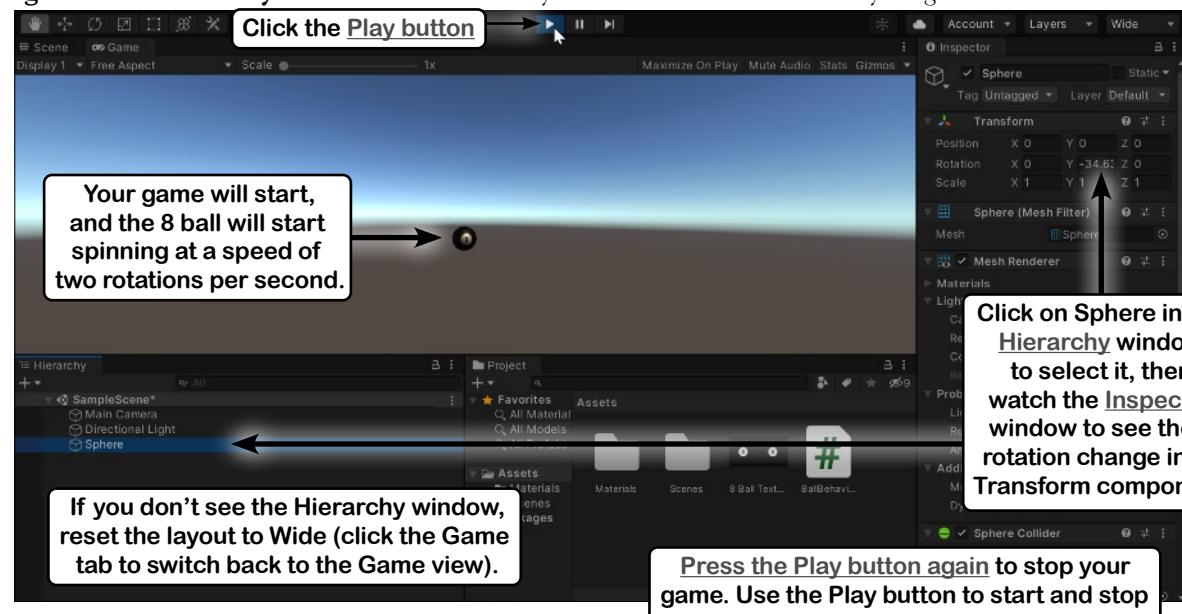
You opened your C# script in Visual Studio by clicking on it in the Hierarchy window, which shows you a list of every GameObject in the current scene. When Unity created your project, it added a scene called SampleScene with a camera and a light. You added a sphere to it, so your Hierarchy window will 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:



Your Code Up Close



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

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.

```
public class BallBehaviour : MonoBehaviour
```

```
{
```

// Start is called before the first frame update

```
void Start()
```

```
{
```

```
}
```

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.

// Update is called once per frame

```
void Update()
```

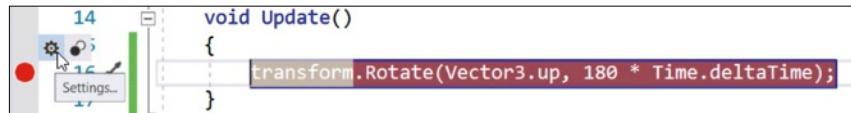
```
{
```

```
    transform.Rotate(Vector3.up, 180 * Time.deltaTime);
```

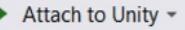
The transform.Rotate method causes a GameObject to rotate. The first parameter is the axis to rotate around. In this case, your code used Vector3.up, which tells it to rotate around the Y axis. The second parameter is the number of degrees to rotate.

Add a breakpoint and debug your game

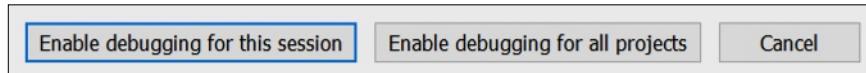
Let's debug your Unity game. First **stop your game** if it's still running (by pressing the Play button again). Then switch over to Visual Studio, and **add a breakpoint** on the line that you added to the Update method.



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 ($\mathcal{H} \leftrightarrow$)

Click that button to **start the debugger**. Now switch back to the Unity editor. If this is the first time you're debugging this project, the Unity editor will pop up a dialog window with these buttons:



Press the “Enable debugging for this session” button (or if you want to keep that pop-up from appearing again, press “Enable debugging for all projects”). Visual Studio is now **attached** to Unity, which means it can debug your game.

Now **press the Play button in Unity** to start your game. Since Visual Studio is attached to Unity, it **breaks immediately** on the breakpoint that you added, just like with any other breakpoint you've set.

Use a hit count to skip frames

 Congratulations, you're now debugging a game!

Sometimes it's useful to let your game run for a while before your breakpoint stops it. For example, you might want your game to spawn and move its enemies before your breakpoint hits. Let's tell your breakpoint to break every 500 frames. You can do that by adding a **Hit Count condition** to your breakpoint:

- ★ On Windows, right-click on the breakpoint dot (●) at the left side of the line, choose **Conditions** from the pop-up menu, select *Hit Count* and *Is a multiple of* from the dropdowns, and enter 500 in the box:



- ★ On macOS, right-click on the breakpoint dot (●), choose **Edit breakpoint...** from the menu, then choose *When hit count is a multiple of* from the dropdown and enter 500 in the box:



Now the breakpoint will only pause the game every 500 times the Update method is run—or every 500 frames. 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.

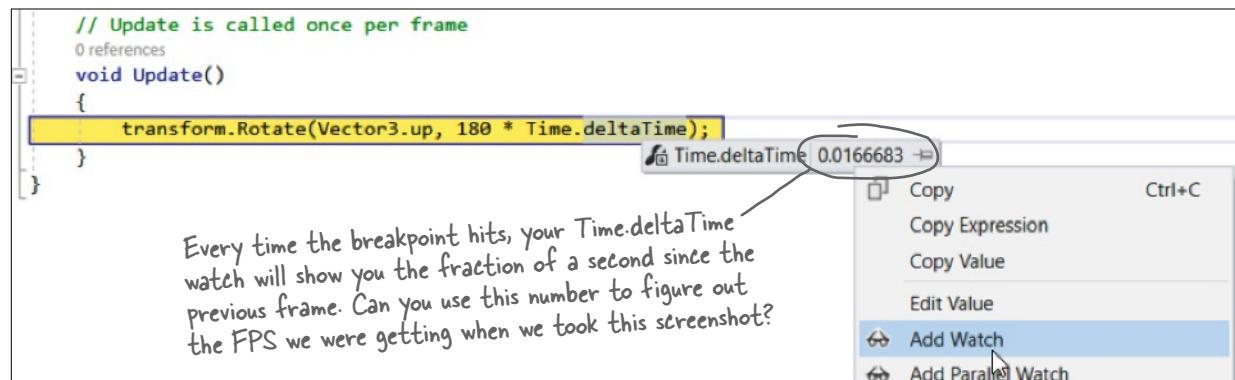
Unity Lab #2

Write C# Code for Unity

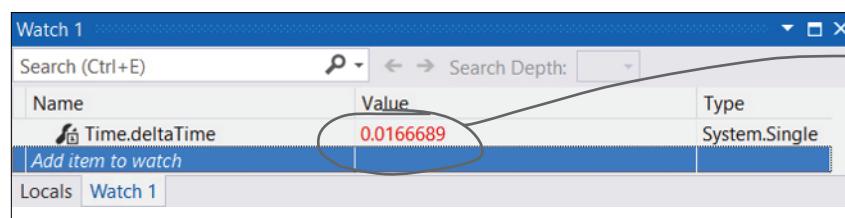
Use the debugger to understand Time.deltaTime

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.



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.



Press the Continue button to get another Time.deltaTime value, then another. You can get your approximate FPS by dividing $1 / \text{Time.deltaTime}$.

Stop debugging (Shift+F5 on Windows, ⌘⌘← on macOS) to stop your program. Then **start debugging again**. Since your game is still running, the breakpoint will continue to work when you reattach Visual Studio to Unity. Once you're done debugging, **toggle your breakpoint again** so the IDE will still keep track of it but not break when it's hit. **Stop debugging** one more time to detach from Unity.

Go back to Unity and **stop your game**—and save it, because the Play button doesn't automatically save the game.

The Play button in Unity starts and stops your game. Visual Studio will stay attached to Unity even when the game is stopped.

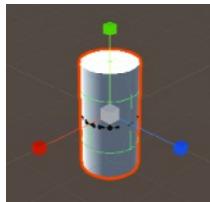


Debug your game again and hover over "Vector3.up" to inspect its value—you'll have to put your mouse cursor over up. It has a value of (0.0, 1.0, 0.0). What do you think that means?

Add a cylinder to show where the Y axis is

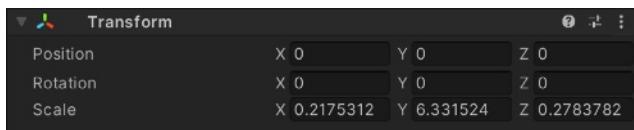
Your sphere is rotating around the Y axis at the very center of the scene. Let's add a very tall and very skinny cylinder to make it visible. **Create a new cylinder** by choosing *3D Object >> Cylinder* from the GameObject menu. Make sure it's selected in the Hierarchy window, then look at the Inspector window and check that Unity created it at position (0, 0, 0)—if not, use the context menu (>Edit) to reset it.

Let's make the cylinder tall and skinny. Choose the Scale tool from the toolbar: either click on it (S) or press the R key. You should see the Scale Gizmo appear on your cylinder:



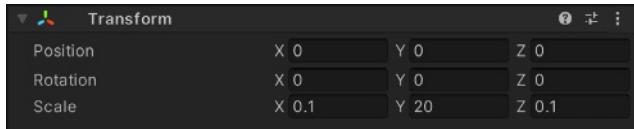
The Scale Gizmo looks a lot like the Move Gizmo, except that it has cubes instead of cones at the end of each axis. Your new cylinder is sitting on top of the sphere—you might see just a little of the sphere showing through the middle of the cylinder. When you make the cylinder narrower by changing its 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.

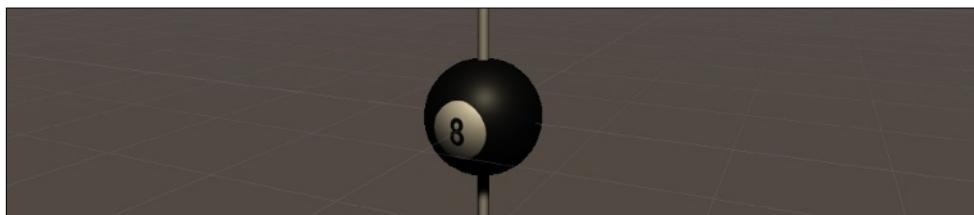


Click on the X label in the Scale row in the Transform panel and drag up and down. Make sure you click the actual X label to the left of the input box with the number. When you click the label it turns blue, and a blue box appears around the X value. As you drag your mouse up and down, the number in 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.



Now your sphere has a very long cylinder going through it that shows the Y axis where Y = 0.



Unity Lab #2

Write C# Code for Unity

Add fields to your class for the rotation angle and speed

In Chapter 3 you learned how C# classes can have **fields** that store values methods can use. Let's modify your code to use fields. Add these four lines just under the class declaration, **immediately after the first curly brace { :**

```
public class BallBehaviour : MonoBehaviour  
{  
    public float XRotation = 0;  
    public float YRotation = 1;  
    public float ZRotation = 0;  
    public float DegreesPerSecond = 180;
```

These are just like the fields that you added to the projects in Chapters 3 and 4. They're variables that keep track of their values—each time **Update** is called it reuses the same field over and over again.

The XRotation, YRotation, and ZRotation fields each contain a value between 0 and 1, which you'll combine to create a **vector** that determines the direction that the ball will rotate:

```
new Vector3(XRotation, YRotation, ZRotation)
```

The DegreesPerSecond field contains the number of degrees to rotate per second, which you'll multiply by Time.deltaTime just like before. **Modify your Update method to use the fields.** This new code creates a Vector3 variable called **axis** and passes it to the transform.Rotate method:

```
void Update()  
{  
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);  
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);  
}
```

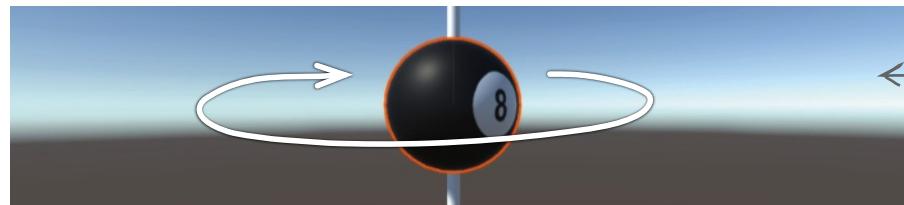
Select the Sphere in the Hierarchy window. Your fields now show up in the Script component. When the Script component renders fields, it adds spaces between the capital letters to make them easier to read.



When you add **public** fields to a class in your Unity script, the Script component displays input boxes that let you modify those fields. If you modify them while the game is not running, the updated values will get saved with your scene. You can also modify them while the game is running, but they'll revert when you stop the game.

Run your game again. **While it's running**, select the Sphere in the Hierarchy window and change the degrees per second to 360 or 90—the ball starts to spin at twice or half the speed. Stop your game—and the field will reset to 180.

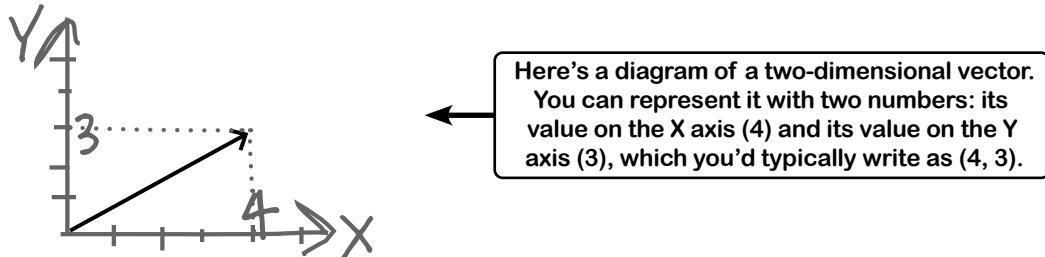
While the game is stopped, use the Unity editor to change the X Rotation field to 1 and the Y Rotation field to 0. Start your game—the ball will rotate away from you. Click the X Rotation label and drag it up and down to change the value while the game is running. As soon as the number turns negative, 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:



That's not hard to understand...on an intellectual level. But even those of us who took a math class that covered vectors don't always have an **intuitive** grasp of how vectors work, especially in 3D. Here's another area where we can use C# and Unity as a tool for learning and exploration.

Use Unity to visualize vectors in 3D

You're going to add code to your game to help you really “get” how 3D vectors work. Start by having a closer look at the first line of your Update method:

```
Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
```

What does this line tell us about the vector?

- ★ **It has a type: Vector3.** Every variable declaration starts with a type. Instead of using string, int, or bool, you’re declaring it with the type Vector3. This is a type that Unity uses for 3D vectors.
- ★ **It has a variable name: axis.**
- ★ **It uses the new keyword to create a Vector3.** It uses the XRotation, YRotation, and ZRotation fields to create a vector with those values.

So what does that 3D vector look like? There’s no need to guess—we can use one of Unity’s useful debugging tools to draw the vector for us. **Add this line of code to the end of your Update method:**

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow);
}
```

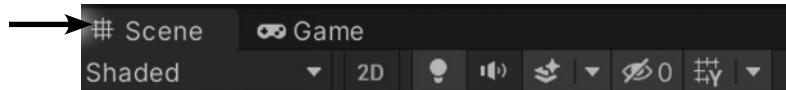
The Debug.DrawRay method is a special method that Unity provides to help you debug your games. It draws a **ray**—which is a vector that goes from one point to another—and takes parameters for its start point, end point, and color. There’s one catch: **the ray only appears in the Scene view**. The methods in Unity’s Debug class are designed so that they don’t interfere with your game. They typically only affect how your game interacts with the Unity editor.

Unity Lab #2

Write C# Code for Unity

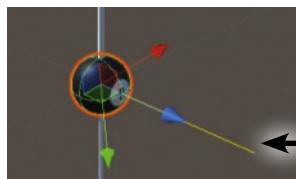
Run the game to see the ray in the Scene view

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.

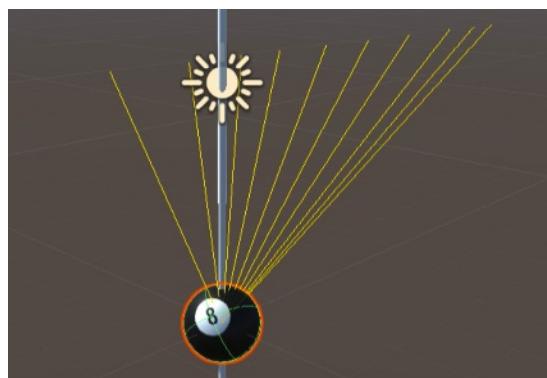
- ★ Set the Z Rotation back to 3. Experiment with dragging the X Rotation and Y Rotation values to see what they do to the ray. Make sure to reset the Transform component each time you change them.
- ★ Use the Hand tool and the Scene Gizmo to get a better view. Click the X cone on the Scene Gizmo to 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);
```

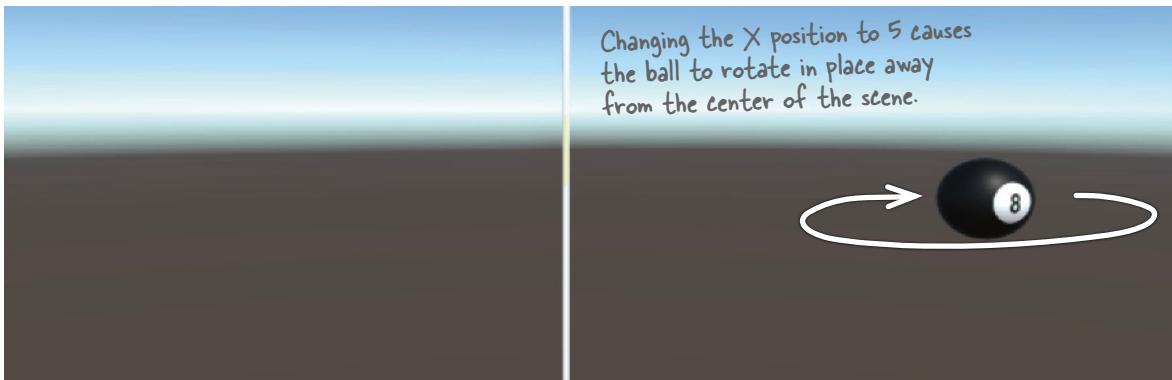
Now run the game again and switch to the Scene view. Now when you drag the numbers up and down, you'll see a trail of rays left behind. This looks really interesting, but more importantly, it's a great tool to visualize 3D vectors.



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.

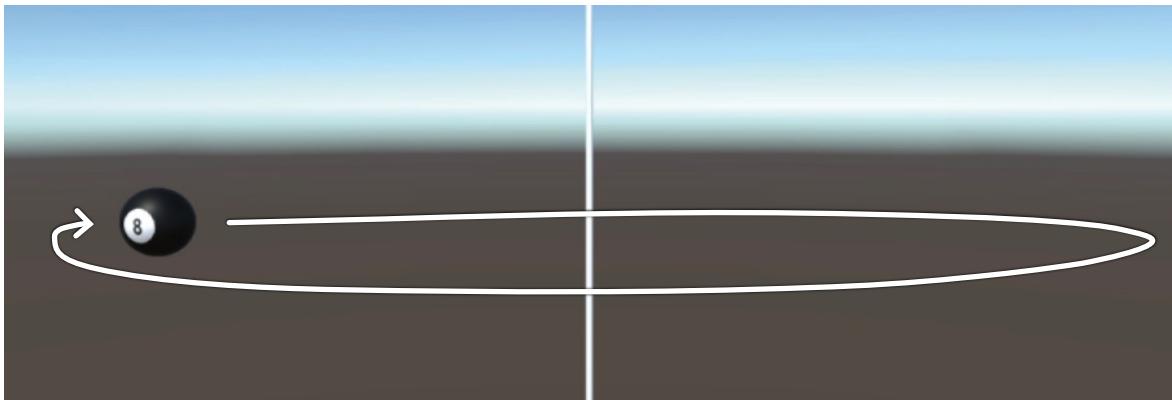


Let's modify the Update method to use a different kind of rotation. Now we'll make the ball rotate around the center point of the scene, coordinate (0, 0, 0), using the **transform.RotateAround** method, which rotates a GameObject around a point in the scene. (This is *different* from the transform.Rotate 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:

```
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:



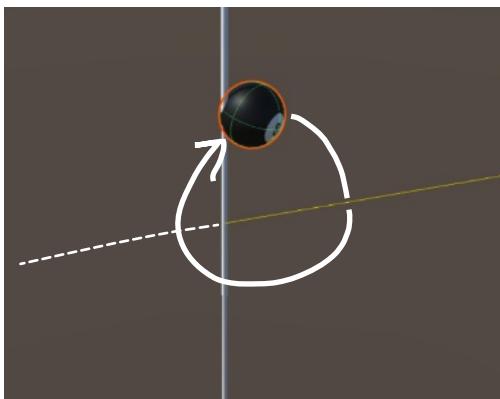
This new Update method rotates the ball around the point (0, 0, 0) in the scene.

Use Unity to take a closer look at rotation and vectors

You're going to be working with 3D objects and scenes in the rest of the Unity Labs throughout the book. Even those of us who spend a lot of time playing 3D video games don't have a perfect feel for how vectors and 3D objects work, and how to move and rotate in a 3D space. Luckily, Unity is a great tool to **explore how 3D objects work**. Let's start experimenting right now.

While your code is running, try changing parameters to experiment with the rotation:

- ★ **Switch back to the Scene view** so you can see the yellow ray that Debug.DrawRay renders in your BallBehaviour.Update method.
- ★ Use the Hierarchy window to **select the Sphere**. You should see its components in the Inspector window.
- ★ Change the **X Rotation, Y Rotation, and Z Rotation values** in the Script component to **10** so you see the vector rendered as a long ray. Use the Hand tool (Q) to rotate the Scene view until you can clearly see the ray.
- ★ Use the Transform component's context menu (⋮) to **reset the Transform component**. Since the center of the sphere is now at the zero point in the scene, (0, 0, 0), it will rotate around its own center.
- ★ Then **change the X position in** the Transform component to **2**. The ball should now be rotating around the vector. You'll see the ball cast a shadow on the Y axis cylinder as it flies by.



While the game is running, set the X, Y, and Z Rotation fields in the BallBehaviour Script component to 10, reset the sphere's Transform component, and change its X position to 2—as soon as you do, it starts rotating around the ray.

Try **repeating the last three steps** for different values of X, Y, and Z rotation, resetting the Transform component each time so you start from a fixed point. Then try clicking the rotation field labels and dragging them up and down—see if you can get a feel for how the rotation works.

Unity is a great tool to explore how 3D objects work by modifying properties on your GameObjects in real time.

Get creative!

This is your chance to **experiment on your own with C# and Unity**. You've seen the basics of how you combine C# and Unity GameObjects. Take some time and play around with the different Unity tools and methods that you've learned about in the first two Unity Labs. Here are some ideas:

- ★ Add cubes, cylinders, or capsules to your scene. Attach new scripts to them—make sure you give each script a unique name!—and make them rotate in different ways.
- ★ Try putting your rotating GameObjects in different positions around the scene. See if you can make interesting visual patterns out of multiple rotating GameObjects.
- ★ Try adding a light to the scene. What happens when you use `transform.rotateAround` to rotate the new light around various axes?
- ★ Here's a quick coding challenge: try using `+=` to add a value to one of the fields in your BallBehaviour script. Make sure you multiply that value by `Time.deltaTime`. Try adding an `if` statement that resets the field to 0 if it gets too large.



Before you run the code, try to figure out what it will do. Does it act the way you expected it to act? Trying to predict how the code you added will act is a great technique for getting better at C#.

Take the time to experiment with the tools and techniques you just learned. This is a great way to take advantage of Unity and Visual Studio as tools for exploration and learning.

BULLET POINTS

- The **Scene Gizmo** always displays the camera's orientation.
- You can **attach a C# script** to any GameObject. The script's `Update` method will be called once per frame.
- The **transform.Rotate** method causes a GameObject to rotate a number of degrees around an axis.
- Inside your `Update` method, multiplying any value by `Time.deltaTime` turns it into that value per second.
- You can **attach** the Visual Studio debugger to Unity to debug your game while it's running. It will stay attached to Unity even when your game is not running.
- Adding a **Hit Count condition** to a breakpoint to makes it break after the statement has executed a certain number of times.
- A **field** is a variable that lives inside of a class outside of its methods, and it retains its value between method calls.
- Adding public fields to the class in your Unity script makes the Script component show **input boxes** that let you **modify those fields**. It adds spaces between capital letters in the field names to make them easier to read.
- You can create 3D vectors using **new Vector3**. (You learned about the `new` keyword in Chapter 3.)
- The **Debug.DrawRay** method draws a vector in the Scene view (but not the Game view). You can use vectors as a debugging tool, but also as a learning tool.
- The **transform.RotateAround** method rotates a GameObject around a point in the scene.

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.

Let's help Owen roll for damage

Owen was so happy with his ability score calculator that he wanted to create more C# programs he can use for his games, and you're going to help him. In the game he's currently running, any time there's a sword attack he rolls dice and uses a formula that calculates the damage. Owen wrote down how the **sword damage formula** works in his game master notebook.

Here's a **class called SwordDamage** that does the calculation. Read through the code carefully—you're about to create an app that uses it.

```
class SwordDamage
{
    public const int BASE_DAMAGE = 3; ←
    public const int FLAME_DAMAGE = 2;

    public int Roll;
    public decimal MagicMultiplier = 1M;
    public int FlamingDamage = 0;
    public int Damage;

    public void CalculateDamage() ←
    {
        Damage = (int)(Roll * MagicMultiplier) + BASE_DAMAGE + FlamingDamage;
    }

    public void SetMagic(bool isMagic)
    {
        if (isMagic)
        {
            MagicMultiplier = 1.75M;
        }
        else
        {
            MagicMultiplier = 1M;
        }
        CalculateDamage();
    }

    public void SetFlaming(bool isFlaming)
    {
        CalculateDamage();
        if (isFlaming)
        {
            Damage += FLAME_DAMAGE;
        }
    }
}
```

Here's the description of the sword damage formula in Owen's game master notebook.

* TO FIND THE NUMBER OF HIT POINTS (HP) OF DAMAGE FOR A SWORD ATTACK, ROLL 3D6 (THREE 6-SIDED DICE) AND ADD "BASE DAMAGE" OF 3HP.

* SOME SWORDS ARE FLAMING, WHICH CAUSES AN EXTRA 2HP OF DAMAGE.

* SOME SWORDS ARE MAGIC. FOR MAGIC SWORDS, THE 3D6 ROLL IS MULTIPLIED BY 1.75 AND ROUNDED DOWN, AND THE BASE DAMAGE AND FLAMING DAMAGE ARE ADDED TO THE RESULT.

Here's a useful C# tool. Since the base damage or flame damage won't be changed by the program, you can use the **const** keyword to declare them as **constants**, which are like variables except that their value can never be changed. If you write code that tries to change a constant, you'll get a compiler error.

Here's where the damage formula gets calculated. Take a minute and read the code to see how it implements the formula.

NOW I CAN SPEND LESS TIME CALCULATING DAMAGE AND MORE TIME MAKING THE GAME FUN FOR THE PLAYERS.

Since flaming swords cause extra damage in addition to the roll, the **SetFlaming** method calculates the damage and then adds **FLAME_DAMAGE** to it.





Create a console app to calculate damage

Let's build a console app for Owen that uses the SwordDamage class. It will print a prompt to the console asking the user to specify whether the sword is magic and/or flaming, then it will do the calculation. Here's an example of the output of the app:

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 0
Rolled 11 for 14 HP
    Rolling 11 for a non-magic, non-flaming
    sword will cause 11 + 3 = 14 HP of damage.
```

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 0
Rolled 15 for 18 HP
```

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 1
Rolled 11 for 22 HP
    Rolling 11 for a magic sword will cause
    (round down 11 × 1.75 = 19) + 3 = 22
```

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 1
Rolled 8 for 17 HP
```

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 2
Rolled 10 for 15 HP
```

Rolling 17 for a magic flaming sword causes
(round down 17 × 1.75 = 29) + 3 + 2 = 34.

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 3
Rolled 17 for 34 HP
```

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: q
Press any key to continue...
```



Exercise

Draw a class diagram for the SwordDamage class. Then create a new console app and add the SwordDamage class. While you're carefully entering the code, take a really close look at how the SetMagic and SetFlaming methods work, and how they work a little differently from each other. Once you're confident you understand it, you can build out the Main method. Here's what it will do:

1. Create a new instance of the SwordDamage class, and also a new instance of Random.
2. Write the prompt to the console and read the key. Call Console.ReadKey(false) so the key that the user typed is printed to the console. If the key isn't 0, 1, 2, or 3, **return** to end the program.
3. Roll 3d6 by calling random.Next(1, 7) three times and adding the results together, and set the Roll field.
4. If the user pressed 1 or 3 call SetMagic(true); otherwise call SetMagic(false). You don't need an **if** statement to do this: `key == '1'` returns true, so you can use `||` to check the key directly inside the argument.
5. If the user pressed 2 or 3, call SetFlaming(true); otherwise call SetFlaming(false). Again, you can do this in a single statement using `==` and `||`.
6. Write the results to the console. Look carefully at the output and use `\n` to insert line breaks where needed.



Exercise Solution

This console app rolls for damage by creating a new instance of the SwordDamage class that we gave you (and an instance of Random to generate the 3d6 rolls) and generates output that matches the example.

```
public static void Main(string[] args)
{
    Random random = new Random();
    SwordDamage swordDamage = new SwordDamage();
    while (true)
    {
        Console.Write("0 for no magic/flaming, 1 for magic, 2 for flaming, " +
                     "3 for both, anything else to quit: ");
        char key = Console.ReadKey().KeyChar;
        if (key != '0' && key != '1' && key != '2' && key != '3') return;
        int roll = random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
        swordDamage.Roll = roll;
        swordDamage.SetMagic(key == '1' || key == '3');
        swordDamage.SetFlaming(key == '2' || key == '3');
        Console.WriteLine("\nRolled " + roll + " for " + swordDamage.Damage + " HP\n");
    }
}
```

SwordDamage

Roll
MagicMultiplier
FlamingDamage
Damage
CalculateDamage
SetMagic
SetFlaming



THAT IS EXCELLENT!
BUT I WAS WONDERING...DO YOU
THINK YOU CAN BUILD A MORE VISUAL
APP FOR IT?

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

Let's find a way to **reuse** the SwordDamage class in a WPF app. The first challenge for us is how to provide an *intuitive* user interface. A sword can be magic, flaming, both, or none, so we need to figure out how we want to handle that in a GUI—and there are a lot of options. We could have a radio button or dropdown list with four options, just like the console app gave four options. However, we think it would be cleaner and more visually obvious to use **checkboxes**.

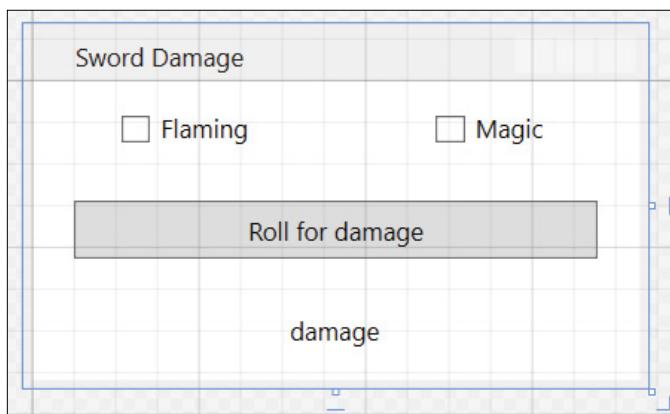
In WPF, CheckBox uses the Content property to display the label to the right of the box, just like a Button uses the Content property for the text that it displays. We have SetMagic and SetFlaming methods, so we can use the CheckBox control's **Checked and Unchecked events**, which let you specify methods that get called when the user checks or unchecks the box.

Go to the Visual Studio for Mac Learner's Guide for the Mac version of this project.

Design the XAML for a WPF version of the damage calculator

Create a new WPF app, and set the main window's title to **Sword Damage**, height to **175**, and width to **300**.

Add three rows and two columns to the grid. The top row should have two CheckBox controls labeled Flaming and Magic, the middle row a Button control labeled “Roll for damage” that spans both columns, and the bottom row a TextBlock control that spans both columns.



Design this!

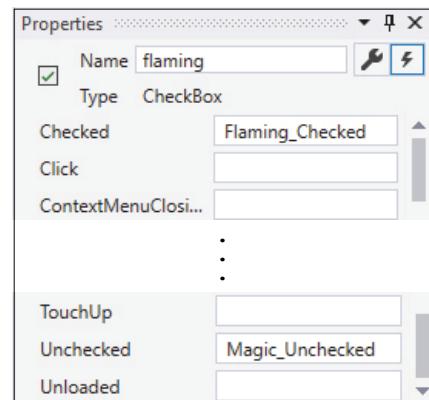
Select a CheckBox, then use the Events button in the Properties window to display the events. Once you enter the control name at the top of the window, you can double-click on the Checked and Unchecked boxes—the IDE will add them automatically, and use the control names to generate the names of the event handler methods.

Here's the XAML—you can definitely use the designer to build your form, but you should also feel comfortable editing the XAML by hand:

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

    <CheckBox x:Name="flaming" Content="Flaming"
              HorizontalAlignment="Center" VerticalAlignment="Center"
              Checked="Flaming_Checked" Unchecked="Flaming_Unchecked"/>
    <CheckBox x:Name="magic" Content="Magic" Grid.Column="1"
              HorizontalAlignment="Center" VerticalAlignment="Center"
              Checked="Magic_Checked" Unchecked="Magic_Unchecked" />
    <Button Grid.Row="1" Grid.ColumnSpan="2" Margin="20,10"
           Content="Roll for damage" Click="Button_Click"/>
    <TextBlock x:Name="damage" Grid.Row="2" Grid.ColumnSpan="2" Text="damage"
              VerticalAlignment="Center" HorizontalAlignment="Center"/>
</Grid>
```

Name the CheckBox controls **magic** and **flaming**, and the TextBlock control **damage**. Make sure the names appear in the XAML correctly in the **x:Name** properties.



The Checked and Unchecked event handlers get called when the user checks or unchecks the boxes.

This text will be replaced by the output (“Rolled 17 for 34 HP”).

hmm...something's not right

The code-behind for the WPF damage calculator

Add this **code-behind** to your WPF app. It creates instances of SwordDamage and Random, and makes the checkboxes and button calculate damage:



```
public partial class MainWindow : Window
{
    Random random = new Random();
    SwordDamage swordDamage = new SwordDamage();

    public MainWindow()
    {
        InitializeComponent();
        swordDamage.SetMagic(false);
        swordDamage.SetFlaming(false);
        RollDice();
    }

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

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

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        RollDice();
    }

    private void Flaming_Checked(object sender, RoutedEventArgs e)
    {
        swordDamage.SetFlaming(true);
        DisplayDamage();
    }

    private void Flaming_Unchecked(object sender, RoutedEventArgs e)
    {
        swordDamage.SetFlaming(false);
        DisplayDamage();
    }

    private void Magic_Checked(object sender, RoutedEventArgs e)
    {
        swordDamage.SetMagic(true);
        DisplayDamage();
    }

    private void Magic_Unchecked(object sender, RoutedEventArgs e)
    {
        swordDamage.SetMagic(false);
        DisplayDamage();
    }
}
```



Ready Bake Code

You've already seen that there are *many different ways* to write the code for a specific program. For most projects in this book it's great if you can find a different—but equally effective—way to solve the problem. But for Owen's damage calculator, we'd like you to enter the code exactly as it appears here because (spoiler alert) **we've included a few bugs on purpose.**

**Read through this code very carefully.
Can you spot any bugs before you run it?**

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

It's game night! Owen's entire gaming party is over, and he's about to unveil his brand-new sword damage calculator. Let's see how that goes.

OK, PARTY,
WE'VE GOT A NEW TABLE RULE.
PREPARE TO BE DAZZLED BY THIS STUNNING NEW FEAT
OF TECHNOLOGICAL AMAZEMENT.

Jayden: Owen, what are you talking about?

Owen: I'm talking about this new app that will calculate sword damage...*automatically*.

Matthew: Because rolling dice is so very, very hard.

Jayden: Come on, people, no need for sarcasm. Let's give it a chance.

Owen: Thank you, Jayden. This is a perfect time, too, because Brittany just attacked the rampaging were-cow with her flaming magic sword. Go ahead, B. Give it a shot.

Brittany: OK. We just started the app. I checked the Magic box. Looks like it's got an old roll in there, let me click roll to do it again, and...

Jayden: Wait, that's not right. Now you rolled 14, but it still says 3 HP. Click it again. Rolled 11 for 3 HP. Click it some more. 9, 10, 5, all give 3 HP. Owen, what's the deal?

Brittany: Hey, it sort of works. If you click roll, then check the boxes a few times, eventually it gives the right answer. Looks like I rolled 10 for 22 HP.

Jayden: You're right. We just have to click things in a **really specific order**. *First* we click roll, *then* we check the right boxes, and *just to be sure* we check the Flaming box twice.

Owen: You're right. If we do things in **exactly that order**, the program works. But if we do it in any other order, it breaks. OK, we can work with this.

Matthew: Or...maybe we can just do things the normal way, with real dice?



Brittany and Jayden are right. The program works, but only if you do things in a specific order. Here's what it looks like when it starts.

Sword Damage	-	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="checkbox"/> Flaming	<input type="checkbox"/> Magic	Roll for damage	
Rolled 10 for 3 HP			

Let's try to calculate damage for a flaming magic sword by checking Flaming first, then Magic second. Uh-oh—that number is wrong.

Sword Damage	-	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> Flaming	<input type="checkbox"/> Magic	Roll for damage	
Rolled 10 for 20 HP			

But once we click the Flaming box twice, it displays the right number.

Sword Damage	-	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> Flaming	<input checked="" type="checkbox"/> Magic	Roll for damage	
Rolled 10 for 22 HP			

Let's try to fix that bug

When you run the program, what's the first thing that it does? Let's take a closer look at this method at the very top of the MainWindow class with the code-behind for the window:

```
public partial class MainWindow : Window
{
    Random random = new Random();
    SwordDamage swordDamage = new SwordDamage();

    public MainWindow() ←
    {
        InitializeComponent();
        swordDamage.SetMagic(false);
        swordDamage.SetFlaming(false);
        RollDice();
    }
}
```

This method is a constructor. It gets called when the MainWindow class is first instantiated, so we can use it to initialize the instance. It doesn't have a return type and its name matches the class name.



When a class has a constructor, it's the very first thing that gets run when a new instance of that class is created. When your app starts up and creates an instance of MainWindow, first it initializes the fields—including creating a new SwordDamage object—and then it calls the constructor. So the program calls RollDice just before showing you the window, and we see the problem every time we click roll, so maybe we can fix this by hacking a solution into the RollDice method. **Make these changes to the RollDice method:**

```
public void RollDice()
{
    swordDamage.Roll = random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
    swordDamage.SetFlaming(flaming.IsChecked.Value);
    swordDamage.SetMagic(magic.IsChecked.Value); ←
    DisplayDamage();
}
```

✓ Redo this!

Calling IsChecked.Value on a checkbox returns true if it's checked or false if it's not.

Now **test your code**. Run your program and click the button a few times. So far so good—the numbers look correct. Now **check the Magic box** and click the button a few more times. OK, it looks like our fix worked! There's just one more thing to test. **Check the Flaming box** and click the button and...**oops!** It's still not working. When you click the button, it does the 1.75 magic multiplier, but it doesn't add the extra 3 HP for flaming. You still need to check and uncheck the Flaming checkbox to get the right number. So the program's still broken.



WE TOOK A GUESS AND QUICKLY WROTE SOME CODE, BUT IT DIDN'T FIX THE PROBLEM BECAUSE WE DIDN'T REALLY THINK ABOUT WHAT ACTUALLY CAUSED THE BUG.

Always think about what caused a bug before you try to fix it.

When something goes wrong in your code, it's really tempting to jump right in and immediately start writing more code to try to fix it. It may feel like you're taking action quickly, but it's way too easy to just add more buggy code. It's always safer to take the time to figure out what really caused the bug, rather than just try to stick in a quick fix.

Use `Debug.WriteLine` to print diagnostic information

In the last few chapters you used the debugger to track down bugs, but that's not the only way developers find problems in their code. In fact, when professional developers are trying to track down bugs in their code, one of the most common things they'll do first is to **add statements that print lines of output**, and that's exactly what we'll do to track down this bug.

String interpolation

You've been using the `+ operator` to concatenate your strings. It's a pretty powerful tool—you can use any value (as long as it's not `null`) and it will safely convert it to a string (usually by calling its `ToString` method). The problem is that concatenation can make your code really hard to read.

Luckily, C# gives us a great tool to concatenate strings more easily. It's called **string interpolation**, and to use it all you need to do is put a dollar sign in front of your string. Then to include a variable, a field, or a complex expression—or even call a method!—you put it inside curly brackets. If you want to include curly brackets in your string, just include two of them, like this: `{}{}`

Open the Output window in Visual Studio by choosing Output (Ctrl+O W) from the View menu. Any text that you print by calling `Console.WriteLine` from a WPF app is displayed in this window. You should only use `Console.WriteLine` for *displaying output your users should see*. Instead, any time you want to print output lines just for debugging purposes you should use **Debug.WriteLine**. The `Debug` class is in the `System.Diagnostics` namespace, so start by adding a `using` line to the top of your `SwordDamage` class file:

```
using System.Diagnostics;
```

Next, **add a `Debug.WriteLine` statement** to the end of the `CalculateDamage` method:

```
public void CalculateDamage()
{
    Damage = (int)(Roll * MagicMultiplier) + BASE_DAMAGE + FlamingDamage;
    Debug.WriteLine($"CalculateDamage finished: {Damage} (roll: {Roll})");
}
```

Now add another `Debug.WriteLine` statement to the end of the `SetMagic` method, and one more to the end of the `SetFlaming` method. They should be identical to the one in `CalculateDamage`, except that they print “`SetMagic`” or “`SetFlaming`” instead of “`CalculateDamage`” to the output:

```
public void SetMagic(bool isMagic)
{
    // the rest of the SetMagic method stays the same
    Debug.WriteLine($"SetMagic finished: {Damage} (roll: {Roll})");
}

public void SetFlaming(bool isFlaming)
{
    // the rest of the SetFlaming method stays the same
    Debug.WriteLine($"SetFlaming finished: {Damage} (roll: {Roll})");
}
```

Now your program will print useful diagnostic information to the Output window.

debugging without a debugger

You can sleuth out this bug without setting any breakpoints. That's something developers do all the time... so you should learn how to do it, too!



Sleuth it out



Let's use the **Output window** to debug the app. Run your program and watch the Output window. As it loads, you'll see a bunch of lines with messages informing you that the CLR loaded various DLLs (that's normal, just ignore them for now).

Once you see the main window, press the Clear All (X) button to clear the Output window. Then check the Flaming box. When we took this screenshot our roll was 9, so this is what it printed:

```
Output
Show output from: Debug
CalculateDamage finished: 12 (roll: 9)
SetFlaming finished: 14 (roll: 9)
```

14 is the correct answer—9 plus base damage of 3 plus 2 for a flaming sword. So far so good.

And you can see from the Output window what happened: the SetFlaming method first called CalculateDamage, which calculated 12. It then added FLAME_DAMAGE for 14, and finally executed the Debug.WriteLine statement that you added.

Now press the button to roll again. The program should write three more lines to the Output window:

```
Output
Show output from: Debug
SetFlaming finished: 17 (roll: 12)
CalculateDamage finished: 15 (roll: 12)
SetMagic finished: 15 (roll: 12)
```

We rolled a 12, so it should calculate 17 HP. So what does the debug output tell us about what happened?

First it called SetFlaming, which set Damage to 17—that's correct: 12 + 3 (base) + 2 (flaming).

But then the program called the CalculateDamage method, which **overwrote the Damage field** and set it back to 15.

The problem is that **SetFlaming was called before CalculateDamage**, so even though it added the flame damage correctly, calling CalculateDamage afterward undid that. So the real reason that the program doesn't work is that the fields and methods in the SwordDamage class need to be used in a very specific order:

1. Set the Roll field to the 3d6 roll.
2. Call the SetMagic method.
3. Call the SetFlaming method.
4. Do not call the CalculateDamage method, because SetFlaming does that for you.

And that's why the console app worked, but the WPF didn't. The console app used the SwordDamage class in the specific way that it works. The WPF app called the methods in the wrong order, so it got incorrect results.

Aha! Now we actually know why the program is broken.

Debug.WriteLine is one of the most basic—and useful!—debugging tools in your developer toolbox. Sometimes the quickest way to sleuth out a bug in your code is to strategically add Debug.WriteLine statements to give you important clues that help you crack the case.



SO THE METHODS JUST NEED TO BE CALLED IN A PARTICULAR ORDER. WHAT'S THE BIG DEAL? I JUST NEED TO FLIP AROUND THE ORDER THAT I CALL THEM, AND MY CODE WILL START WORKING.

People won't always use your classes in exactly the way you expect.

And most of the time those “people” who are using your classes are you! You might be writing a class today that you’ll be using tomorrow, or next month. Luckily, C# gives you a powerful technique to make sure your program always works correctly—even when people do things you never thought of. It’s called **encapsulation** and it’s really helpful for working with objects. The goal of encapsulation is to restrict access to the “guts” of your classes so that all of the class members are **safe to use and difficult to misuse**. This lets you design classes that are much more difficult to use incorrectly—and that’s a **great way to prevent bugs** like the one you sleuthed out in your sword damage calculator.

— there are no
Dumb Questions —

Q: What's the difference between `Console.WriteLine` and `Debug.WriteLine`?

A: The `Console` class is used by console apps to get input from and send output to the user. It uses the three **standard streams** provided by your operating system: standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). Standard input is the text that goes into the program, and standard output is what it prints. (If you’ve ever piped input or output in a shell or command prompt using `<`, `>`, `|`, `<<`, `>>`, or `||` you’ve used `stdin` and `stdout`.) The `Debug` class is in the `System.Diagnostics` namespace, which gives you a hint about its use: it’s for helping diagnose problems by tracking down and fixing them. `Debug.WriteLine` sends its output to **trace listeners**, or special classes that monitor diagnostic output from your program and write them to the console, log files, or a diagnostic tool that collects data from your program for analysis.



We'll do a lot more work with constructors later in the chapter.

For now, just think of a constructor as a special method that you can use to initialize an object.

Q: Can I use constructors in my own code?

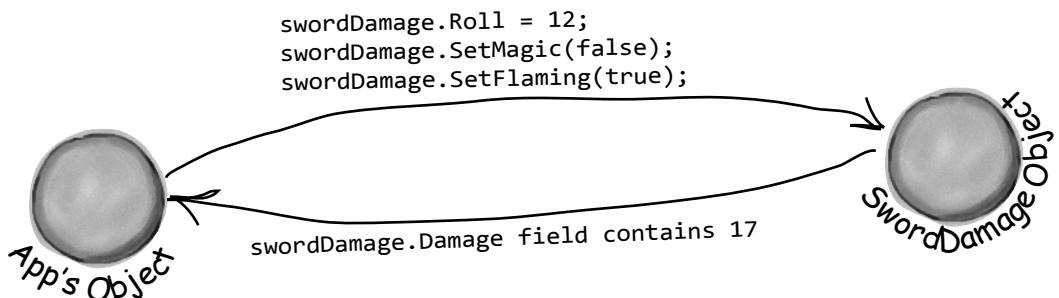
A: Absolutely. A constructor is a method that the CLR calls when it first creates a new instance of an object. It’s just an ordinary method—there’s nothing weird or special about it. You can add a constructor to any class by declaring a method **without a return type** (so no void, int, or other type at the beginning) that has the **same name as the class**. Any time the CLR sees a method like that in a class, it recognizes it as a constructor and calls it any time it creates a new object and puts it on the heap.

It's easy to accidentally misuse your objects

Owen's app ran into problems because we assumed that the CalculateDamage method would, well, calculate the damage. It turned out that **it wasn't safe to call that method directly** because it replaced the Damage value and erased any calculations that were already done. Instead, we needed to let the SetFlaming method call CalculateDamage for us—but **even that wasn't enough**, because we *also* had to make sure that SetMagic was always called first. So even though the SwordDamage class *technically* works, it causes problems when code calls it in an unexpected way.

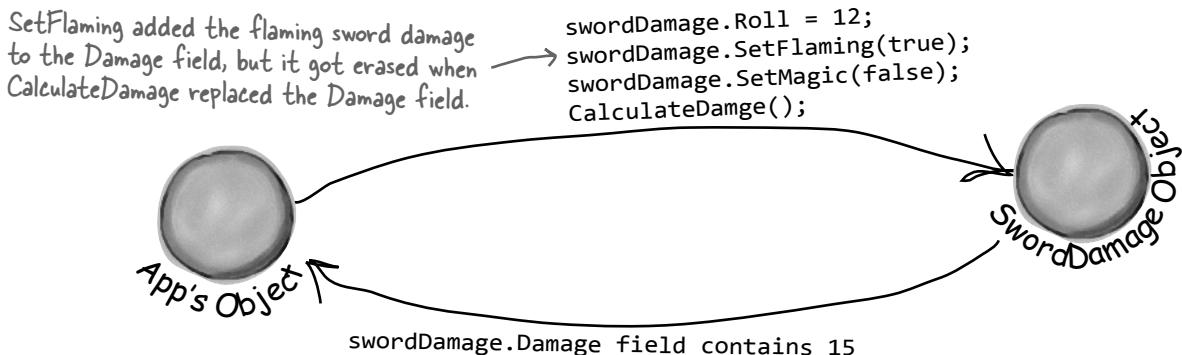
How the SwordDamage class expected to be used

The SwordDamage class gave the app a good method to calculate the total damage for a sword. All it had to do was set the roll, then call the SetMagic method, and finally call the SetFlaming method. If things are done in that order, the Damage field is updated with the calculated damage. But that's not what the app did.



How the SwordDamage class was actually used

Instead, it set the Roll field, then it called SetFlaming, which added the extra damage for the flaming sword to the Damage field. Then it called SetMagic, and finally it called CalculateDamage, which reset the Damage field and discarded the extra flaming damage.



Encapsulation means keeping some of the data in a class private

There's a way to avoid the problem of misusing your objects: make sure that there's only one way to use your class. C# helps you do that by letting you declare some of your fields as **private**. So far, you've only seen public fields. If you've got an object with a public field, any other object can read or change that field. If you make it a private field, then **that field can only be accessed from inside that object** (or by another instance of *the same class*).

```
class SwordDamage
{
    public const int BASE_DAMAGE = 3;
    public const int FLAME_DAMAGE = 2;

    public int Roll;
    private decimal magicMultiplier = 1M; ←
    private int flamingDamage = 0; ←
    public int Damage;

    private void CalculateDamage()
    {
        ...
    }
}
```

By making the CalculateDamage method *private*, we prevent the app from accidentally calling it and resetting the Damage field. Changing the fields involved in the calculation to make them *private* keeps an app from interfering with the calculation. When you make some data *private* and then write code to use that data, it's called *encapsulation*. When a class protects its data and provides members that are safe to use and difficult to misuse, we say that it's *well-encapsulated*.



When in doubt,
make it private.

Worried about trying to figure out which fields and methods to make private? Start by making every member private, and change them to public only if you need to. **In this case, laziness can work to your advantage.** If you leave off the “private” or “public” declaration, then C# will just assume that your field or method is private.

If you want to make a field private, all you need to do is use the **private** keyword when you declare it. This tells C# that if you've got an instance of *SwordDamage*, its *magicMultiplier* and *flamingDamage* fields can only be read and written by methods in an instance of *SwordDamage*. Other objects won't see them at all.

Did you notice that we also changed the *private* field names so they start with lowercase letters?

en-cap-su-la-ted, adj.
enclosed by a protective coating or membrane. *The divers were fully encapsulated by their submersible, and could only enter and exit through the airlock.*

Use encapsulation to control access to your class's methods and fields

When you make all of your fields and methods public, any other class can access them. Everything your class does and knows about becomes an open book for every other class in your program...and you just saw how that can cause your program to behave in ways you never expected.

That's why the `public` and `private` keywords are called **access modifiers**: they modify access to class members. Encapsulation lets you control what you share and what you keep private inside your class. Let's see how this works.

- 1 Super-spy Herb Jones is a *secret agent object in a 1960s spy game* defending life, liberty, and the pursuit of happiness as an undercover agent in the USSR. His object is an instance of the `SecretAgent` class.



RealName: "Herb Jones"
Alias: "Dash Martin"
Password: "the crow flies at midnight"

SecretAgent
Alias
RealName
Password
AgentGreeting

- 2 Agent Jones has a plan to help him evade the enemy agent object. He added an `AgentGreeting` method that takes a password as its parameter. If he doesn't get the right password, he'll only reveal his alias, Dash Martin.
- 3 Seems like a foolproof way to protect the agent's identity, right? As long as the agent object that calls it doesn't have the right password, the agent's name is safe.

EnemyAgent
Borsch
Vodka
ContactComrades
OverthrowCapitalists

This instance of `EnemyAgent` is trying to discover our heroic secret agent's super secret identity.



AgentGreeting("the jeep is parked outside")

"Dash Martin"

The enemy only gets the alias of the secret agent. Perfect! Right?



The enemy agent used the wrong password in his greeting.

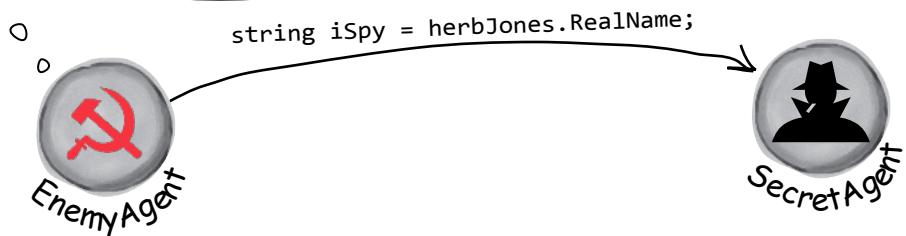
But is the `RealName` field REALLY protected?

So as long as the enemy doesn't know any `SecretAgent` object passwords, the agents' real names are safe. Right? But that doesn't do any good if that data's kept in public fields.

```
public string RealName;
public string Password;
```

Making your fields public means they can be accessed (and even changed!) by any other object.

AHA! HE LEFT THE FIELD PUBLIC! WHY GO THROUGH ALL OF THE TROUBLE TO GUESS THE PASSWORD FOR THE AGENTGREETING METHOD? I CAN JUST GET HIS NAME DIRECTLY!



What can Agent Jones do? He can use **private** fields to keep his identity secret from enemy spy objects. Once he declares the `realName` field as private, the only way to get to it is **by calling methods that have access to the private parts of the class**. So the enemy agent is foiled!

The `EnemyAgent` object can't access the `SecretAgent`'s private fields because they're instances of different classes.

Just replace `public` with `private`, and now the field is hidden from any object that isn't an instance of the same class. Keeping the right fields and methods private makes sure no outside code is going to change values you're using when you don't expect it. We renamed the fields to start with lowercase letters to make our code more readable.

```
private string _realName;
private string _password;
```



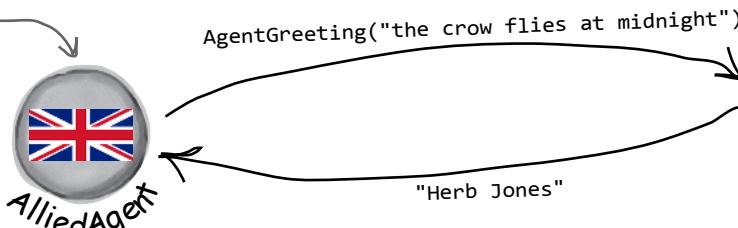
Making the methods and fields in the damage calculator app private prevents bugs by keeping the app from using them directly. **But there's still a problem!** We'll still get the wrong answer if `SetMagic` is called before `SetFlaming`. Can the `private` keyword help you prevent that?

Private fields and methods can only be accessed from instances of the same class

There's only one way that an object can get at the data stored inside another object's private fields: by using the public fields and methods that return the data. EnemyAgent and AlliedAgent agents need to use the AgentGreeting method, but friendly spies that are also SecretAgent instances can see everything...because **any class can see private fields in other instances of the same class.**

Other SecretAgent instances can see the private class members. All other objects have to use the public ones.

The AlliedAgent class represents a spy from an allied country that's allowed to know the secret agent's identity. But the AlliedAgent instance still doesn't have access to the SecretAgent object's private fields. Only another SecretAgent object can see them.



there are no
Dumb Questions

Q: Why would I ever want a field in an object that another object can't read or write?

A: Sometimes a class needs to keep track of information that is necessary for it to operate, but that no other object really needs to see—and you already saw an example of this. In the last chapter you saw that the Random class used special *seed values* to initialize the pseudo-random number generator. Under the hood it turns out that every instance of the Random class actually contains an array of several dozen numbers that it uses to make sure that the Next method always gives you a random number. But that array is private—when you create an instance of Random you can't access that array. If you had access to it, you might be able to put values in it that would cause it to give nonrandom values. So the seeds have been completely encapsulated from you.

Q: OK, so I need to access private data through public methods. What happens if the class with the private field doesn't give me a way to get at that data, but my object needs to use it?

A: Then you can't access the data from outside the object. When you're writing a class, you should always make sure that you give other objects some way to get at the data they need. Private fields are a very important part of encapsulation, but they're only part of the story. Writing a class with good encapsulation means giving a sensible, easy-to-use way for other objects to get the data they need, without giving them access to hijack data your class depends on.

Q: Hey, I just noticed that when I use "Generate method" in the IDE it uses the **private** keyword. Why does it do that?

A: Because it's the safest thing for the IDE to do. Not only are the methods created with "Generate method" private, but when you double-click on a control to add an event handler the IDE creates a private method for that, too. The reason is that it's **safest to make a field or method private** to prevent the kinds of bugs that we saw in the damage calculator. You can always make your class members public later if you need another class to access the data.

The only way that one object can get to data stored in a private field inside another object of a different class is by using public methods that return the data.



Exercise

Let's get a little practice using the **private** keyword by **creating a small Hi-Lo game**. The game starts with a pot of 10 bucks, and it picks a random number from 1 to 10. The player will guess if the next number will be higher or lower. If the player guesses right they win a buck, otherwise they lose a buck. Then the next number becomes the current number, and the game continues.

Go ahead and **create a new console app** for the game. Here's the Main method:

```
public static void Main(string[] args)
{
    Console.WriteLine("Welcome to HiLo.");
    Console.WriteLine($"Guess numbers between 1 and {HiLoGame.MAXIMUM}.");
    HiLoGame.Hint();
    while (HiLoGame.GetPot() > 0)
    {
        Console.WriteLine("Press h for higher, l for lower, ? to buy a hint,");
        Console.WriteLine($"or any other key to quit with {HiLoGame.GetPot()}.");
        char key = Console.ReadKey(true).KeyChar;
        if (key == 'h') HiLoGame.Guess(true);
        else if (key == 'l') HiLoGame.Guess(false);
        else if (key == '?') HiLoGame.Hint();           Don't forget—it's
        else return;                                   not cheating to
                                                       peek at the solution!
    }
    Console.WriteLine("The pot is empty. Bye!");
}
```

Next, add a **static class** called HiLoGame and **add the following members**. Since this is a static class, all of the members need to be static. Make sure to include either **public** or **private** in the declaration for each member:

1. A constant integer **MAXIMUM** that defaults to 10. Remember, you can't use the **static** keyword with constants.
2. An instance of Random called **random**.
3. An integer field called **currentNumber** that's initialized to the first random number to guess.
4. An integer field called **pot** with the number of bucks in the pot. **Make this field private.** ←

We made **pot** private because we don't want other classes to be able to add money, but the **Main** method still needs to be able to print the size of the pot to the console. Look carefully at the code in the **Main** method—can you figure out how to let the **Main** method get the value of the **pot** field without giving it a way to set the field?

5. A **method** called **Guess** with a bool parameter called **higher** that does the following (look closely at the **Main** method to see how it's called):
 - It picks the next random number for the player to guess.
 - If the player guessed higher and the next number is \geq the current number **OR** if the player guessed lower and the next number is \leq the current number, **write** "You guessed right!" to the console and increment the pot.
 - Otherwise, **write** "Bad luck, you guessed wrong." to the console and decrement the pot.
 - **Replace** the current number with the one chosen at the beginning of the method and **writes** "The current number is" followed by the number to the console.
6. A method called **Hint** that finds half the maximum, then writes either "The number is at least {half}" or "The number is at most {half}" to the console and decrements the pot.

BONUS QUESTION: If you make **HiLoGame.random** a public field, can you figure out a way to use what you know about how the Random class generates its numbers **to help you cheat at the game?**



Exercise Solution

Here's the rest of the code for the Hi-Lo game. The game starts with a pot of 10 bucks, and it picks a random number from 1 to 10. The player will guess if the next number will be higher or lower. If the player guesses right they win a buck, otherwise they lose a buck. Then the next number becomes the current number, and the game continues.

Here's the code for the HiLoGame class:

```
static class HiLoGame
{
    public const int MAXIMUM = 10;
    private static Random random = new Random();
    private static int currentNumber = random.Next(1, MAXIMUM + 1);
    private static int pot = 10;

    public static int GetPot() { return pot; } ← The pot field is private, but the Main method can use the GetPot method to get its value without having a way to modify it.

    public static void Guess(bool higher)
    {
        int nextNumber = random.Next(1, MAXIMUM + 1);
        if ((higher && nextNumber >= currentNumber) ||
            (!higher && nextNumber <= currentNumber))
        {
            Console.WriteLine("You guessed right!");
            pot++;
        }
        else
        {
            Console.WriteLine("Bad luck, you guessed wrong.");
            pot--;
        }
        currentNumber = nextNumber;
        Console.WriteLine($"The current number is {currentNumber}");
    }

    public static void Hint()
    {
        int half = MAXIMUM / 2;
        if (currentNumber >= half)
            Console.WriteLine($"The number is at least {half}");
        else Console.WriteLine($"The number is at most {half}");
        pot--;
    }
}
```

This is a good example of encapsulation. You protected the pot field by making it private. It can only be modified by calling the Guess or Hint methods, and the GetPot method provides read-only access.

↑
This is an important point.
Take a few minutes to really figure out how it works.

The Hint method needs to be public because it's called from Main. Notice how we didn't include the curly brackets for the if/else statement? An if or else clause that only has a single line doesn't need brackets.

BONUS: You can replace the public random field with a new instance of Random that you **initialized with a different seed**. Then you can use a new instance of Random with the same seed to find the numbers in advance!

```
HiLoGame.random = new Random(1);
Random seededRandom = new Random(1);
Console.Write("The first 20 numbers will be: ");
for (int i = 0; i < 20; i++)
    Console.Write($"{seededRandom.Next(1, HiLoGame.MAXIMUM + 1)}, ");
```

Every instance of Random initialized with the same seed will generate the same sequence of pseudo-random numbers.



SOMETHING'S REALLY NOT RIGHT HERE.
IF I MAKE A FIELD PRIVATE, ALL THAT DOES IS KEEP
MY PROGRAM FROM COMPILEING IF I'M USING IT IN
ANOTHER CLASS. BUT IF I JUST CHANGE THE "PRIVATE" TO
"PUBLIC" MY PROGRAM BUILDS AGAIN! ADDING "PRIVATE"
CAN ONLY EVER BREAK MY PROGRAM.

SO WHY WOULD I EVER WANT TO
MAKE A FIELD PRIVATE?

Because sometimes you want your class to hide information from the rest of the program.

A lot of people find encapsulation a little odd the first time they come across it because the idea of hiding one class's fields, properties, or methods from another class is a little counterintuitive. There are some very good reasons that you'll want to think about what information should be exposed to the rest of the program.

Encapsulation means having one class hide information from another. It helps you prevent bugs in your programs.



Watch it!

Encapsulation is not the same as security. Private fields are not secure.

If you're building a game with 1960s spies, encapsulation is a great way to prevent bugs. If you're building a program for real spies, encapsulation is a terrible way to protect their data. For example, go back to your Hi-Lo game. Place a breakpoint on the first line of the Main method, add a watch for `HiLoGame.random`, and debug the program. If you **expand the Non-Public Members section** you can see all of the internals of the Random class, including an array called `_seedArray` that it uses to generate its pseudo-random numbers.

It's not just the IDE that can see your objects' *privates*. .NET has a tool called **reflection** that lets you write code to access objects in memory and look at their contents, even private fields. Here's a quick example of how it works. **Create a new console app** and add a class called HasASecret:

```
class HasASecret
{
    // This class has a secret field. Does the private keyword make it secure?
    private string secret = "xyzzy";
}
```

The reflection classes are in the **System.Reflection namespace**, so add this `using` statement to the file with the Main method:

```
using System.Reflection;
```

Here's the main class with a Main method that creates a new instance of HasASecret, and then uses reflection to read its `secret` field. It calls the `GetType` method, which is a method that you can call from any object to get information about its type:

```
class MainClass
{
    public static void Main(string[] args)
    {
        HasASecret keeper = new HasASecret();

        // Uncommenting this Console.WriteLine statement causes a compiler error:
        // 'HasASecret.secret' is inaccessible due to its protection level
        // Console.WriteLine(keeper.secret);

        // But we can still use reflection to get the value of the secret field
        FieldInfo[] fields = keeper.GetType().GetFields(
            BindingFlags.NonPublic | BindingFlags.Instance);

        // This foreach loop will cause "xyzzy" to be printed to the console
        foreach (FieldInfo field in fields)
        {
            Console.WriteLine(field.GetValue(keeper));
        }
    }
}
```

Every object has a `GetType` method that returns a `Type` object. The `Type.GetFields` method returns an array of `FieldInfo` objects, one for each of its fields. Each `FieldInfo` object contains information about its fields. If you call its `GetValue` method with an instance of an object, it will return the value stored in that object's field—even if the field is private.

Why encapsulation? Think of an object as a black box...

Sometimes you'll hear a programmer refer to an object as a "black box," and that's a pretty good way of thinking about them. When we say something is a black box, we're saying that we can see how it behaves, but we have no way of knowing how it actually works.

When you call an object's method, you don't really care how that method works—at least, not right now. All you care about is that it takes the inputs you gave it and does the right thing.



When developers talk about a "black box" we mean something that hides any internal mechanisms so you don't need to know how it works to use it. If it just does one thing, and you don't need to give it any parameters, it's the code equivalent of a black box with a single button on it.

You *could* include a lot more controls, like a window that shows you what's going on inside the box, and knobs and dials that let you muck with its internals. But if they don't actually do anything that your system needs, then they don't do you any good and can only cause problems.

Encapsulation makes your classes...

★ Easier to use

You already know that classes use fields to keep track of their state. Many of them use methods to keep those fields up to date—methods that no other class will ever call. It's pretty common to have a class that has fields, methods, and properties that will never be called by any other class. If you make those members private, then they won't show up in the IntelliSense window later when you need to use that class. Less clutter in the IDE will make your class easier to use.

★ Less prone to bugs

That bug in Owen's program happened because the app accessed a method directly rather than letting the other methods in the class call it. If that method had been private, we could have avoided that bug.

★ Flexible

A lot of times, you'll want to go back and add features to a program you wrote a while ago. If your classes are well-encapsulated, then you'll know exactly how to use them and add on to them later.



How could building a poorly encapsulated class now make your programs harder to modify later?

A few ideas for encapsulating classes

★ Is everything in your class public?

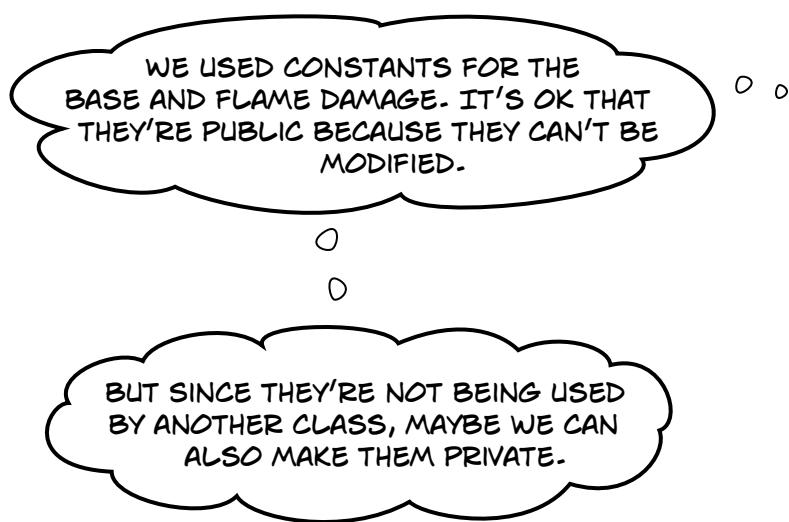
If your class has nothing but public fields and methods, you probably need to spend a little more time thinking about encapsulation.

★ Think about ways fields and methods can be misused.

What can go wrong if they're not set or called properly?

★ What fields require some processing or calculation to happen when they're set?

Those are prime candidates for encapsulation. If someone writes a method later that changes the value in any one of them, it could cause problems for the work your program is trying to do.



★ Only make fields and methods public if you need to.

If you don't have a reason to declare something public, don't—you could make things really messy for yourself by making all of the fields in your program public. But don't just go making everything private, either. Spending a little time up front thinking about which fields really need to be public and which don't can save you a lot of time later.



A WELL-ENCAPSULATED CLASS DOES
**EXACTLY THE SAME THING AS ONE THAT HAS POOR
ENCAPSULATION!**

Exactly! The difference is that the well-encapsulated one is built in a way that prevents bugs and is easier to use.

It's easy to take a well-encapsulated class and turn it into a poorly encapsulated class: do a search and replace to change every occurrence of `private` to `public`.

And that's a funny thing about the `private` keyword: you can generally take any program and do that search and replace, and it will still compile and work in exactly the same way. That's one reason that encapsulation can be a little difficult for some programmers to really "get" when they first see it.

When you come back to code that you haven't looked at in a long time, it's easy to forget how you intended it to be used. That's where encapsulation can make your life a lot easier!

This book so far has been about making programs **do things**—perform certain behaviors. Encapsulation is a little different. It doesn't change the way your program behaves. It's more about the "chess game" side of programming: by hiding certain information in your classes when you design and build them, you set up a strategy for how they'll interact later. The better the strategy, the **more flexible and maintainable** your programs will be, and the more bugs you'll avoid.



And just like chess, there
are an almost unlimited
number of possible
encapsulation strategies!

**If you encapsulate
your classes well
today, that makes
them a lot easier
to reuse tomorrow.**

BULLET POINTS

- Always **think about what caused a bug** before you try to fix it. Take the time to really understand what's going on.
- Adding statements that print lines of output can be an effective debugging tool. Use **Debug.WriteLine** when you add statements to print diagnostic information.
- A **constructor** is a method that the CLR calls when it first creates a new instance of an object.
- **String interpolation** makes string concatenation more readable. Use it by adding a \$ in front of a string and including values in {curly brackets}.
- The System.Console class writes its output to **standard streams** that provide the input and output for console apps.
- The System.Diagnostics.Debug class writes its output to **trace listeners**—special classes that perform specific actions with diagnostic output—including one that writes that output to the IDE's Output (Windows) or Application Output (macOS) window.
- People won't always use your classes in exactly the way you expect. **Encapsulation** is a technique for making your class members flexible and difficult to misuse.
- Encapsulation usually involves using the **private** keyword to keep some of the fields or methods in a class private so they can't be misused by other classes.
- When a class protects its data and provides members that are safe to use and difficult to misuse, we say that it's **well-encapsulated**.



OK, WE KNOW THE CODE FOR OUR SWORD DAMAGE APP HAS SOME PROBLEMS. WHAT CAN WE DO ABOUT IT?

SwordDamage
Roll
MagicMultiplier
FlamingDamage
Damage
CalculateDamage
SetMagic
SetFlaming

← Remember how you used `Debug.WriteLine` earlier in the chapter to sleuth out the bug in your app? You discovered that the `SwordDamage` class only works if its methods are called in a very specific order. This chapter is all about encapsulation, so it's a pretty safe bet that you'll use encapsulation later in the chapter to fix this problem. But...how, exactly?

Let's use encapsulation to improve the SwordDamage class

We just covered some great ideas for encapsulating classes. Let's see if we can start to apply those ideas to the SwordDamage class to keep it from being confused, misused, and abused by any app that we include it in.

Is every member of the SwordDamage class public?

Yes, indeed. The four fields (Roll, MagicMultiplier, FlamingDamage, and Damage) are public, and so are the three methods (CalculateDamage, SetMagic, and SetFlaming). We could stand to think about encapsulation.

Are fields or methods being misused?

Absolutely. In the first version of the damage calculator app, we called CalculateDamage when we should have just let the SetFlaming method call it. Even our attempt to fix it failed because we misused the methods by calling them in the wrong order.

Is there calculation required after setting a field?

Certainly. After setting the Roll field, we really want the instance to calculate damage immediately.

So what fields and methods really need to be public?

That's a great question. Take some time to think about the answer. We'll tackle it at the end of the chapter.

Making members of a class private can prevent bugs caused by other classes calling its public methods or updating its public fields in unexpected ways.



Think about those questions, then take another look at how the SwordDamage class works. What would you do to fix the SwordDamage class?

this is encapsulated but we can do better

Encapsulation keeps your data safe

We've seen how the **private** keyword **protects class members from being accessed directly**, and how that can prevent bugs caused by other classes calling methods or updating fields in ways we didn't expect—like how your GetPot method in the Hi-Lo game gave read-only access to the private pot field, and only the Guess or Hint methods could modify it. This next class works in exactly the same way.

Let's use encapsulation in a class

Let's build a PaintballGun class for a paintball arena video game. The player can pick up magazines of paintballs and reload at any time, so we want the class to keep track of the total number of balls the player has *and* the number of balls currently loaded. We'll add a method to check if the gun is empty and needs to be reloaded. We also want it to keep track of the magazine size. Any time the player gets more ammo we want the gun to automatically reload a full magazine, so we'll make sure that always happens by providing a method to set the number of balls that calls the Reload method.

```
class PaintballGun
{
    public const int MAGAZINE_SIZE = 16;           ← We'll keep this constant public because
    private int balls = 0;                          it's going to be used by the Main method.
    private int ballsLoaded = 0;

    public int GetBallsLoaded() { return ballsLoaded; }

    public bool IsEmpty() { return ballsLoaded == 0; }

    public int GetBalls() { return balls; }

    public void SetBalls(int numberOfBalls)
    {
        if (numberOfBalls > 0)
            balls = numberOfBalls;
        Reload();
    }

    public void Reload()
    {
        if (balls > MAGAZINE_SIZE)
            ballsLoaded = MAGAZINE_SIZE;
        else
            ballsLoaded = balls;
    }

    public bool Shoot()
    {
        if (ballsLoaded == 0) return false;
        ballsLoaded--;
        balls--;
        return true;
    }
}
```

When the game needs to display the number of balls left and the number of balls loaded in the UI, it can call the GetBalls and GetBallsLoaded methods.

The game needs to be able to set the number of balls. The SetBalls method protects the balls field by only allowing the game to set a positive number of balls. Then it calls Reload to automatically reload the gun.

The only way to reload the gun is to call the Reload method, which loads the gun with a full magazine, or the remaining number of balls if there isn't a full magazine's worth. This keeps the balls and ballsLoaded fields from getting out of sync.

The Shoot method returns true and decrements the balls field if the gun is loaded, or false if it isn't.

Does the IsEmpty method make code that calls this class easier to read? Or is it redundant? There's no right or wrong answer—you could argue either side.



Write a console app to test the PaintballGun class

Let's try out our new PaintballGun class. **Create a new console app** and add the PaintballGun class to it. Here's the Main method—it uses a loop to call the various methods in the class:

```
static void Main(string[] args)
{
    PaintballGun gun = new PaintballGun();
    while (true)
    {
        Console.WriteLine($"{gun.GetBalls()} balls, {gun.GetBallsLoaded()} loaded");
        if (gun.IsEmpty()) Console.WriteLine("WARNING: You're out of ammo");
        Console.WriteLine("Space to shoot, r to reload, + to add ammo, q to quit");
        char key = Console.ReadKey(true).KeyChar;
        if (key == ' ') Console.WriteLine($"Shooting returned {gun.Shoot()}");
        else if (key == 'r') gun.Reload();
        else if (key == '+') gun.SetBalls(gun.GetBalls() + PaintballGun.MAGAZINE_SIZE);
        else if (key == 'q') return;
    }
}
```

A console app with a loop that tests an instance of a class should be really familiar by now. Make sure you can read the code and understand how it works.

Our class is well-encapsulated, but...

The class works, and we encapsulated it pretty well. The **balls field is protected**: it doesn't let you set a negative number of balls, and it stays in sync with the ballsLoaded field. The Reload and Shoot methods work as expected, and there don't seem to be any *obvious* ways we could accidentally misuse this class.

But have a closer look at this line from the Main method:

```
else if (key == '+') gun.SetBalls(gun.GetBalls() + PaintballGun.MAGAZINE_SIZE);
```

Let's be honest—that's a downgrade from a field. If we still had a field, we could use the `+ =` operator to increase it by the magazine size. Encapsulation is great, but we don't want it to make our class annoying or difficult to use.

Is there a way to keep the balls field protected but still get the convenience of `+ =`?

Use different cases for private and public fields

We used camelCase for the private fields and PascalCase for the public ones. PascalCase means capitalizing the first letter in every word in the variable name. camelCase is similar to PascalCase, except that the first letter is lowercase. It's called camelCase because it makes the uppercase letters look like "humps" of a camel.

Using different cases for public and private fields is a convention a lot of programmers follow. Your code is easier to read if you use consistent case when choosing names for fields, properties, variables, and methods.

Properties make encapsulation easier

So far you've learned about two kinds of class members, methods and fields. There's a third kind of class member that helps you encapsulate your classes: they **property**. A property is a class member that **looks like a field** when it's used, but it **acts like a method** when it runs.

A property is declared just like a field, with a type and a name, except instead of ending with a semicolon it's followed by curly brackets. Inside those brackets are **property accessors**, or methods that either return or set the property value. There are two types of accessors:

- ★ A **get property accessor**, usually just referred to as a **get accessor** or **getter**, that returns the value of the property. It starts with the **get** keyword, followed by a method inside curly brackets. The method must return a value that matches the type in the property declaration.
- ★ A **set property accessor**, usually just referred to as a **set accessor** or **setter**, that sets the value of the property. It starts with the **set** keyword, followed by a method inside curly brackets. Inside the method, the **value** keyword is a read-only variable that contains the value being set.

It is very common for a property to get or set a **backing field**, which is what we call a private field that's encapsulated by restricting access to it through a property.

Replace
this!

Replace the GetBalls and SetBalls methods with a property

Here are the GetBalls and SetBalls methods from your PaintballGun class:

```
public int GetBalls() { return balls; }

public void SetBalls(int numberofBalls)
{
    if (numberofBalls > 0)
        balls = numberofBalls;
    Reload();
}
```

Let's replace them with a property. **Delete both methods**. Then **add this Balls property**:

```
public int Balls
{
    get { return balls; }
    set
    {
        if (value > 0)
            balls = value;
        Reload();
    }
}
```

This is the declaration. It says that the name of the property is Balls, and its type is int.

The get accessor (or getter) is identical to the GetBalls method that it replaced.

The set accessor (or setter) is almost identical to the SetBalls method. The only difference is it uses the value keyword where SetBalls used its parameter. The value keyword will always contain the value being assigned by the set accessor.

The old SetBalls method took an int parameter called numberofBalls with the new value for the backing field. The setter uses the "value" keyword everywhere the SetBalls method used numberofBalls.

Modify your Main method to use the Balls property

Now that you've replaced the GetBalls and SetBalls methods with a single property called Balls, your code won't build anymore. You need to update the Main method to use the Balls property instead of the old methods.

The GetBalls method was called in this `Console.WriteLine` statement:

```
Console.WriteLine($"{gun.GetBalls()} balls, {gun.GetBallsLoaded()} loaded");
```

You can fix that by **replacing `GetBalls()` with `Balls`**—when you do this, the statement will work just like before. Let's have a look at the other place where GetBalls and SetBalls were used:

```
else if (key == '+') gun.SetBalls(gun.GetBalls() + PaintballGun.MAGAZINE_SIZE);
```

This was that messy line of code that looked ugly and clunky. Properties are really useful because they work like methods but you use them like fields. So let's use the Balls property like a field—**replace that line** with this statement that uses the `+=` operator exactly like it would if Balls were a field:

```
else if (key == '+') gun.Balls += PaintballGun.MAGAZINE_SIZE;
```

Here's the updated Main method:

```
static void Main(string[] args)
{
    PaintballGun gun = new PaintballGun();
    while (true)
    {
        Console.WriteLine($"{gun.Balls} balls, {gun.GetBallsLoaded()} loaded");
        if (gun.IsEmpty()) Console.WriteLine("WARNING: You're out of ammo");
        Console.WriteLine("Space to shoot, r to reload, + to add ammo, q to quit");
        char key = Console.ReadKey(true).KeyChar;
        if (key == ' ') Console.WriteLine($"Shooting returned {gun.Shoot()}");
        else if (key == 'r') gun.Reload();
        else if (key == '+') gun.Balls += PaintballGun.MAGAZINE_SIZE;
        else if (key == 'q') return;
    }
}
```

Debug your PaintballGun class to understand how the property works

Use the debugger to really get a good sense of how your new Ball property works:

- ★ Place a breakpoint inside the curly brackets of the get accessor (`return balls;`).
- ★ Place another breakpoint on the first line of the set accessor (`if (value > 0)`).
- ★ Place a breakpoint at the top of the Main method and start debugging. Step over each statement.
- ★ When you step over `Console.WriteLine`, the debugger will hit the breakpoint in the getter.
- ★ Keep stepping over methods. When you execute the `+=` statement, the debugger will hit the breakpoint in the setter. Add a watch for the backing field **`balls`** and the **`value`** keyword.

Update
this!

Auto-implemented properties simplify your code



A very common way to use a property is to create a backing field and provide get and set accessors for it. Let's create a new BallsLoaded property that **uses the existing ballsLoaded field** as a backing field:

```
private int ballsLoaded = 0;

public int BallsLoaded {
    get { return ballsLoaded; }
    set { ballsLoaded = value; }
}
```

This property uses a private backing field.
Its getter returns the value in the field,
and its setter updates the field.

Now you can **delete the GetBallsLoaded method** and modify your Main method to use the property:

```
Console.WriteLine($"{gun.Balls} balls, {gun.BallsLoaded} loaded");
```

Run your program again. It should still work exactly the same way.

Use the prop snippet to create an auto-implemented property

An **auto-implemented property**—sometimes called an **automatic property** or **auto-property**—is a property that has a getter that returns the value of the backing field, and a setter that updates it. In other words, it works just like the BallsLoaded property that you just created. There's one important difference: when you create an automatic property **you don't define the backing field**. Instead, the C# compiler creates the backing field for you, and the only way to update it is to use the get and set accessors.

Visual Studio gives you a really useful tool for creating automatic properties: a **code snippet**, or a small, reusable block of code that the IDE inserts automatically. Let's use it to create a BallsLoaded auto-property.

- 1 Remove the BallsLoaded property and backing field.** Delete the BallsLoaded property you added, because we're going to replace it with an auto-implemented property. Then delete the ballsLoaded backing field (`private int ballsLoaded = 0;`) too, because any time you create an automatic property the C# compiler generates a hidden backing field for you.

- 2 Tell the IDE to start the prop snippet.** Put your cursor where the field used to be, and then **type prop and press the Tab key twice** to tell the IDE to start a snippet. It will add this line to your code:

```
public int MyProperty { get; set; }
```

The snippet is a template that lets you edit parts of it—the prop snippet lets you edit the type and the property name. Press the Tab key once to switch to the property name, then **change the name to BallsLoaded** and press Enter to finalize the snippet

```
public int BallsLoaded { get; set; }
```

You don't have to declare a backing field
for an automatic property because the
C# compiler creates it automatically.

- 3 Fix the rest of the class.** Since you removed the ballsLoaded field, your PaintballGun class doesn't compile anymore. That has a quick fix—the `ballsLoaded` field appears five times in the code (once in the IsEmpty method, and twice in the Reload and Shoot methods). Change them to `BallsLoaded`—now your program works again.

Use a private setter to create a read-only property

Let's take another look at the auto-implemented property that you just created:

```
public int BallsLoaded { get; set; }
```

This is definitely a great replacement for a property with get and set accessors that just update a backing field. It's more readable and has less code than the ballsLoaded field and GetBallsLoaded method. So that's an improvement, right?

But there's one problem: ***we've broken the encapsulation.*** The whole point of the private field and public method was to make the number of balls loaded read-only. The Main method could easily set the BallsLoaded property. We made the field private and created a public method to get the value so that it could only be modified from inside the PaintballGun class.

Make the BallsLoaded setter private

Luckily, there's a quick way to make our PaintballGun class well-encapsulated again. When you use a property, you can put an access modifier in front of the **get** or **set** keyword.

You can make a **read-only property** that can't be set by another class by making its set accessor **private**. In fact, you can leave out the set accessor entirely for normal properties—but not automatic properties, which *must* have a set accessor or your code won't compile.

You can make your automatic property read-only by making its setter private.

So let's **make the set accessor private**:

```
public int BallsLoaded { get; private set; }
```



Now the BallsLoaded field is a **read-only property**. It can be read anywhere, but it can only be updated from inside the PaintballGun class. The PaintballGun class is well-encapsulated again.

there are no Dumb Questions

Q: We replaced methods with properties. Is there a difference between how a method works and how a getter or setter works?

A: No. Get and set accessors are a special kind of method—they look just like a field to other objects, and are called whenever that “field” is set. Getters always return a value that’s the same type as the field. A setter works just like a method with one parameter called **value** whose type is the same as the field.

Q: So you can have ANY kind of statement in a property?

A: Absolutely. Anything you can do in a method, you can do in a property—you can even include complicated logic that does anything you can do in a normal method. A property can call other methods, access other fields, even create instances of objects. Just remember that they only get called when a property gets accessed, so they should only include statements that have to do with getting or setting the property.

Q: Why would I need complicated logic in a get or set accessor? Isn’t it just a way of modifying fields?

A: Because sometimes you know that every time you set a field, you’ll have to do some calculation or perform some action. Think about Owen’s problem—he ran into trouble because the app didn’t call the SwordDamage methods in the right order after setting the Roll field. If we replaced all of the methods with properties, then we could make sure the setters do the damage calculation correctly. (In fact, you’re about to do exactly that at the end of the chapter!)

What if we want to change the magazine size?

Replace
this!

Right now the PaintballGun class uses a `const` for the magazine size:

```
public const int MAGAZINE_SIZE = 16;
```

What if we want the game to set the magazine size when it instantiates the gun? Let's **replace it with a property**.

- ➊ Remove the `MAGAZINE_SIZE` constant and replace it with a read-only property.

```
public int MagazineSize { get; private set; }
```

- ➋ Modify the `Reload` method to use the new property.

```
if (balls > MagazineSize)  
    BallsLoaded = MagazineSize;
```

- ➌ Fix the line in the `Main` method that adds the ammo.

```
else if (key == '+') gun.Balls += gun.MagazineSize;
```

But there's a problem...how do we initialize `MagazineSize`?

The `MAGAZINE_SIZE` constant used to be set to 16. Now we've replaced it with an auto-property, and if we want, we can initialize it to 16 just like a field by **adding an assignment to the end of the declaration**:

```
public int MagazineSize { get; private set; } = 16;
```

But what if we want the game to be able to specify the number of balls in the magazine? Maybe most guns are spawned loaded, but in some rapid onslaught levels we want some guns to spawn unloaded so the player needs to reload before firing. **How do we do that?**

Q: Can you explain what a constructor is again?

there are no
Dumb Questions

A: A **constructor** is a method that's called when a new instance of a class is created. It's always declared as a method with **no return type** and a name that **matches the class name**. To see how it works, **create a new console app** and add this `ConstructorTest` class with a constructor and a public field called `i`:

```
public class ConstructorTest  
{  
    public int i = 1;  
  
    public ConstructorTest()  
    {  
        Console.WriteLine($"i is {i}");  
    }  
}
```

Use the debugger to really understand how the constructor works.

Add three breakpoints:

- At the field declaration (on `i = 1`)
- On the first line of the constructor
- On the bracket `}` after the last line of the Main method

The debugger will first break at the field declaration, then in the constructor, and finally at the end of the Main method. There's no mystery here—the CLR initializes the fields first, then runs the constructor, and finally picks up where it left off after the new statement.

Then add this new statement to the Main method: `new ConstructorTest();`

Use a constructor with parameters to initialize properties

You saw earlier in the chapter that you can initialize an object with a constructor, or a special method that's called when the object is first instantiated. Constructors are just like any other method—which means they can have **parameters**. We'll use a constructor with parameters to initialize the properties.

The constructor you just created in the Q&A answer looks like this: `public ConstructorTest()`. That's a **parameterless constructor**, so just like any other method without parameters the declaration ends with `()`. Now let's **add a constructor with parameters** to the PaintballGun class. Here's the constructor to add:

```
Add a constructor to a class by
creating a method that has the same
name as the class and no return type.

This constructor takes three parameters,
an int called balls, an int called
magazineSize, and a bool called loaded.

public PaintballGun(int balls, int magazineSize, bool loaded)
{
    this.balls = balls;
    MagazineSize = magazineSize;
    if (!loaded) Reload();
}
```

The constructor runs as soon as a new instance is created, so we put code into the body of the method to set the number of balls and magazine size, and reload the gun if needed. Notice the this keyword in the first line. Why do you think we need to use it?

Uh-oh—there's a problem. As soon as you add the constructor, the IDE will tell you that the Main method has an error:

 CS7036 There is no argument given that corresponds to the required formal parameter 'bullets' of 'MachineGun.MachineGun(int, int, bool)'

What do you think we need to do to fix this error?



Watch it!

When a parameter has the same name as a field, it masks the field.

The constructor's `balls` parameter has the same name as the field called `balls`. Since they have the same name, the parameter takes precedence inside the body of the constructor. That's called **masking**—when a parameter or a variable in a method has the same name as a field, using that name in the method refers to the parameter or variable, not the field. That's why we need to use the `this` keyword in the `PaintballGun` constructor:

`this.balls = balls;`

When we just use `balls` it refers to the parameter. We want to set the field, and since it has the same name, we need to use `this.balls` to refer to the field.

And by the way, this doesn't just apply to constructors. It's true for **any** method.

Specify arguments when you use the “new” keyword

When you added the constructor, the IDE told you that the Main method has an error on the new statement (`PaintballGun gun = new PaintballGun()`). Here’s what that error looks like:

 CS7036 There is no argument given that corresponds to the required formal parameter 'bullets' of 'MachineGun.MachineGun(int, int, bool)'

Read the text of the error—it’s telling you exactly what’s wrong. Your constructor now takes arguments, so it needs parameters. Start typing the new statement again, and the IDE will tell you exactly what you need to add:

```
MachineGun gun = new MachineGun()  
    MachineGun(int bullets, int magazineSize, bool loaded)
```

You’ve been using new to create instances of classes. So far, all of your classes have had parameterless constructors, so you never needed to provide any arguments.

Now you have a constructor with parameters, and like any method with parameters, it requires you to specify arguments with types that match those parameters.

Let’s modify your Main method to **pass parameters to the PaintballGun constructor**.

Modify
this!

1 Add the ReadInt method that you wrote for Owen’s ability score calculator in Chapter 4.

You need to get the arguments for the constructor from somewhere. You already have a perfectly good method that prompts the user for int values, so it makes sense to reuse it here.

2 Add code to read values from the console input.

Now that you’ve added the ReadInt method from Chapter 4, you can use it to get two int values. Add these four lines of code to the top of your Main method:

```
int numberofBalls = ReadInt(20, "Number of balls");  
int magazineSize = ReadInt(16, "Magazine size");  
  
Console.WriteLine("Loaded [false]: ");  
bool.TryParse(Console.ReadLine(), out bool isLoaded);
```

If TryParse can’t parse the line, it will leave isLoaded with the default value, which for a bool is false.

3 Update the new statement to add arguments.

Now that you have values in variables with types that match the parameters in the constructor, you can update the new statement to pass them to the constructor as arguments:

```
PaintballGun gun = new PaintballGun(numberofBalls, magazineSize, isLoaded);
```

4 Run your program.

Now run your program. It will prompt you for the number of balls, the magazine size, and whether or not the gun is loaded. Then it will create a new instance of PaintballGun, passing arguments to its constructor that match your choices.



Pool Puzzle

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You **may**

use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make classes that will compile and run and produce output that matches the sample.

This program is a math quiz game that asks a series of random multiplication or addition questions and checks the answer. Here's what it looks like when you play it:

8 + 5 = 13

Right!

4 * 6 = 24

Right!

4 * 9 = 37

Wrong! Try again.

4 * 9 = 36

Right!

9 * 8 = 72

Right!

6 + 5 = 12

Wrong! Try again.

6 + 5 = 9

Wrong! Try again.

6 + 5 = 11

Right!

8 * 4 = 32

Right!

8 + 6 = Bye

Thanks for playing!

The game generates random addition or multiplication questions.

If you get a question wrong, it keeps asking until you get it right.

The game ends when you enter an answer that isn't a number.

Note: each snippet from the pool can be used more than once!

a	Q
b	add
c	Main
i	Op
j	Random
k	R
q	N1
r	N2
s	out
Next()	
Next(1, 10)	
Next(2)	
Next(1, 9)	
Check	

```

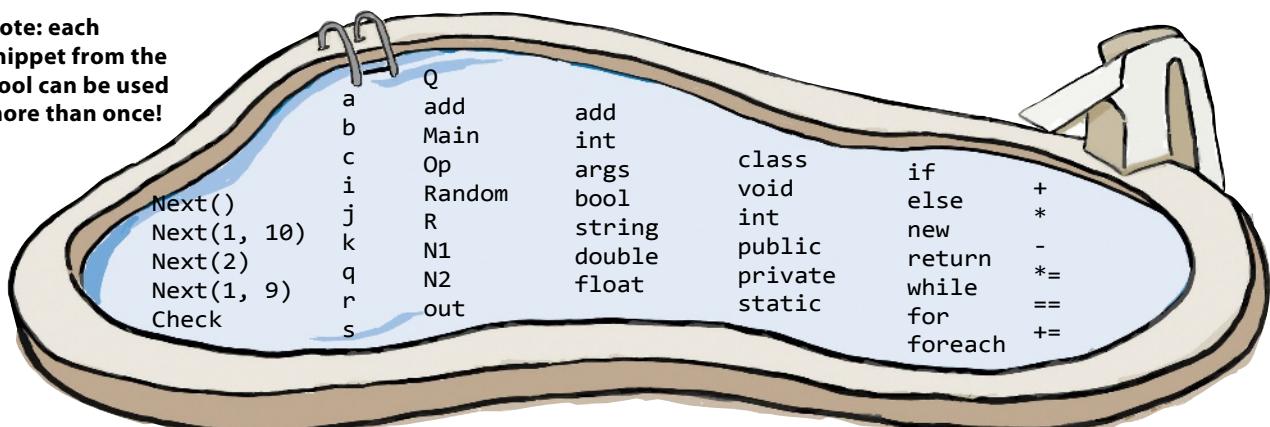
class Q {
    public Q(bool add) {
        if (add) _____ = "+";
        else _____ = "*";
        N1 = _____;
        N2 = _____;
    }

    public _____ Random R = new Random();
    public _____ N1 { get; _____ set; }
    public _____ Op { get; _____ set; }
    public _____ N2 { get; _____ set; }

    public _____ Check(int _____)
    {
        if (_____ == "+") return (a _____ N1 + N2);
        else return (a _____ _____ * _____);
    }
}

class Program {
    public static void Main(string[] args) {
        Q _____ = _____ Q(_____.R._____ == 1);
        while (true) {
            Console.WriteLine($"{q._____} {q._____} {q._____} = ");
            if (!int.TryParse(Console.ReadLine(), out int i)) {
                Console.WriteLine("Thanks for playing!");
                _____;
            }
            if (_____.(_____)(_____)) {
                Console.WriteLine("Right!");
                _____ = _____ Q(_____.R._____ == 1);
            }
            else Console.WriteLine("Wrong! Try again.");
        }
    }
}

```



We leveled up the difficulty for this puzzle! Remember, it's not cheating to peek at the solution if you get stuck.

this puzzle is tough but you can do it!

```
class Q {
    public Q(bool add) {
        if (add) Op = "+";
        else Op = "*";
        N1 = R.Next(1, 10);
        N2 = R.Next(1, 10);
    }

    public static Random R = new Random();
    public int N1 { get; private set; }
    public string Op { get; private set; }
    public int N2 { get; private set; }

    public bool Check(int a)
    {
        if (Op == "+") return (a == N1 + N2);
        else return (a == N1 * N2);
    }
}

class Program {
    public static void Main(string[] args) {
        Q q = new Q(Q.R.Next(2) == 1);
        while (true) {
            Console.WriteLine($"{q.N1} {q.Op} {q.N2} = ");
            if (!int.TryParse(Console.ReadLine(), out int i)) {
                Console.WriteLine("Thanks for playing!");
                return;
            }
            if (q.Check(i)) {
                Console.WriteLine("Right!");
                q = new Q(Q.R.Next(2) == 1);
            }
            else Console.WriteLine("Wrong! Try again.");
        }
    }
}
```

Note: each snippet from the pool can be used more than once!

We put a check next to each snippet that was used in the solution.



Pool Puzzle Solution

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make classes that will compile and run and produce output that matches the sample.

This program is a math quiz game that asks a series of random multiplication or addition questions and checks the answer. Here's what it looks like when you play it:

8 + 5 = 13

Right!

4 * 6 = 24

Right!

4 * 9 = 37

Wrong! Try again.

4 * 9 = 36

Right!

9 * 8 = 72

Right!

6 + 5 = 12

Wrong! Try again.

6 + 5 = 9

Wrong! Try again.

6 + 5 = 11

Right!

8 * 4 = 32

Right!

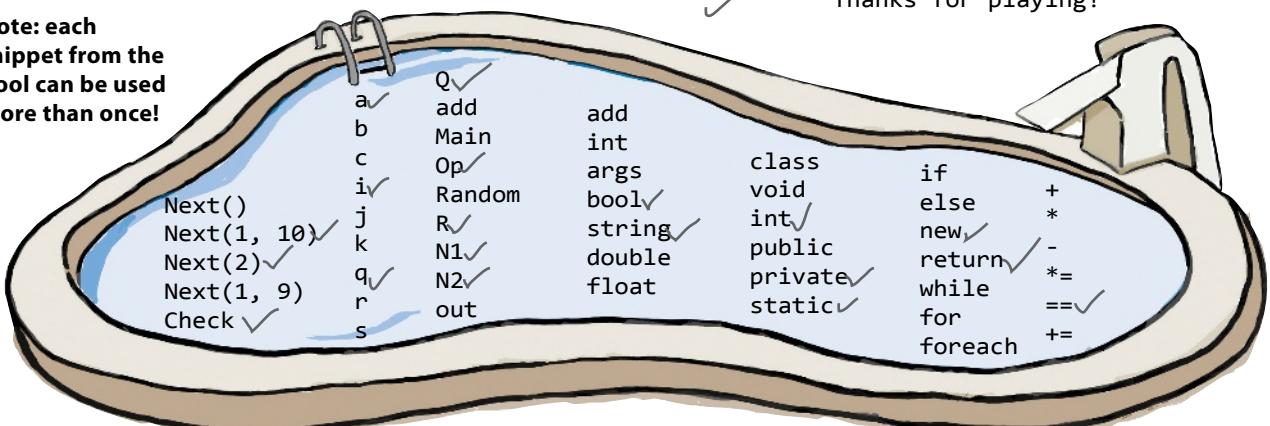
8 + 6 = Bye

Thanks for playing!

The game generates random addition or multiplication questions.

If you get a question wrong, it keeps asking until you get it right.

The game ends when you enter an answer that isn't a number.



A few useful facts about methods and properties

- ★ **Every method in your class has a unique signature.**

The first line of a method, which contains the access modifier, return value, name, and parameters is called the method's **signature**. Properties have signatures, too—they consist of the access modifier, type, and name.

- ★ **You can initialize properties in an object initializer.**

You used object initializers before:

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
```

You can also specify properties in an object initializer. If you do, the constructor is run first, then the properties are set. And you can only initialize public fields and properties in the object initializer.

- ★ **Every class has a constructor, even if you didn't add one yourself.**

The CLR needs a constructor to instantiate an object—it's part of the behind-the-scenes mechanics of how .NET works. So if you don't add a constructor to your class, the C# compiler automatically adds a parameterless constructor for you.

- ★ **You can keep a class from being instantiated by other classes by adding a private constructor.**

Sometimes you need to have really careful control over how your objects are created. One way to do that is to make your constructor private—then it can only be called from inside the class. Take a minute and try it out:

```
class NoNew {
    private NoNew() { Console.WriteLine("I'm alive!"); }
    public static NoNew CreateInstance() { return new NoNew(); }
}
```

Add that `NoNew` class to a console app. If you try to add `new NoNew();` to your `Main` method, the C# compiler gives you an error (*'NoNew.NoNew()' is inaccessible due to its protection level*), but the **NoNew.CreateInstance** method creates a new instance just fine.

This is a really good time to talk about aesthetics in video games. If you think about it, encapsulation doesn't really give you a way to do anything that you couldn't before. You could still write the same programs without properties, constructors, and private methods—but they would sure look really different. That's because not everything in programming is about making your code do something different. Often, it's about making your code do the same thing but in a better way. Think about that when you read about aesthetics. They don't change the way your game behaves; they change the way the player thinks and feels about the game.



Aesthetics

Game design... and beyond

How did you feel the last time you played a game? Was it fun? Did you feel a thrill, a rush of adrenaline? Did it give you a sense of discovery or accomplishment? Did you get a feeling of competition or cooperation with other players? Was there an engaging story? Was it funny? Sad? Games bring out emotional responses in us, and that's the idea behind aesthetics.

Does it seem weird to talk about feelings and video games? It shouldn't—emotions and feelings have always played an important part in game design, and the most successful games have an important aesthetic aspect to them. Think about that satisfying feeling you get when you drop a long piece in Tetris and it clears four rows of blocks. Or that rush in Pac-Man when Blinky (the red ghost) is just pixels behind you when you swallow the power pellet.

- It's obvious how **art and visuals, music and sound**, or story writing can influence aesthetics, but aesthetics is more than the artistic elements of a game. Aesthetics can come from the way the game is **structured**.
- And it's not just video games—you can find **aesthetics in tabletop games**. Poker is known for its emotional highs and lows, the feeling of pulling off a great bluff. Even a simple card game like Go Fish! has its own aesthetics: the growing back and forth, as players figure out each other's hands; the crescendo toward a winner, as players put each new book on the table; the thrill of drawing that card you need; just saying "Go fish!" when asked for the wrong card.
- Sometimes we talk about "**fun**" and "**gameplay**," but it helps to get more precise when we're talking about aesthetics.
- When a game provides **challenge** it gives players obstacles to get past, creating a feeling of accomplishment and personal victory.
- A game's **narrative** draws the player into the drama of a story.
- The pure **tactile sensation** of a game—the beat of a rhythm game, the satisfying "gulp" of eating a power pellet, the "vroom" and blur of an accelerating car—provides pleasure.
- Playing a cooperative or multiplayer game brings a sense of **fellowship** with others.
- A game that provides **fantasy** not only transports a player to another world, but lets that player be another person (or non-person!) entirely.
- Games with **expression** give the player self-discovery, a way to learn more about themselves.

Believe it or not, we can use these ideas behind aesthetics to learn a **larger lesson about development** that applies to any kind of program or app, not just a game. Let these ideas sink in for now—we'll return to this in the next chapter.



Some developers are really skeptical when they read about aesthetics because they assume that only the mechanics of the game matter. Here's a quick thought experiment to show how important aesthetics can be. Say you have two games with identical mechanics. There's just one very tiny difference between them. In one game you're kicking boulders out of the way to save a village. In the other game, you're kicking puppies and kittens because you are a horrible person. Even if every other aspect of those games is identical, those are two very different games. That's the power of aesthetics.



This code has problems. It's supposed to be code for a simple gumball vending machine: you put in a coin and it dispenses gum. We've identified four specific problems that will cause bugs. Use the lines provided to write down what you think is wrong with the lines the arrows are pointing to.

```
class GumballMachine {
    private int gumballs;
    .....  

    .....  

    private int price;
    public int Price
    {
        get
        {
            return price;
        }
    }
    .....  

    .....  

    public GumballMachine(int gumballs, int price)
    {
        gumballs = this.gumballs;   

        price = Price;   

    }
    .....  

    .....  

    public string DispenseOneGumball(
        int price, int coinsInserted)
    {
        // check the price backing field
        if (this.coinsInserted >= price) {
            gumballs -= 1;
            return "Here's your gumball";
        } else {
            return "Insert more coins";
        }
    }
}
```



Sharpen your pencil Solution

This code has problems. We pointed out four specific lines that will cause bugs. Here's what's wrong with them.

Lowercase-p price refers to the constructor parameter, not the field. This line sets the PARAMETER to the value returned by the Price getter, but Price hasn't even been set yet, so it doesn't do anything useful. If you flip this around to set Price = price, it will work.

```
public GumballMachine(int gumballs, int price)
{
    gumballs = this.gumballs; ←
    price = Price;           →
}
```

The "this" keyword is on the wrong "gumballs." this.gumballs refers to the property, while gumballs refers to the parameter.

This parameter masks the private field called price, and the comment says the method is supposed to be checking the value of the price backing field.

```
public string DispenseOneGumball(int price, int coinsInserted)
```

The "this" keyword is on a parameter, where it doesn't belong. It should be on price, because that field is masked by a parameter. }

```
// check the price backing field
if (this.coinsInserted >= price) {
    gumballs -= 1;
    return "Here's your gumball";
} else {
    return "Insert more coins";
}
```

It's worth your time to take an extra few minutes to **really look at this code**. These are common mistakes that new programmers make when working with objects. If you learn to avoid them, you'll find it much more satisfying to write code.

^{there are no} Dumb Questions }

Q: If my constructor is a method, why doesn't it have a return type?

A: Your constructor doesn't have a return type because **every** constructor is always void—which makes sense, because there's no way for it to return a value. It would be redundant to make you type void at the beginning of each constructor.

Q: Can I have a getter without a setter?

A: Yes! When you have a get accessor but no set, you create a read-only property. For example, the SecretAgent class might have a public read-only field with a backing field for the name:

```
string spyNumber = "007";
public string SpyNumber {
    get { return spyNumber; }
```

Q: And I bet I can have a setter without a getter, right?

A: Yes—unless it's an auto-property, in which case you'll get an error ("Auto-implemented properties must have get accessors"). If you create a property with a setter but no getter, then your property can only be written. The SecretAgent class could use that for a property that other spies could write to, but not see:

```
public string DeadDrop {
    set {
        StoreSecret(value);
    }
}
```

Both of those techniques—set without get, or vice versa—can come in really handy when you're doing encapsulation.



Go to the Visual Studio for Mac Learner's Guide for the Mac version of this exercise.

Use what you've learned about encapsulation to fix Owen's sword damage calculator. First, modify the SwordDamage class to replace the fields with properties and add a constructor. Once that's done, update the console app to use it. Finally, fix the WPF app. (This exercise will go more easily if you create a new console app for the first two parts and a new WPF app for the third.)

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

1. Delete the Roll field and replace it with a property named Roll and a backing field named roll. The getter returns the value of the backing field. The setter updates the backing field, then calls the CalculateDamage method.
2. Delete the SetFlaming method and replace it with a property named Flaming and a backing field named flaming. It works like the Roll property—the getter returns the backing field, the setter updates it and calls CalculateDamage.
3. Delete the SetMagic method and replace it with a property named Magic and a backing field named magic that works exactly like the Flaming and Roll properties.
4. Create an auto-implemented property named Damage with a public get accessor and private set accessor.
5. Delete the MagicMultiplier and FlamingDamage fields. Modify the CalculateDamage method so it checks the property values for the Roll, Magic, and Flaming properties and does the entire calculation inside the method.
6. Add a constructor that takes the initial roll as its parameter. Now that the CalculateDamage method is only called from the property set accessors and constructor, there's no need for another class to call it. Make it private.
7. Add XML code documentation to all of the public class members.

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

1. Create a static method called RollDice that returns the results of a 3d6 roll. You'll need to store the Random instance in a static field instead of a variable so both the Main method and RollDice can use it.
2. Use the new RollDice method for the SwordDamage constructor argument and to set the Roll property.
3. Change the code that calls SetMagic and SetFlaming to set the Magic and Flaming properties instead.

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

1. Copy the code from Part 1 into a new WPF app. Copy the XAML from the project earlier in the chapter.
2. In the code-behind, declare the MainWindow.swordDamage field like this (and instantiate it in the constructor):

```
SwordDamage swordDamage;
```
3. In the MainWindow constructor, set the swordDamage field to a new instance of SwordDamage initialized with a random 3d6 roll. Then call the CalculateDamage method.
4. The RollDice and Button_Click methods are exactly the same as earlier in the chapter.
5. Change the DisplayDamage method to use string interpolation, but it should still display the same string as before.
6. Change the Checked and Unchecked event handlers for both checkboxes to use the Magic and Flaming properties instead of the old SetMagic and SetFlaming methods, then call DisplayDamage.

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



Exercise Solution

Now Owen finally has a class for calculating damage that's much easier to use without running into bugs. Each property recalculates the damage, so it doesn't matter what order you call them in. Here's the code for the well-encapsulated `SwordDamage` class:

```
class SwordDamage
{
    private const int BASE_DAMAGE = 3;
    private const int FLAME_DAMAGE = 2;

    /// <summary>
    /// Contains the calculated damage.
    /// </summary>
    public int Damage { get; private set; }

    private int roll;

    /// <summary>
    /// Sets or gets the 3d6 roll.
    /// </summary>
    public int Roll
    {
        get { return roll; }
        set
        {
            roll = value;
            CalculateDamage();
        }
    }

    private bool magic;

    /// <summary>
    /// True if the sword is magic, false otherwise.
    /// </summary>
    public bool Magic
    {
        get { return magic; }
        set
        {
            magic = value;
            CalculateDamage();
        }
    }

    private bool flaming;

    /// <summary>
    /// True if the sword is flaming, false otherwise.
    /// </summary>
    public bool Flaming
    {
        get { return flaming; }
        set
        {
            flaming = value;
            CalculateDamage();
        }
    }
}
```

Since these constants aren't going to be used by any other class, it makes sense to keep them private.

The `Damage` property's private set accessor makes it read-only, so it can't be overwritten by another class.

Here's the `Roll` property with its private backing field. The set accessor calls the `CalculateDamage` method, which keeps the `Damage` property updated automatically.

The `Magic` and `Flaming` properties work just like the `Roll` property. They all call `CalculateDamage`, so setting any of them automatically updates the `Damage` property.



```

/// <summary>
/// Calculates the damage based on the current properties.
/// </summary>
private void CalculateDamage()
{
    decimal magicMultiplier = 1M;
    if (Magic) magicMultiplier = 1.75M;
    Damage = BASE_DAMAGE;
    Damage = (int)(Roll * magicMultiplier) + BASE_DAMAGE;
    if (Flaming) Damage += FLAME_DAMAGE;
}

/// <summary>
/// The constructor calculates damage based on default Magic
/// and Flaming values and a starting 3d6 roll.
/// </summary>
/// <param name="startingRoll">Starting 3d6 roll</param>
public SwordDamage(int startingRoll)
{
    roll = startingRoll;
    CalculateDamage();
}

```

All of the calculation is encapsulated inside the CalculateDamage method. It only depends on the get accessors for the Roll, Magic, and Flaming properties.

The constructor sets the backing field for the Roll property, then calls CalculateDamage to make sure the Damage property is correct.

Here's the code for the Main method of the console app:

```

class Program
{
    static Random random = new Random();

    static void Main(string[] args)
    {
        SwordDamage swordDamage = new SwordDamage(RollDice());
        while (true)
        {
            Console.Write("0 for no magic/flaming, 1 for magic, 2 for flaming, " +
                "3 for both, anything else to quit: ");
            char key = Console.ReadKey().KeyChar;
            if (key != '0' && key != '1' && key != '2' && key != '3') return;
            swordDamage.Roll = RollDice();
            swordDamage.Magic = (key == '1' || key == '3');
            swordDamage.Flaming = (key == '2' || key == '3');
            Console.WriteLine($"\\nRolled {swordDamage.Roll} for {swordDamage.Damage} HP\\n");
        }
    }

    private static int RollDice()
    {
        return random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
    }
}

```

It made sense to move the 3db roll into its own method since it's called from two different places in Main. If you used "Generate method" to create it, the IDE made it private automatically.



Exercise Solution

Here's the code for the code-behind for the WPF desktop app. The XAML is exactly the same.

We didn't ask you to move the 3d6 roll into its own method. Do you think adding a RollDice method (like in the console app) would make this code easier to read? Or is it unnecessary? One way is not necessarily better or worse than the other! Try it both ways and decide what works best for you.

```
public partial class MainWindow : Window
{
    Random random = new Random();
    SwordDamage swordDamage;

    public MainWindow()
    {
        InitializeComponent();
        swordDamage = new SwordDamage(random.Next(1, 7) + random.Next(1, 7)
                                      + random.Next(1, 7));
        DisplayDamage();
    }

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

    void DisplayDamage()
    {
        damage.Text = $"Rolled {swordDamage.Roll} for {swordDamage.Damage} HP";
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        RollDice();
    }

    private void Flaming_Checked(object sender, RoutedEventArgs e)
    {
        swordDamage.Flaming = true;
        DisplayDamage();
    }

    private void Flaming_Unchecked(object sender, RoutedEventArgs e)
    {
        swordDamage.Flaming = false;
        DisplayDamage();
    }

    private void Magic_Checked(object sender, RoutedEventArgs e)
    {
        swordDamage.Magic = true;
        DisplayDamage();
    }

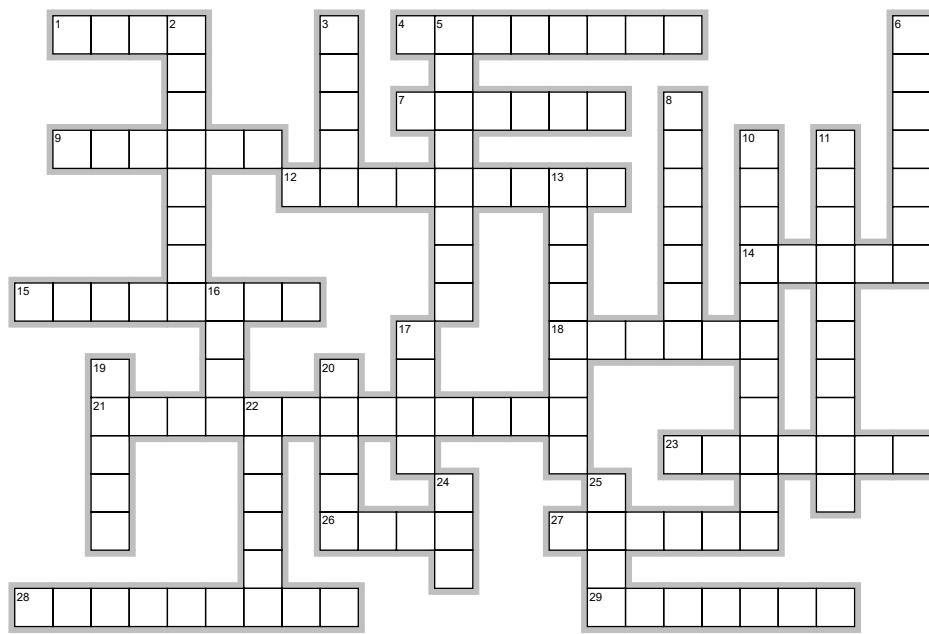
    private void Magic_Unchecked(object sender, RoutedEventArgs e)
    {
        swordDamage.Magic = false;
        DisplayDamage();
    }
}
```

Deciding whether or not to move a single line of duplicated code into its own method is a good example of code aesthetics. Beauty is in the eye of the beholder.

Take a break, sit back, and give your right brain something to do. It's your standard crossword; all of the solution words are from the first five chapters of the book.



Objectcross



EclipseCrossword.com

Across

1. What (int) is doing in this line of code: `x = (int) y;`
4. Looks like a field but acts like a method
7. What kind of sequence is or ?
9. If you want to create instances of a class, don't put this keyword in the declaration
12. A variable that points to an object
14. What an object is an instance of
15. The four whole-number types that only hold positive numbers
18. You can assign any value to a variable of this type
21. What you're doing when you use \$ and curly braces to include values in a string
23. Draw one of these for your class before you start writing code
26. How you start a variable declaration
27. The numeric type that holds the biggest numbers
28. What you use to pass information into a method
29. If you want to store a currency value, use this type

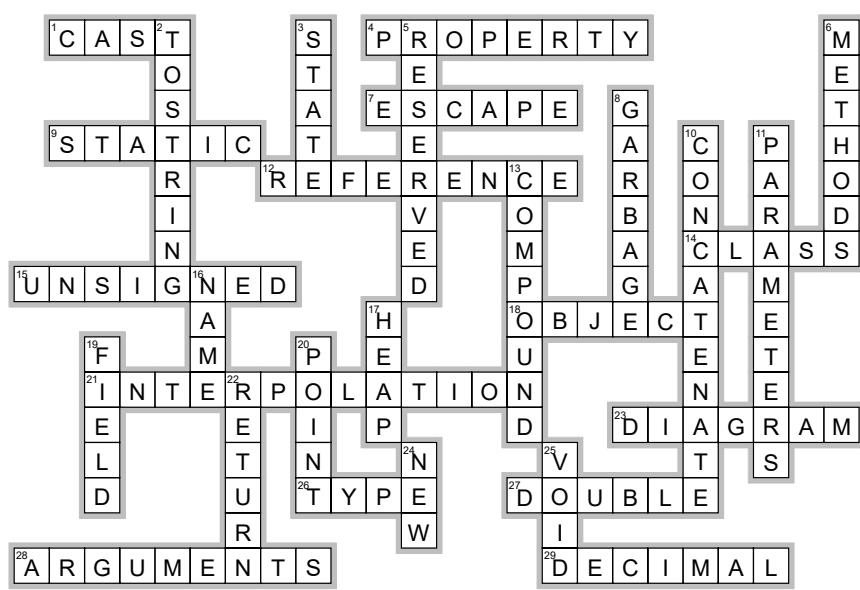
Down

2. Every object has this method that converts it to a string
3. An object's fields keep track of its _____
5. namespace, for, while, using, and new are examples of _____ keywords
6. These define the behavior of a class
8. The kind of collection that happens when the last reference to an object goes away
10. What you're doing when you use the + operator to stick two strings together
11. How a method tells you what to pass to it
13. += and -= are _____ assignment operators
16. The second part of a variable declaration
17. Where objects live
19. A variable declared directly in a class that all its members can access
20. What floats in a float
22. Tells a method to stop immediately, possibly sending a value back to the statement that called it
24. The statement you use to create an object
25. If a method's return type is _____, it doesn't return anything

BULLET POINTS

- **Encapsulation** keeps your code safe by preventing classes from modifying it unexpectedly or otherwise misusing other classes' members.
- Fields that require some processing or calculation to happen when they're set are **prime candidates** for encapsulation.
- Think about ways fields and methods can be **misused**. Only make fields and methods public if you need to.
- Using consistent case when choosing names for fields, properties, variables, and methods makes code easier to read. Many developers use **camelCase** for private fields and **PascalCase** for public ones.
- A **property** is a class member that looks like a field when it's used, but acts like a method when it runs.
- A **get accessor** (or **getter**) is defined by the `get` keyword followed by a method that returns the value of the property.
- A **set accessor** (or **setter**) is defined by the `set` keyword and followed by a method that sets the value of the property. Inside the method, the `value` keyword is a read-only variable that contains the value being set.
- Properties often get or set a **backing field**, or a private field that's encapsulated by restricting access to it through a property.
- An **auto-implemented property**—sometimes called an **automatic property** or **auto-property**—is a property that has a getter that returns the value of the backing field, and a setter that updates it.
- Use the **prop snippet** in Visual Studio to create an auto-implemented property by typing “`prop`” followed by two tabs.
- Use the **private keyword** to restrict access to a get or set accessor. A read-only property has a private set accessor.
- When an object is created, the CLR first **sets** all of the fields that have values set in their declarations and then **executes** the constructor, before **returning** to the `new` statement that created the object.
- Use a **constructor with parameters** to initialize properties. Specify arguments to pass to the constructor when you use the `new` keyword.
- A parameter with the same name as a field **masks** that field. Use the **this keyword** to access the field.
- If you don't add a constructor to your class, the C# compiler automatically adds a **parameterless constructor** for you.
- You can keep a class from being instantiated by other classes by adding a **private constructor**.

Objectcross
solution



EclipseCrossword.com

6 inheritance

Your object's family tree

SO THERE I WAS RIDING MY **BICYCLE** OBJECT DOWN DEAD MAN'S CURVE WHEN I REALIZED IT INHERITED FROM **TWOWHEELER** AND I FORGOT TO OVERRIDE THE **BRAKES** METHOD...LONG STORY SHORT, TWENTY-SIX STITCHES AND MOM SAYS I'M GROUNDED FOR A MONTH.



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 can **avoid duplicate code**, **model the real world** more closely, and end up with apps that are **easier to maintain** and **less prone to bugs**.

Calculate damage for MORE weapons

Do this!

The updated sword damage calculator was huge a hit on game night! Now Owen wants calculators for all of the weapons. Let's start with the damage calculation for an arrow, which uses a 1d6 roll. Let's **create a new ArrowDamage class** to calculate the arrow damage using the arrow formula in Owen's game master notebook.

Most of the code in ArrowDamage will be **identical to the code** in the SwordDamage class. Here's what we need to do to get started building the new app.

- ➊ **Create a new .NET Console App project.** We want it to do both sword and arrow calculations, so **add the SwordDamage class** to the project.
- ➋ **Create an ArrowDamage class that's an exact copy of SwordDamage.** Create a new class called ArrowDamage, then **copy all of the code from SwordDamage and paste it** into the new ArrowDamage class. Then change the constructor name to ArrowDamage so the program builds.
- ➌ **Refactor the constants.** The arrow damage formula has different values for the base and flame damage, so let's rename the BASE_DAMAGE constant to BASE_MULTIPLIER and update the constant values. We think these constants make the code easier to read, so add a MAGIC_MULTIPLIER constant too:

```
private const decimal BASE_MULTIPLIER = 0.35M;
private const decimal MAGIC_MULTIPLIER = 2.5M;
private const decimal FLAME_DAMAGE = 1.25M;
```

} Do you agree with us that these constants make the code easier to read? It's OK if you don't!

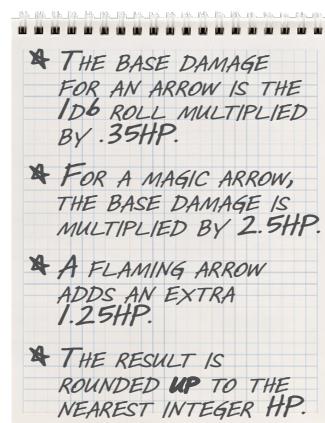
- ➍ **Modify the CalculateDamage method.** Now all you need to do to make your new ArrowDamage class work is to update the CalculateDamage method so it does the correct calculation:

```
private void CalculateDamage()
{
    decimal baseDamage = Roll * BASE_MULTIPLIER;
    if (Magic) baseDamage *= MAGIC_MULTIPLIER;
    if (Flaming) Damage = (int)Math.Ceiling(baseDamage + FLAME_DAMAGE);
    else Damage = (int) Math.Ceiling(baseDamage);
}
```

You can use the `Math.Ceiling` method to round values up. It keeps the type, so you still need to cast to an int.



There are **many** different ways to write code that does the same thing. Can you think of another way to write the code to calculate arrow damage?



ArrowDamage
Roll
Magic
Flaming
Damage

Use a switch statement to match several candidates

Let's update our console app to prompt the user whether to calculate damage from an arrow or a sword. We'll ask for a key, and use the static **Char.ToUpper method** to convert it to uppercase:

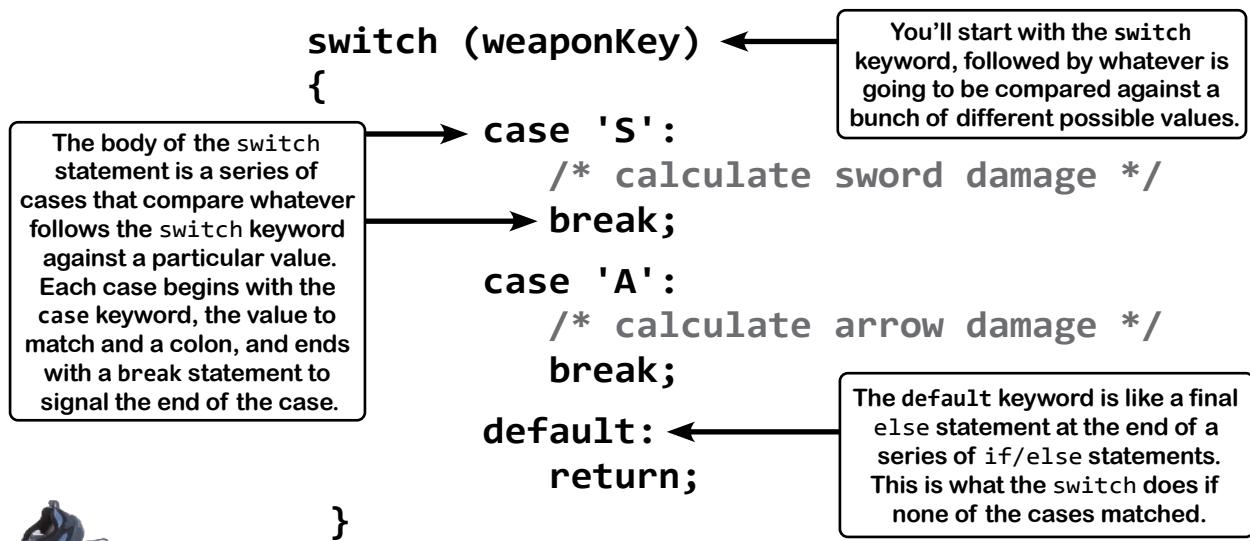
```
Console.Write("\nS for sword, A for arrow, anything else to quit: ");
weaponKey = Char.ToUpper(Console.ReadKey().KeyChar);
```

We **could** use **if/else** statements for this:

```
if (weaponKey == 'S') { /* calculate sword damage */ }
else if (weaponKey == 'A') { /* calculate arrow damage */ }
else return;
```

The **Char.ToUpper** method converts 's' and 'a' to 'S' and 'A'

That's how we've handled input so far. Comparing one variable against many different values is a really common pattern that you'll see over and over again. It's so common that C# has a special kind of statement *designed specifically for this situation*. A **switch statement** lets you compare one variable against many values in a way that's compact and easy to read. Here's a **switch statement** that does exactly the same thing as the **if/else** statements above:



Exercise

Update the Main method to use a **switch statement** to let the user choose the type of weapon. Start by copying the Main and RollDice methods from the exercise solution at the end of the last chapter.

1. Create an instance of ArrowDamage at the top of the method, just after you create the SwordDamage instance.
2. Modify the RollDice method to take an int parameter called `numberOfRolls` so you can call `RollDice(3)` to roll 3d6 (so it calls `random.Next(1, 7)` three times and adds the results), or `RollDice(1)` to roll 1d6.
3. Add the two lines of code exactly like they appear above that write the sword or arrow prompt to the console, read the input using `Console.ReadKey()`, use `Char.ToUpper` to convert the key to uppercase, and store it in `weaponKey`.
4. **Add the switch statement.** It will be exactly the same as the switch statement above, except you'll replace each of the `/* comments */` with code that calculates damage and writes a line of output to the console.



Exercise Solution

We just gave you a totally new piece of C# syntax—the **switch statement**—and asked you to use it in a program. The C# team at Microsoft is constantly improving the language, and being able to incorporate new language elements into your code is a **really valuable C# skill**.

```

class Program
{
    static Random random = new Random();

    static void Main(string[] args)
    {
        SwordDamage swordDamage = new SwordDamage(RollDice(3));
        ArrowDamage arrowDamage = new ArrowDamage(RollDice(1)); ← Create an instance of the new ArrowDamage class that you created.

        while (true)
        {
            Console.Write("0 for no magic/flaming, 1 for magic, 2 for flaming, " +
                "3 for both, anything else to quit: ");
            char key = Console.ReadKey().KeyChar;
            if (key != '0' && key != '1' && key != '2' && key != '3') return;

            Console.WriteLine("\nS for sword, A for arrow, anything else to quit: ");
            char weaponKey = Char.ToUpper(Console.ReadKey().KeyChar);

            switch (weaponKey)
            {
                case 'S':
                    swordDamage.Roll = RollDice(3);
                    swordDamage.Magic = (key == '1' || key == '3');
                    swordDamage.Flaming = (key == '2' || key == '3');
                    Console.WriteLine(
                        $"{\nRolled {swordDamage.Roll} for {swordDamage.Damage} HP\n}");
                    break;

                case 'A':
                    arrowDamage.Roll = RollDice(1);
                    arrowDamage.Magic = (key == '1' || key == '3');
                    arrowDamage.Flaming = (key == '2' || key == '3');
                    Console.WriteLine(
                        $"{\nRolled {arrowDamage.Roll} for {arrowDamage.Damage} HP\n}");
                    break;

                default:
                    return;
            }
        }
    }

    private static int RollDice(int numberofRolls)
    {
        int total = 0;
        for (int i = 0; i < numberofRolls; i++) total += random.Next(1, 7);
        return total;
    }
}

```

This block of code is almost identical to the program from the last chapter. Instead of using it in an if/else block, it's in a case in a switch statement (and it passes an argument to RollDice).

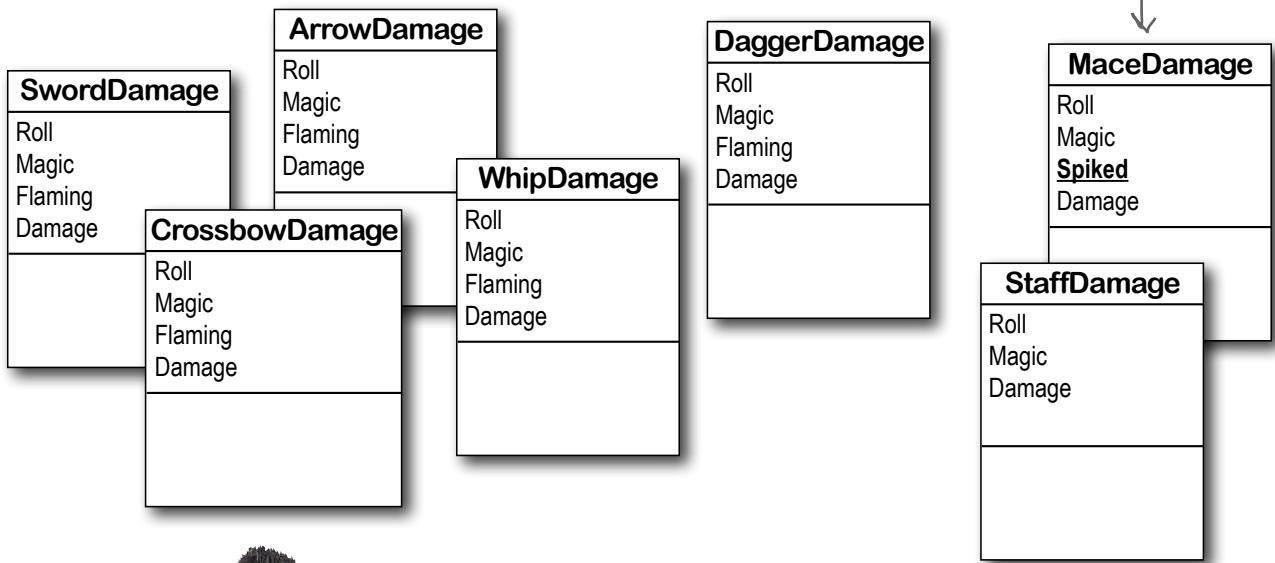
The code to use the instance of ArrowDamage to calculate damage is really similar to the code for SwordDamage. In fact, it's almost identical. Is there a way that we can reduce the duplication of code and make our program easier to read?

Try this! Set a breakpoint on `switch (weaponKey)`, then use the debugger to step through the switch statement. That's a great way to really get a sense of how it works. Then try removing one of the break lines and stepping through it—the execution continues (or falls through) to the next case.

One more thing...can we calculate damage for a dagger? and a mace? and a staff? and...

We've made two classes for sword and arrow damage. But what happens if there are three other weapons? Or four? Or 12? And what if you had to maintain that code and make more changes later? What if you had to make the ***same exact change*** to five or six ***closely related*** classes? What if you had to keep making changes? It's inevitable that bugs would slip through—it's way too easy to update five classes but forget to change the sixth.

What if some of the classes are related, but not quite identical? What if a mace can be spiked or not, but it can't be flaming? Or if a staff can't be either of those things?



WOW, I'D HAVE TO WRITE THE SAME CODE OVER AND OVER AGAIN. THAT'S A REALLY INEFFICIENT WAY TO WORK. THERE'S GOT TO BE A BETTER WAY.

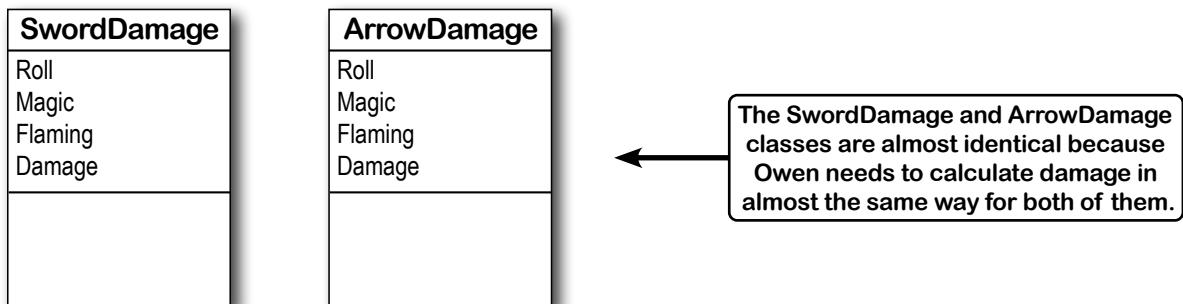
You're right! Having the same code repeated in different classes is inefficient and error-prone.

Lucky for us, C# gives us a better way to build classes that are related to each other and share behavior: **inheritance**.

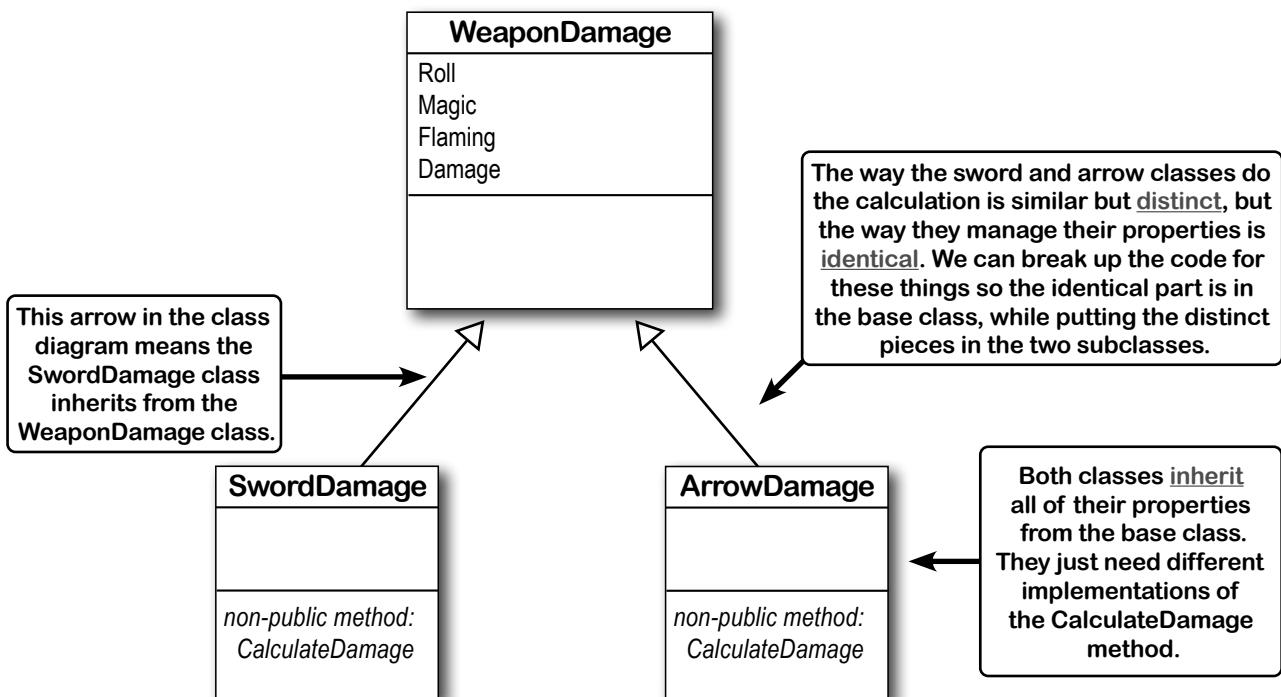
no need to use gold when anything shiny will do

When your classes use inheritance, you only need to write your code once

It's no coincidence that your SwordDamage and ArrowDamage classes have a lot of the same code. When you write C# programs, you often create classes that represent things in the real world, and those things are usually related to each other. Your classes have **similar code** because the things they represent in the real world—two similar calculations from the same role-playing game—have **similar behaviors**.

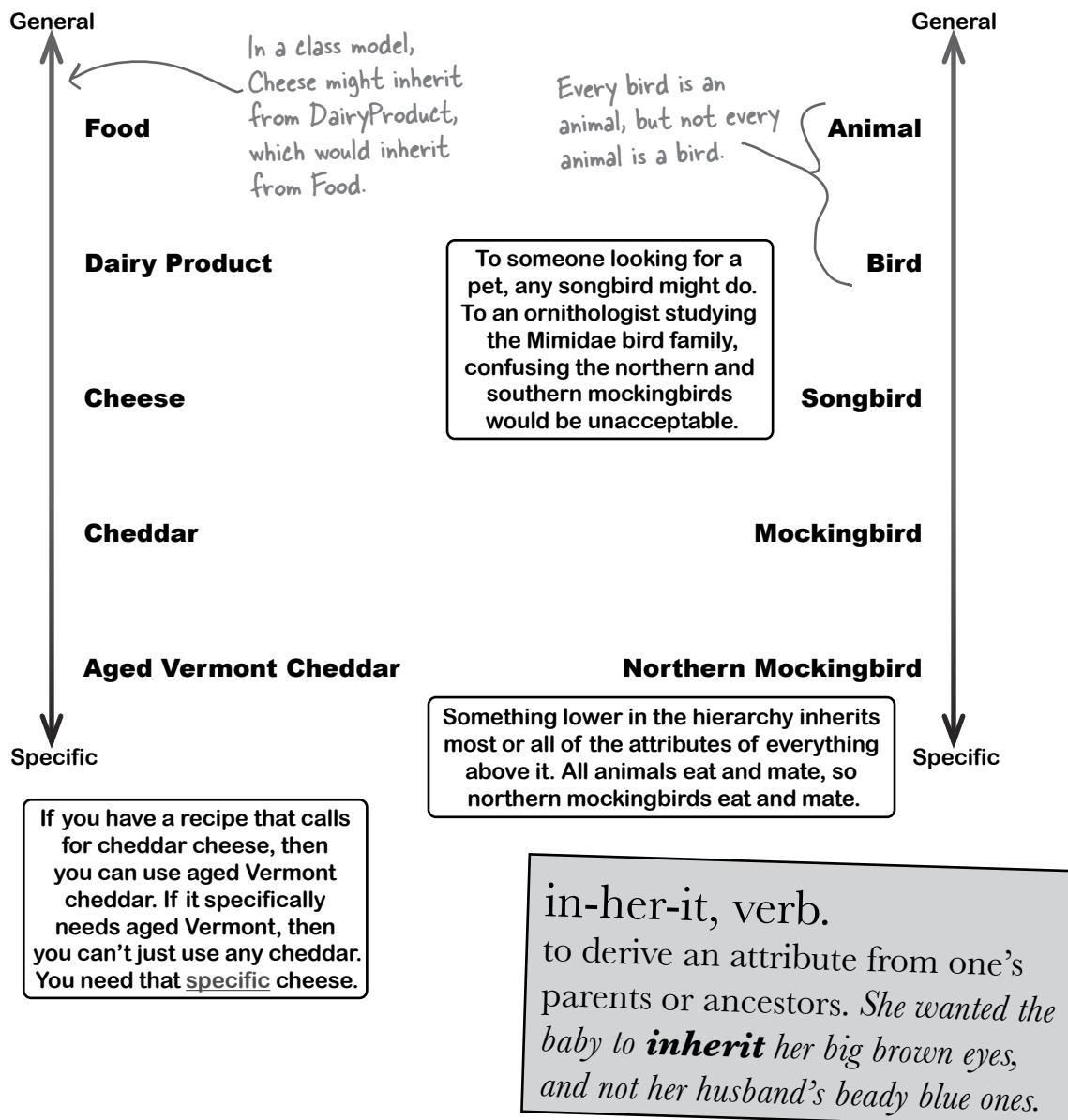


When you have two classes that are specific cases of something more general, you can set them up to **inherit** from the same class. When you do that, each of them is a **subclass** of the same **base class**.



Build up your class model by starting general and getting more specific

When you build a set of classes that represent things (especially things in the real world), you're building a **class model**. Real-world things are often in a **hierarchy** that goes from more general to more specific, and your programs have their own **class hierarchy** that does the same thing. In your class model, classes further down in the hierarchy **inherit** from those above them.



How would you design a zoo simulator?

Lions and tigers and bears...oh my! Also, hippos, wolves, and the occasional dog. Your job is to design an app that simulates a zoo. (Don't get too excited—we're not going to actually build the code, just design the classes to represent the animals. We bet you're already thinking about how you'd do this in Unity!)

We've been given a list of some of the animals that will be in the program, but not all of them. We know that each animal will be represented by an object, and that the objects will move around in the simulator, doing whatever it is that each particular animal is programmed to do.

More importantly, we want the program to be easy for other programmers to maintain, which means they'll need to be able to add their own classes later on if they want to add new animals to the simulator.

Let's start by building a class model for the animals we know about.

So what's the first step? Well, before we can talk about **specific** animals, we need to figure out the **general** things they have in common—the abstract characteristics that **all** animals have. Then we can build those characteristics into a base class that all animal classes can inherit from.

The terms **parent**, **superclass**, and **base class** are often used interchangeably. Also, the terms **extend** and **inherit from** mean the same thing. The terms **child** and **subclass** are also synonymous, but **subclass** can also be used as a verb.



Some people use the term "base class" to specifically mean the class at the top of the inheritance tree...but not the **VERY** top, because every class inherits from Object or a subclass of Object.

1

Look for things the animals have in common.

Take a look at these six animals. What do a lion, a hippo, a tiger, a bobcat, a wolf, and a dog have in common? How are they related? You'll need to figure out their relationships so you can come up with a class model that includes all of them.



The zoo simulator includes a guard dog that roams the grounds protecting the animals.

2 Build a base class to give the animals everything they have in common.

The fields, properties, and methods in the base class will give all of the animals that inherit from it a common state and behavior. They're all animals, so it makes sense to call the base class Animal.

You already know that we should avoid duplicate code: it's hard to maintain, and always leads to headaches down the road. So let's choose fields and methods for an Animal base class that you **only have to write once**, and each of the animal subclasses can inherit them. Let's start with the public properties:

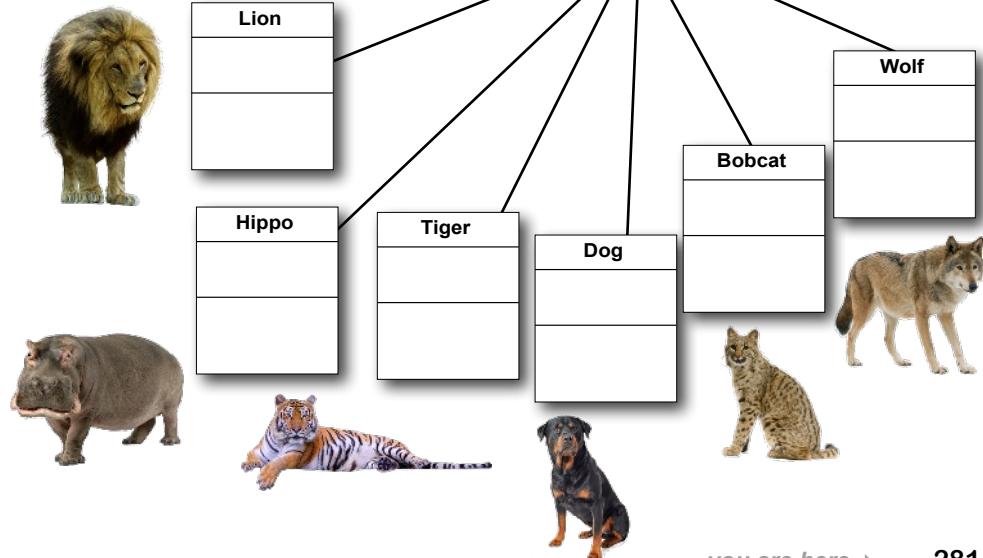
- ★ Picture: a path to an image file.
- ★ Food: the type of food this animal eats. Right now, there can be only two values: meat and grass.
- ★ Hunger: an int representing the hunger level of the animal. It changes depending on when (and how much) the animal eats.
- ★ Boundaries: a reference to a class that stores the height, width, and location of the pen that the animal will roam around in.
- ★ Location: the X and Y coordinates where the animal is standing.

In addition, the Animal class has four methods the animals can inherit:

- ★ MakeNoise: a method to let the animal make a sound.
- ★ Eat: behavior for when the animal encounters its preferred food.
- ★ Sleep: a method to make the animal lie down and take a nap.
- ★ Roam: a method to make animals wander around their pens.

Animal
Picture Food Hunger Boundaries Location
MakeNoise Eat Sleep Roam

Choosing a base class is about making choices. You could have decided to use a ZooOccupant class that defines the feed and maintenance costs, or an Attraction class with methods for how the animals entertain the zoo visitors. We think Animal makes the most sense here. What do you think?



Different animals have different behaviors

Lions roar, dogs bark, and as far as *we* know hippos don't make any sound at all. All of the classes that inherit from Animal will have a MakeNoise method, but each of those methods will work a different way and will have different code. When a subclass changes the behavior of one of the methods that it inherited, we say that it **overrides** the method.

Just because a property or a method is in the Animal base class, that doesn't mean every subclass has to use it the same way...or at all!

➊ Figure out what each animal does that the Animal class does differently—or not at all.

Every animal needs to eat, but a dog might take small bites of meat while a hippo eats huge mouthfuls of grass. What would the code for that behavior look like? Both the dog and the hippo would override the Eat method. The hippo's method would have it consume, say, 20 pounds of hay each time it was called. The dog's Eat method, on the other hand, would reduce the zoo's food supply by one 12-ounce can of dog food.

So when you've got a subclass that inherits from a base class, it **must** inherit all of the base class's behaviors... but you **can** modify them in the subclass so they're not performed exactly the same way. That's what overriding is all about.



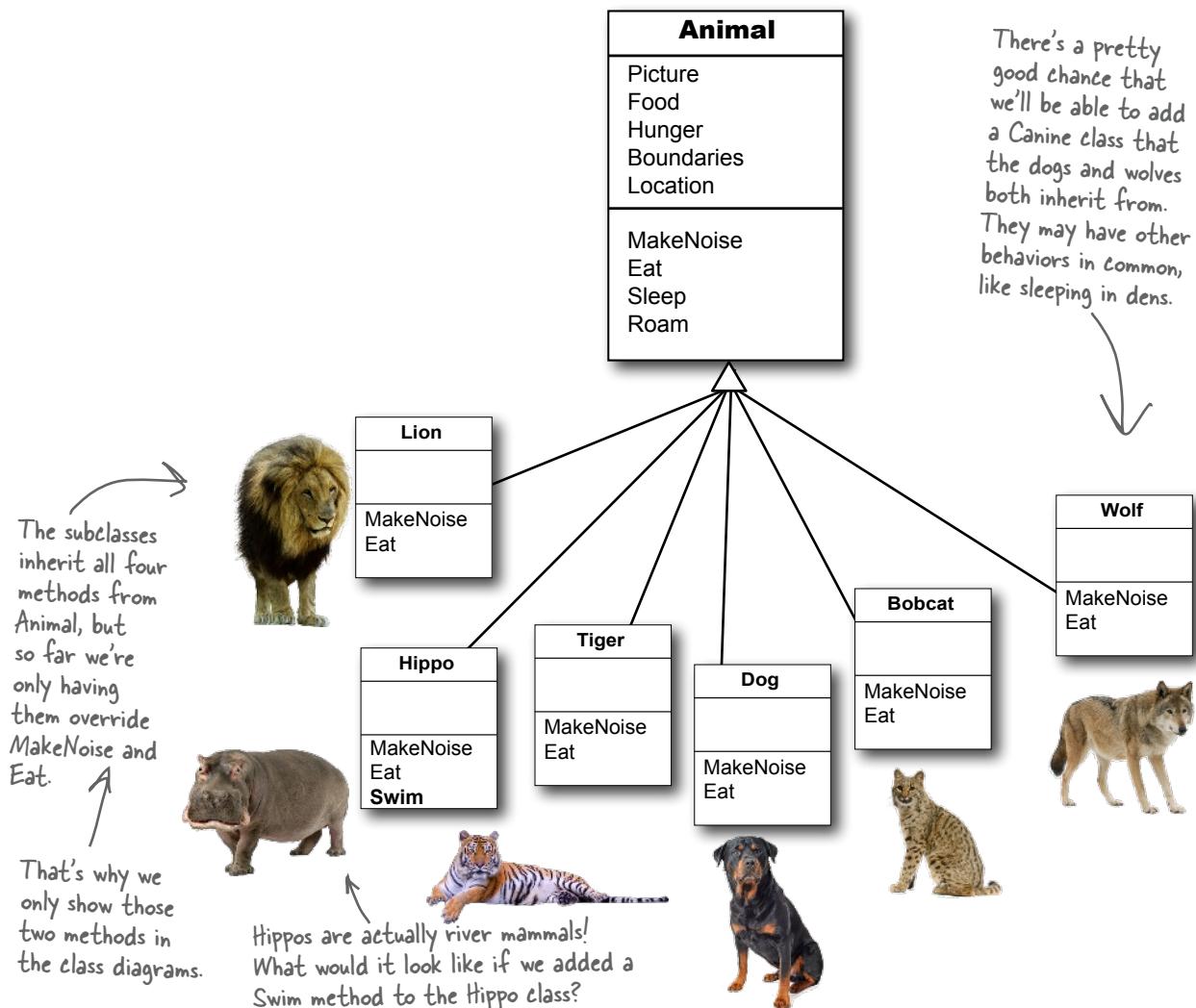
Animal
Picture
Food
Hunger
Boundaries
Location
MakeNoise
Eat
Sleep
Roam



We already know that some animals will override the MakeNoise and Eat methods. Which animals will override Sleep or Roam? Will any of them?

4 Look for classes that have a lot in common.

Don't dogs and wolves seem pretty similar? They're both canines, and it's a good bet that if you look at their behavior they have a lot in common. They probably eat the same food and sleep the same way. What about bobcats, tigers, and lions? It turns out all three of them move around their habitats in exactly the same way. It's a good bet that you'll be able to have a general Feline class that lives between Animal and those three feline classes that can help prevent duplicate code between them.



5

Finish your class hierarchy.

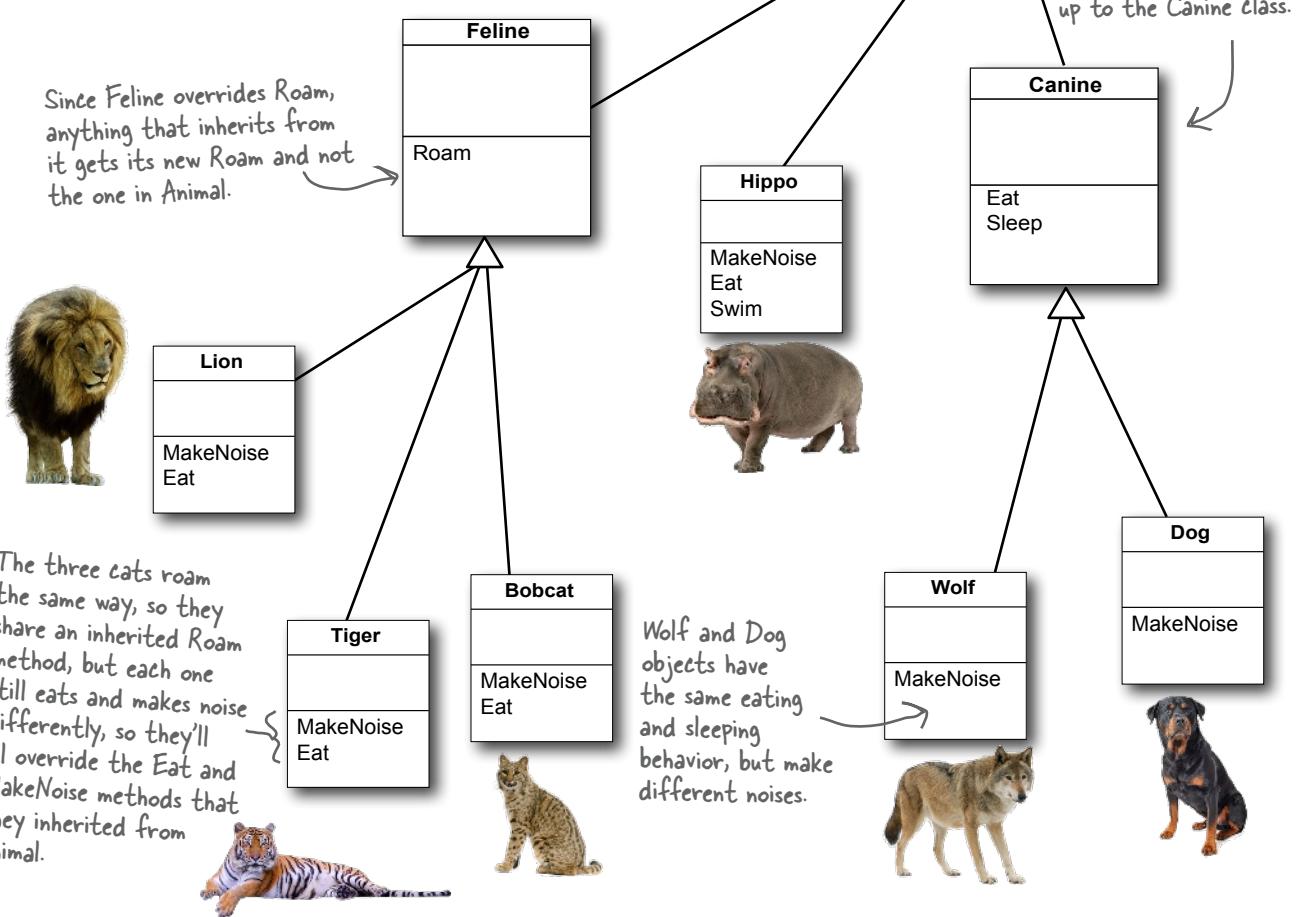
Now that you know how you'll organize the animals, you can add the Feline and Canine classes.

When you create your classes so that there's a base class at the top with subclasses below it, and those subclasses have their own subclasses that inherit from them, what you've built is called a **class hierarchy**. This is about more than just avoiding duplicate code, although that is certainly a great benefit of a sensible hierarchy. One benefit of this is code that's a lot easier to understand and maintain.

When you're looking at the zoo simulator code and you see a method or property defined in the Feline class, then you *immediately* know that you're looking at something that all of the cats share. Your hierarchy becomes a map that helps you find your way through your program.

Animal
Picture Food Hunger Boundaries Location
MakeNoise Eat Sleep Roam

Our wolves and dogs eat the same way, so we moved their common Eat method up to the Canine class.



Every subclass extends its base class

You're not limited to the methods that a subclass inherits from its base class...but you already know that! After all, you've been building your own classes all along. When you modify a class to make it inherit members—and we'll see that in C# code soon!—what you're doing is taking the class you've already built and **extending** it by adding all of the fields, properties, and methods in the base class. So if you want to add a Fetch method to Dog, that's perfectly normal. It won't inherit or override anything—only the Dog class will have that method, and it won't end up in Wolf, Canine, Animal, Hippo, or any other class.

MAKES A NEW INSTANCE OF Dog **Dog spot = new Dog();**

CALLS THE VERSION IN Dog

spot.MakeNoise();

CALLS THE VERSION IN ANIMAL

spot.Roam();

CALLS THE VERSION IN CANINE

spot.Eat();

CALLS THE VERSION IN CANINE

spot.Sleep();

CALLS THE VERSION IN Dog

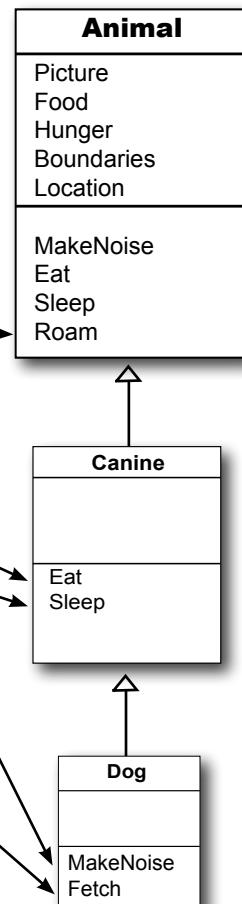
spot.Fetch();

C# always calls the most specific method

If you tell your Dog object to roam, there's only one method that can be called—the one in the Animal class. What about telling your Dog to make noise? Which MakeNoise is called?

Well, it's not too hard to figure it out. A method in the Dog class tells you how dogs make noise. If it's in the Canine class, it's telling you how all canines do it. If it's in Animal, then it's a description of that behavior that's so general that it applies to every single animal. So if you ask your Dog to make a noise, first C# will look inside the Dog class to find the behavior that applies specifically to dogs. If Dog didn't have a MakeNoise method it'd check Canine, and after that it'd check Animal.

hi-er-ar-chy, noun.
an arrangement or classification
in which groups or things are
ranked one above the other. *The
president of Dynamco had worked
her way up from the mailroom to the
top of the corporate **hierarchy**.*



you need a bird, here's a woodpecker

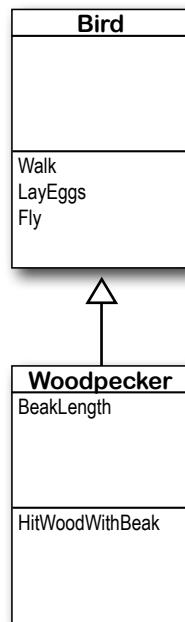
Any place where you can use a base class, you can use one of its subclasses instead

One of the most useful things you can do with inheritance is to **extend** a class. So if your method takes a Bird object, then you can pass an instance of Woodpecker. All that method knows is that it has a bird. It doesn't know what kind of bird it has, so it can only ask it to do things that all birds do: it can ask the bird to Walk and LayEggs, but it can't ask it to HitWoodWithBeak, because only Woodpeckers have that behavior—and the method doesn't know that it's specifically a Woodpecker, just that it's a more general Bird. It **only has access to the fields, properties, and other methods that are part of the class it knows about.**

Let's see how this works in code. Here's a method that takes a Bird reference:

```
public void IncubateEggs(Bird bird)
{
    bird.Walk(incubatorEntrance);
    Egg[] eggs = bird.LayEggs();
    AddEggsToHeatingArea(eggs);
    bird.Walk(incubatorExit);
}
```

Even if we pass a
Woodpecker object to
IncubateEggs, it's a Bird
reference so we can only
use Bird class members.



If you want to incubate some Woodpecker eggs, you can pass a Woodpecker reference to the IncubateEggs method, because a Woodpecker is a *kind of* Bird—which is why it inherits from the Bird class:

```
public void GetWoodpeckerEggs()
{
    Woodpecker woody = new Woodpecker();
    IncubateEggs(woody);
    woody.HitWoodWithBeak();
}
```

You can **replace a superclass with a subclass**, but you can't replace a subclass with its superclass. You can pass a Woodpecker to method that takes a Bird reference, but not vice versa:

```
public void GetWoodpeckerEggs_Take_Two()
{
    Woodpecker woody = new Woodpecker();
    woody.HitWoodWithBeak();
```

This should make
intuitive sense. If
someone asks you for
a bird and you hand
them a woodpecker,
they'll be happy. But
if they ask you for a
woodpecker and you
hand them a pigeon,
they'll be confused.

```
// This line copies the Woodpecker reference to a Bird variable
Bird birdReference = woody;
IncubateEggs(birdReference);
```

You can assign woody to a Bird
variable because a woodpecker is

```
// THE NEXT LINE WILL HAVE A COMPILER ERROR!!!
Woodpecker secondWoodyReference = birdReference;
```

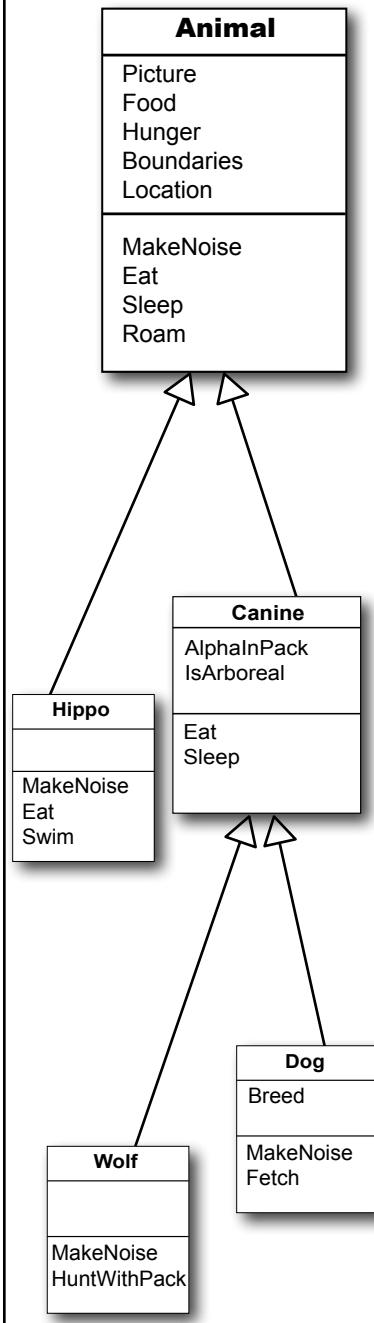
a kind of bird...

```
secondWoodyReference.HitWoodWithBeak();
}
```

...but you can't assign birdReference back to a
Woodpecker variable, because not every bird is a
woodpecker! That's why this line will cause an error.



The code below is from a program that uses the class model that includes Animal, Hippo, Canine, Wolf, and Dog. Draw a line through each statement that won't compile, and write an explanation for the problem next to it.



```

Canine canis = new Dog();
Wolf charon = new Canine();
charon.IsArboreal = false;
Hippo bailey = new Hippo();
bailey.Roam();
bailey.Sleep();
bailey.Swim();
bailey.Eat();
  
```

```

Dog fido = canis;
Animal visitorPet = fido;
Animal harvey = bailey;
harvey.Roam();
harvey.Swim();
harvey.Sleep();
harvey.Eat();
  
```

```

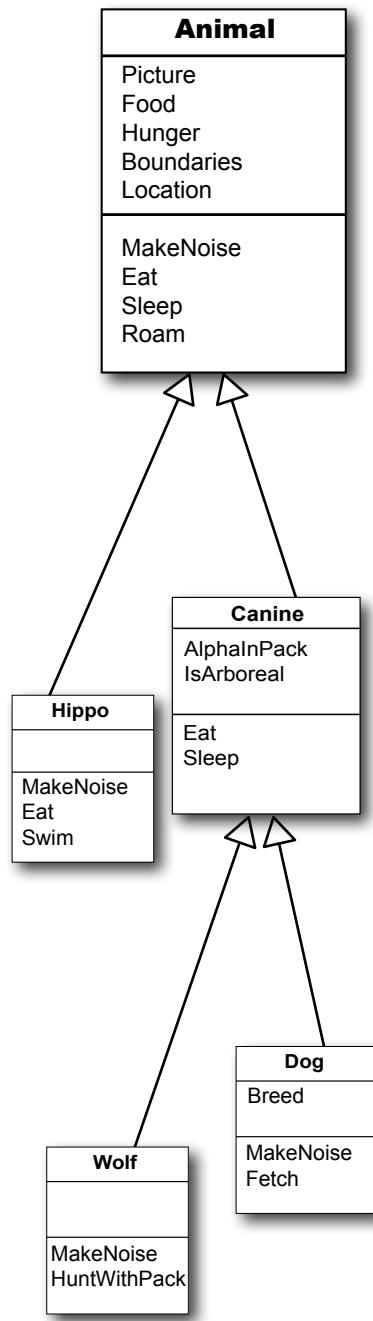
Hippo brutus = harvey;
brutus.Roam();
brutus.Sleep();
brutus.Swim();
brutus.Eat();
  
```

```

Canine london = new Wolf();
Wolf egypt = london;
egypt.HuntWithPack();
egypt.HuntWithPack();
egypt.AlphaInPack = false;
Dog rex = london;
rex.Fetch();
  
```

Sharpen your pencil

Solution



Six of the statements below won't compile because they conflict with the class model. You can test this out yourself! Build your own version of the class model with empty methods, type in the code, and read the compiler errors.

Canine canis = new Dog();
~~Wolf charon = new Canine();~~
charon.IsArboreal = false;
Hippo bailey = new Hippo();
bailey.Roam();
bailey.Sleep();
bailey.Swim();
bailey.Eat();

Wolf is a subclass of Canine, so you can't assign a Canine object to a Wolf. Think of it this way: a wolf is a type of canine, but not every canine is a wolf.

~~Dog fido = canis;~~
Animal visitorPet = fido;

Even though the **canis** variable is a reference to a **Dog** object, the type of the variable is **Canine**, so you can't assign it to a **Dog**.

Animal harvey = bailey;
harvey.Roam();
~~harvey.Swim();~~
~~harvey.Sleep();~~
~~harvey.Eat();~~

harvey is a reference to a **Hippo** object, but the **harvey** variable is an **Animal** so you can't use it to call the **Hippo.Swim** method.

~~Hippo brutus = harvey;~~
brutus.Roam();
brutus.Sleep();
brutus.Swim();
brutus.Eat();

This doesn't work for the same reason that **Dog fido = canis**; didn't work. **harvey** may point to a **Hippo** object, but its type is **Animal**, and you can't assign an **Animal** to a **Hippo** variable.

Canine london = new Wolf();
~~Wolf egypt = london;~~
egypt.HuntWithPack();
~~egypt.HuntWithPack();~~
egypt.AlphaInPack = false;
~~Dog rex = london;~~
rex.Fetch();

This is the same problem! You can assign a **Wolf** to a **Canine**, but you can't assign a **Canine** to a **Wolf**...

...and you definitely can't assign a **Wolf** to a **Dog**.



THIS IS ALL GREAT...IN THEORY. HOW IS IT GOING TO HELP WITH MY DAMAGE CALCULATOR APP?

BRAIN POWER

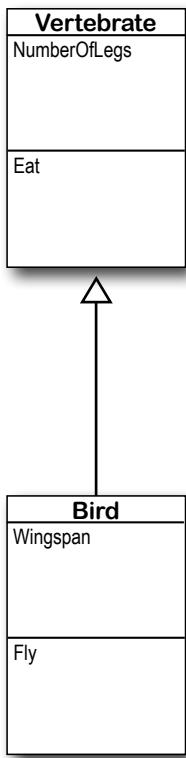
Owen asked a really good question. Go back to the app you built for Owen that calculates sword and arrow damage. How would you use inheritance and subclasses to improve the code? (Spoiler alert: you'll be doing that later in the chapter!)

BULLET POINTS

- A **switch statement** lets you compare one variable against many values. Each case executes code if its value matches. The default block runs if no cases match.
- **Inheritance** lets you build classes that are related to each other and share behavior. Use arrows to show inheritance in a class diagram.
- When you have two classes that are **specific** cases of something more **general**, you can set them up to inherit from the same general class. When you do that, each of them is a **subclass** of the same general **base class**.
- When you build a set of classes that represent things, it's called a **class model**. It can include classes that form a **hierarchy** of subclasses and base class.
- The terms **parent**, **superclass**, and **base class** are often used interchangeably. Also, the terms **extend** and **inherit from** mean the same thing.
- The terms **child** and **subclass** mean the same thing. We say that a subclass **extends** its base class. (The word **subclass** can also be used as a verb.)
- When a subclass changes the behavior of one of the methods that it inherited, we say that it **overrides** the method.
- C# always calls the **most specific method**. If a method in the base class uses a method or property that the subclass overrides, it will call the overridden version in the subclass.
- Always **use a subclass reference** in place of a base class. If a method takes an Animal parameter and Dog extends Animal, you can pass it a Dog argument.
- You can always **use a subclass in place of the base class** it inherits from, but you can't always use a base class in place of a subclass that extends it.

Use a colon to extend a base class

When you're writing a class, you use a **colon** (**:**) to have it inherit from a base class. That makes it a subclass, and gives it **all of the fields, properties, and methods** of the class it inherits from. This Bird class is a subclass of Vertebrate:



```
class Vertebrate
{
    public int Legs { get; set; }

    public void Eat() {
        // code to make it eat
    }
}
```

The Bird class uses a colon to inherit from the Vertebrate class. This means that it inherits all of the fields, properties, and methods from Vertebrate.

```
class Bird : Vertebrate
{
    public double Wingspan;
    public void Fly() {
        // code to make a bird fly
    }
}
```

The base class follows the colon in the class declaration. In this case, Bird extends Vertebrate.

```
public void Main(string[] args) {
    Bird tweety = new Bird();
    Console.WriteLine(tweety.Wingspan);
    tweety.Fly();
    tweety.Legs = 2;
    Console.WriteLine(tweety.Eat());
}
```

tweety is an instance of Bird, so it has the Bird methods, properties, and fields as usual.

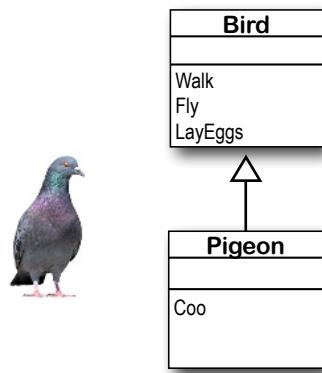
When a subclass extends a base class, it inherits its **members**: all of the fields, properties, and methods in the base class. They're automatically added to the subclass.



Since the Bird class extends Vertebrate, every instance of Bird also has the members defined in the Vertebrate class.

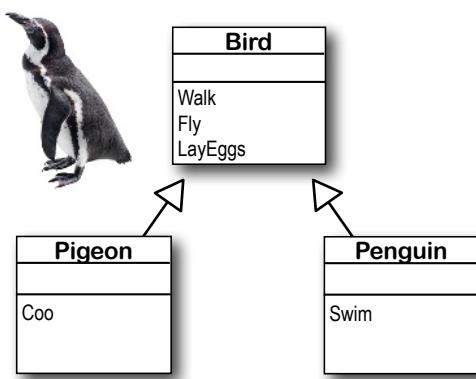
We know that inheritance adds the base class fields, properties, and methods to the subclass...

We've seen inheritance when a subclass needs to inherit **all** of the base class's methods, properties, and fields.



...but some birds don't fly!

What do you do if your base class has a method that your subclass needs to **modify**?



Oops—we've got a problem.
Penguins are birds, and the Bird class has a Fly method, but we don't want our penguins to fly. It would be great if we could display a warning if a penguin tries to fly.

```

class Bird {
    public void Fly() {
        /* code to make birds fly */
    }
    public void LayEggs() { ... };
    public void PreenFeathers() { ... };
}

class Pigeon : Bird {
    public void Coo() { ... }
}

public void SimulatePigeon() {
    Pigeon Harriet = new Pigeon();

    // Since Pigeon is a subclass of Bird,
    // we can call methods from either class.
    Harriet.Walk();
    Harriet.LayEggs();
    Harriet.Coo();
    Harriet.Fly();
}

class Penguin : Bird {
    public void Swim() { ... }
}

public void SimulatePenguin() {
    Penguin Izzy = new Penguin();
    Izzy.Walk();
    Izzy.LayEggs();
    Izzy.Swim();
    Izzy.Fly(); ← This code will compile because Penguin
                  extends Bird. Is there a way to
                  change the Penguin class so it displays
                  a warning if a penguin tries to fly?
}
  
```



If these classes were in your zoo simulator, what would you do about flying penguins?

A subclass can override methods to change or replace members it inherited

Sometimes you've got a subclass that you'd like to inherit *most* of the behaviors from the base class, but *not all of them*. When you want to change the behaviors that a class has inherited, you can **override methods or properties**, replacing them with new members with the same name.

When you **override a method**, your new method needs to have exactly the same signature as the method in the base class it's overriding. In the case of the penguin, that means it needs to be called Fly, return void, and have no parameters.

o-ver-ride, verb.
to use authority to replace, reject, or cancel. *Once she became president of Dynamco, she could **override** poor management decisions.*

① Add the **virtual** keyword to the method in the base class.

A subclass can only override a method if it's marked with the **virtual** keyword. Adding **virtual** to the Fly method declaration tells C# that a subclass of the Bird class is allowed to override the Fly method.

```
class Bird {  
    public virtual void Fly() {  
        // code to make the bird fly  
    }  
}
```

Adding the **virtual** keyword to the Fly method tells C# that a subclass is allowed to override it.

② Add the **override** keyword to a method with the same name in the subclass.

The subclass's method will need to have exactly the same signature—the same return type and parameters—and you'll need to use the **override** keyword in the declaration. Now a Penguin object prints a warning when its Fly method is called.

```
class Penguin : Bird {  
    public override void Fly() {  
        Console.Error.WriteLine("WARNING");  
        Console.Error.WriteLine("Flying Penguin Alert");  
    }  
}
```

To override the Fly method, add an identical method to the subclass and use the **override** keyword.

We used `Console.Error` to write error messages to the standard error stream (`stderr`), which is typically used by console apps to print error messages and important diagnostic information.





Mixed Messages

Exercise

Instructions:

- 1. Fill in the four blanks in the code.**
- 2. Match the code candidates to the output.**

```
class A {
    public int ivar = 7;
    public _____ string m1() {
        return "A's m1, ";
    }
    public string m2() {
        return "A's m2, ";
    }
    public _____ string m3() {
        return "A's m3, ";
    }
}

class B : A {
    public _____ string m1() {
        return "B's m1, ";
    }
}
```

a = 6;
b = 5;
a = 5;

A short C# program is listed below. One block of the program is missing! Your challenge is to match the candidate block of code (on the left) with the output—what's in the message box that the program pops up—that you'd see if the block were inserted. Not all the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching output.

```
class C : B {
    public _____ string m3() {
        return "C's m3, " + (ivar + 6);
    }
}

Here's the entry point for the program.
class Mixed5 {
    public static void Main(string[] args) {
        A a = new A();
        B b = new B();
        C c = new C();
        A a2 = new C(); ← Hint: think really hard about what this line really means.
        string q = "";
        _____
        Console.WriteLine(q);
    }
}
```

Candidate code goes here (three lines)

Code candidates: `q += b.m1();
q += c.m2();
q += a.m3(); }
_____`

Draw a line from each three-line code candidate to the line of output that's produced if you use the candidate in the box.

`q += c.m1();
q += c.m2(); }
q += c.m3();
_____`

`q += a.m1();
q += b.m2(); }
q += c.m3();
_____`

`q += a2.m1();
q += a2.m2(); }
q += a2.m3(); }
_____`

Lines of output:

A's m1, A's m2, C's m3, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, C's m3, 6

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13

(Don't just type this into the IDE—you'll learn a lot more if you figure this out on paper!)



Exercise Solution

```
class A {
    public virtual string m1() {
        ...
    }
    public virtual string m3() {
    }
}
```

You can always substitute a reference to a subclass in place of a base class because you're using something more specific in place of something more general. So this line:

```
A a2 = new C();
```

means that you're instantiating a new C object, and then creating an A reference called a2 and pointing it at that object. Names like these make for a good puzzle, but they're pretty hard to understand. Here are a few lines that follow the same pattern, but have names that are more obvious:

```
Canine fido = new Dog();
Bird pidge = new Pigeon();
Feline rex = new Lion();
```

Mixed Messages

```
a = 6;      56
b = 5;      11
a = 5;      65
```

```
class B : A {
    public override string m1() {
        ...
    }
}
```

```
class C : B {
    public override string m3() {
        ...
    }
}
```

```
q += b.m1();
q += c.m2();
q += a.m3(); }
```

A's m1, A's m2, C's m3, 6

```
q += c.m1();
q += c.m2(); }
```

B's m1, A's m2, A's m3,

```
q += c.m3(); }
```

A's m1, B's m2, C's m3, 6

```
q += a.m1();
q += b.m2(); }
```

B's m1, A's m2, C's m3, 13

```
q += c.m3(); }
```

B's m1, C's m2, A's m3,

```
q += a2.m1();
q += a2.m2(); }
```

A's m1, B's m2, A's m3,

```
q += a2.m3(); }
```

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13

there are no Dumb Questions

Q: A switch statement does exactly the same thing as a series of if/else statements, right? Isn't that redundant?

A: Not at all. There are many situations where switch statements are much more readable than if/else statements. For example, let's say you're displaying a menu in a console app, and the user can press a key to choose one of 10 different options. What would 10 if/else if statements in a row look like? We think a switch statement would be cleaner and easier to read. You could see at a glance exactly what's being compared, where each option is handled, and what happens in the default case if the user chooses an unsupported option. Also, it's surprisingly easy to accidentally leave off an else. If you miss an else in the middle of a long string of if/else statements, you end up with a really annoying bug that's surprisingly difficult to track down. There are some times when a switch statement is easier to read, and other times when if/else statements are easier. It's up to you to write code in the way that you think is easiest to understand.

Q: Why does the arrow point up, from the subclass to the base class? Wouldn't the diagram look better with the arrow pointing down instead?

A: It might seem more intuitive, but it wouldn't be as accurate. When you set up a class to inherit from another one, you build that relationship into the subclass—the base class remains the same. Its behavior is completely unchanged when you add a class that inherits from it. The base class isn't even aware of this new class. Its methods, fields, and properties remain entirely intact—but the subclass definitely changes its behavior. Every instance of the subclass automatically gets all of the properties, fields, and methods from the base class, and it all happens just by adding a colon. That's why you draw the arrow on your diagram so that it points from the subclass to the base class that it inherits from.



Exercise

Let's get some practice extending a base class. We'll give you the Main method for a program that tracks birds laying eggs. Your job is to implement two subclasses of the Bird class.

1. Here's the Main method. It prompts the user for the type of bird and number of eggs to lay:

```
static void Main(string[] args)
{
    while (true)
    {
        Bird bird;
        Console.Write("\nPress P for pigeon, O for ostrich: ");
        char key = Char.ToUpper(Console.ReadKey().KeyChar);
        if (key == 'P') bird = new Pigeon();
        else if (key == 'O') bird = new Ostrich();
        else return;
        Console.Write("\nHow many eggs should it lay? ");
        if (!int.TryParse(Console.ReadLine(), out int numberOfEggs)) return;
        Egg[] eggs = bird.LayEggs(numberOfEggs);
        foreach (Egg egg in eggs)
        {
            Console.WriteLine(egg.Description);
        }
    }
}
```

2. Add this Egg class—the constructor sets the size and color:

```
class Egg
{
    public double Size { get; private set; }
    public string Color { get; private set; }
    public Egg(double size, string color)
    {
        Size = size;
        Color = color;
    }
    public string Description {
        get { return $"A {Size:0.0}cm {Color} egg"; }
    }
}
```

3. This is the Bird class that you'll extend:

```
class Bird
{
    public static Random Randomizer = new Random();
    public virtual Egg[] LayEggs(int numberOfEggs)
    {
        Console.Error.WriteLine("Bird.LayEggs should never get called");
        return new Egg[0];
    }
}
```

4. Create the Pigeon class that extends Bird. Override the LayEggs method and have it lay eggs with the color "white" and a size between 1 and 3 centimeters.
5. Create the Ostrich class that also extends Bird. Override the LayEggs method and have it lay eggs with the color "speckled" and a size between 12 and 13 centimeters.

Here's what the program output looks like:

```
Press P for pigeon, O for ostrich: P
How many eggs should it lay? 4
A 3.0cm white egg
A 1.1cm white egg
A 2.4cm white egg
A 1.9cm white egg

Press P for pigeon, O for ostrich: O
How many eggs should it lay? 3
A 12.1cm speckled egg
A 13.0cm speckled egg
A 12.8cm speckled egg
```



Exercise Solution

Here are the Pigeon and Ostrich classes. They each have their own version of the LayEggs method that uses the `override` keyword in the method declaration. The `override` keyword causes the method in the subclass to replace the one that it inherited.

Pigeon is a subclass of Bird, so if you override the LayEggs method, when you create a new Pigeon object and assign it to a Bird variable called `bird`, `calling bird.LayEggs` will call the LayEggs method you defined in Pigeon.

```
class Pigeon : Bird
{
    public override Egg[] LayEggs(int numberOfEggs)
    {
        Egg[] eggs = new Egg[numberOfEggs];
        for (int i = 0; i < numberOfEggs; i++)
        {
            eggs[i] = new Egg(Bird.Randomizer.NextDouble() * 2 + 1, "white");
        }
        return eggs;
    }
}
```

The Ostrich subclass works the same way as Pigeon. In both classes, the `override keyword` in the LayEggs method declaration means that this new method will replace the LayEggs that it inherited from Bird. So all we need to do is have it create a set of eggs that are the right size and color.

```
class Ostrich : Bird
{
    public override Egg[] LayEggs(int numberOfEggs)
    {
        Egg[] eggs = new Egg[numberOfEggs];
        for (int i = 0; i < numberOfEggs; i++)
        {
            eggs[i] = new Egg(Bird.Randomizer.NextDouble() + 12, "speckled");
        }
        return eggs;
    }
}
```



Some members are only implemented in a subclass

All the code we've seen so far that works with subclasses has accessed the members from outside the object—like how the Main method in the code you just wrote calls LayEggs. Inheritance really shines when the base class **uses a method or property that's implemented in the subclass**. Here's an example. Our zoo simulator has vending machines that let patrons buy soda, candy, and feed to give to the animals in the petting zoo area.

```
class VendingMachine
{
    public virtual string Item { get; }

    protected virtual bool CheckAmount(decimal money) {
        return false;
    }

    public string Dispense(decimal money)
    {
        if (CheckAmount(money)) return Item;
        else return "Please enter the right amount";
    }
}
```

VendingMachine is the base class for all vending machines. It has code to dispense items, but those items aren't defined. The method to check if the patron put in the right amount always returns false. Why? Because they **will be implemented in the subclass**. Here's a subclass for dispensing animal feed in the petting zoo:

```
class AnimalFeedVendingMachine : VendingMachine
{
    public override string Item {
        get { return "a handful of animal feed"; }
    }

    protected override bool CheckAmount(decimal money)
    {
        return money >= 1.25M;
    }
}
```

This class uses the **protected** keyword. It's an access modifier that makes a member **public** only to its subclasses, but **private** to every other class.

Using the **override** keyword with a property works just as it does when you override a method.

We're using the **protected** keyword for encapsulation. The CheckAmount method is **protected** because it never needs to be called by another class, so only VendingMachine and its subclasses are allowed to access it.

Use the debugger to understand how overriding works

Let's use the debugger to see exactly what happens when we create an instance of AnimalFeedVendingMachine and ask it to dispense some feed. **Create a new Console App project**, then do this.



- 1 Add the Main method. Here's the code for the method:

```
class Program
{
    static void Main(string[] args)
    {
        VendingMachine vendingMachine = new AnimalFeedVendingMachine();
        Console.WriteLine(vendingMachine.Dispense(2.00M));
    }
}
```

- 2 Add the VendingMachine and AnimalFeedVendingMachine classes. Once they're added, try adding this line of code to the Main method:

```
vendingMachine.CheckAmount(1F);
```

You'll get a compiler error because of the **protected** keyword, because only the VendingMachine class or subclasses of it can access its protected methods.

✖ CS0122 'VendingMachine.CheckAmount(decimal)' is inaccessible due to its protection level

Delete the line so your code builds.

- 3 Put a breakpoint on the **first** line of the Main method. Run your program. When it hits the breakpoint, **use Step Into (F10) to execute every line of code one at a time**. Here's what happens:

- ★ It creates an instance of AnimalFeedVendingMachine and calls its Dispense method.
- ★ That method is only defined in the base class, so it calls VendingMachine.Dispense.
- ★ The first line of VendingMachine.Dispense calls the protected CheckAmount method.
- ★ CheckAmount is overridden in the AnimalFeedVendingMachine subclass, which causes VendingMachine.Dispense to call the CheckAmount method defined in AnimalFeedVendingMachine.
- ★ This version of CheckAmount returns true, so Dispense returns the Item property. AnimalFeedVendingMachine also overrides this property, it returns "a handful of animal feed."

↑
You've been using the Visual Studio debugger to sleuth out bugs in your code. It's also a great tool for learning and exploring C#, like in this "Debug this!" where you can explore how overriding works. Can you think of more ways to experiment with overriding subclasses?



LOOK, I JUST DON'T SEE WHY I NEED TO USE THOSE "VIRTUAL" AND "OVERRIDE" KEYWORDS. IF I DON'T USE THEM THE IDE GIVES ME A WARNING, BUT THE WARNING DOESN'T ACTUALLY MEAN ANYTHING...**MY PROGRAM STILL RUNS!** I MEAN, I'LL PUT THE KEYWORDS IN IF IT'S THE "RIGHT" THING TO DO, BUT IT JUST SEEMS LIKE I'M JUMPING THROUGH HOOPS FOR NO GOOD REASON.

There's an important reason for virtual and override!

The `virtual` and `override` keywords aren't just for decoration. They make a real difference in how your program works. The `virtual` keyword tells C# that a member (like a method, property, or field) can be extended—without it, you can't override it at all. The `override` keyword tells C# that you're extending the member. If you leave out the `override` keyword in a subclass, you're creating a *completely unrelated* method that just *happens to have the same name*.

That sounds a little weird, right? But it actually makes sense—and the best way to really understand how `virtual` and `override` work is by writing code. So let's build a real example to experiment with them.

When a subclass overrides a method in its base class, the more specific version defined in the subclass is always called—even when it's being called by a method in the base class.

Build an app to explore virtual and override

A really important part of inheritance in C# is extending class members. That's how a subclass can inherit some of its behavior from its base class, but override certain members where it needs to—and that's where the **virtual** and **override** keywords come in. The **virtual** keyword determines which class members can be extended. When you want to extend a member, you **must** use the **override** keyword. Let's create some classes to experiment with **virtual** and **override**. You're going to create a class that represents a safe containing valuable jewels—you'll build a class for some sneaky thieves to steal the jewels.



1 Create a new console app and add the Safe class.

Here's the code for the Safe class:

```
class Safe
{
    private string contents = "precious jewels";
    private string safeCombination = "12345";

    public string Open(string combination)
    {
        if (combination == safeCombination) return contents;
        return "";
    }

    public void PickLock(Locksmith lockpicker)
    {
        lockpicker.Combination = safeCombination;
    }
}
```

Do this!

A Safe object keeps valuables in its “contents” field. It doesn't return them unless Open is called with the right combination...or if a locksmith picks the lock.

We're going to add a Locksmith class that can pick the combination lock and get the combination by calling the PickLock method and passing in a reference to itself. The Safe will use its write-only Combination property to give the Locksmith the combination.

2 Add a class for the person who owns the safe.

The safe owner is forgetful and occasionally forgets their extremely secure safe password. Add a SafeOwner class to represent them:

```
class SafeOwner
{
    private string valuables = "";
    public void ReceiveContents(string safeContents)
    {
        valuables = safeContents;
        Console.WriteLine($"Thank you for returning my {valuables}!");
    }
}
```

3 Add Locksmith class that can pick the lock.

If a safe owner hires a professional locksmith to open their safe, they expect that locksmith to return the contents safe and sound. That's exactly what the Locksmith.OpenSafe method does:

```
class Locksmith
{
    public void OpenSafe(Safe safe, SafeOwner owner)
    {
        safe.PickLock(this);
        string safeContents = safe.Open(Combination);
        ReturnContents(safeContents, owner);
    }

    public string Combination { private get; set; }

    protected void ReturnContents(string safeContents, SafeOwner owner)
    {
        owner.ReceiveContents(safeContents);
    }
}
```

The Locksmith's OpenSafe method picks the lock, opens the safe, and then calls ReturnContents to get the valuables safely back to the owner.



4 Add a JewelThief class that wants to steal the valuables.

Uh-oh. Looks like there's a burglar—and the worst kind, one who's also a highly skilled locksmith able to open safes. Add this JewelThief class that extends Locksmith:

```
class JewelThief : Locksmith
{
    private string stolenJewels;
    protected void ReturnContents(string safeContents, SafeOwner owner)
    {
        stolenJewels = safeContents;
        Console.WriteLine($"I'm stealing the jewels! I stole: {stolenJewels}");
    }
}
```

JewelThief extends Locksmith and inherits the OpenSafe method and Combination property, but its ReturnContents method steals the jewels instead of returning them. INGENIOUS!

5 Add a Main method that makes the JewelThief steal the jewels.

It's time for the big heist! In this Main method, the JewelThief sneaks into the house and uses its inherited Locksmith.OpenSafe method to get the safe combination. **What do you think will happen when it runs?**

```
static void Main(string[] args)
{
    SafeOwner owner = new SafeOwner();
    Safe safe = new Safe();
    JewelThief jewelThief = new JewelThief();
    jewelThief.OpenSafe(safe, owner);
    Console.ReadKey(true);
}
```



Read through the code for your program. Before you run it, write down what you think it will print to the console. (Hint: figure out what the JewelThief class inherits from Locksmith.)

A subclass can hide methods in the base class

Go ahead and run the JewelThief program. Here's what you should see:



Thank you for returning my precious jewels!

Did you expect the program's output to be different? Maybe something like this:

I'm stealing the jewels! I stole: precious jewels

It looks like the JewelThief object acted just like a Locksmith object! So what happened?

C# is supposed to call the most specific method, right?
Then why didn't it call JewelThief.ReturnContents?

Hiding methods versus overriding methods

The reason the JewelThief object acted like a Locksmith object when its ReturnContents method was called was because of the way the JewelThief class declared its ReturnContents method. There's a big hint in that warning message you got when you compiled your program:

CS0108 'JewelThief.ReturnContents(string, SafeOwner)' hides inherited member 'Locksmith.ReturnContents(string, SafeOwner)'. Use the new keyword if hiding was intended.

Since the JewelThief class inherits from Locksmith and replaces the ReturnContents method with its own method, it looks like JewelThief is overriding Locksmith's ReturnContents method—but that's not actually what's happening. You probably expected JewelThief to override the method (which we'll talk about in a minute), but instead JewelThief is hiding it.

JewelThief
Locksmith.ReturnContents JewelThief.ReturnContents

There's a big difference. When a subclass **hides** a method, it replaces (technically, it redeclares) a method in its base class **that has the same name**. So now our subclass really has two different methods that share a name: one that it inherits from its base class, and another brand-new one that's defined in that class.

Use the new keyword when you're hiding methods

Take a close look at that warning message. Sure, we know we *should* read our warnings, but sometimes we don't...right? This time, actually read what it says: **Use the new keyword if hiding was intended.**

So go back to your program and add the **new** keyword:

```
new public void ReturnContents(Jewels safeContents, Owner owner)
```

As soon as you add **new** to your JewelThief class's ReturnContents method declaration, that warning message will go away—but your code still won't act the way you expect it to!

It still calls the ReturnContents method defined in the Locksmith class. Why? Because the ReturnContents method is being called **from a method defined by the Locksmith class**—specifically, from inside Locksmith.OpenSafe—even though it's being initiated by a JewelThief object. If JewelThief only hides Locksmith's ReturnContents method, its own ReturnContents method will never be called.

If a subclass just adds a method with the same name as a method in its base class, it only hides the base class method instead of overriding it.

Use different references to call hidden methods

Now we know that JewelThief only **hides** the ReturnContents method (as opposed to **overriding** it). That causes it to act like a Locksmith object **whenever it's called like a Locksmith object**. JewelThief inherits one version of ReturnContents from Locksmith, and it defines a second version of it, which means that there are two different methods with the same name. That means your class needs **two different ways to call it**.

There are two different ways to call the ReturnContents method. If you've got an instance of JewelThief, you can use a JewelThief reference variable to call the new ReturnContents method. If you use a Locksmith reference variable to call it, it will call the hidden Locksmith ReturnContents method.

Here's how that works:

```
// The JewelThief subclass hides a method in the Locksmith base class,
// so you can get different behavior from the same object based on the
// reference you use to call it!

// Declaring your JewelThief object as a Locksmith reference causes it to
// call the base class ReturnContents() method.
Locksmith calledAsLocksmith = new JewelThief();
calledAsLocksmith.ReturnContents(safeContents, owner);

// Declaring your JewelThief object as a JewelThief reference causes it to
// call JewelThief's ReturnContents() method instead, because it hides
// the base class's method of the same name.
JewelThief calledAsJewelThief = new JewelThief();
calledAsJewelThief.ReturnContents(safeContents, owner);
```

Can you figure out how to get JewelThief to override the ReturnContents method instead of just hiding it? See if you can do it before reading the next section!

there are no
Dumb Questions

Q: I still don't get why they're called "virtual" methods—they seem real to me. What's virtual about them?

A: The name "virtual" has to do with how .NET handles the virtual methods behind the scenes. It uses something called a *virtual method table* (or *vtable*). That's a table that .NET uses to keep track of which methods are inherited and which ones have been overridden. Don't worry—you don't need to know how it works to use virtual methods.

Q: You talked about replacing a superclass with a reference to a subclass. Can you go over that one more time?

A: When you've got a diagram with one class that's above another one, the class that's higher up is **more abstract** than the one that's lower down. More **specific** or **concrete** classes (like Shirt or Car) inherit from more abstract ones (like Clothing or Vehicle). If all you need is a vehicle, a car or van or motorcycle will do. If you need a car, a motorcycle won't be useful to you.

Inheritance works exactly the same way. If you have a method with Vehicle as a parameter, and if the Motorcycle class inherits from the Vehicle class, then you can pass an instance of Motorcycle to the method. If the method takes Motorcycle as a parameter, you can't pass any Vehicle object, because it may be a Van instance. Then C# won't know what to do when the method tries to access the Handlebars property.

Use the override and virtual keywords to inherit behavior

We really want our JewelThief class to always use its own ReturnContents method, no matter how it's called. This is the way we expect inheritance to work most of the time: a subclass can **override** a method in the base class so the method in the subclass is called instead. Start by using the **override** keyword when you declare the ReturnContents method:

```
class JewelThief {  
    protected override void ReturnContents  
        (string safeContents, SafeOwner owner)
```

But that's not everything you need to do. If you just add the **override** keyword to the class declaration, you'll get a compiler error:

 CS0506 'JewelThief.ReturnContents(string, SafeOwner)': cannot override inherited member 'Locksmith.ReturnContents(string, SafeOwner)' because it is not marked virtual, abstract, or override

Again, take a really close look and read what the error says. JewelThief can't override the inherited member ReturnContents **because it's not marked** **virtual**, **abstract**, or **override** in Locksmith. Well, that's an error we can fix with a quick change. Mark Locksmith's ReturnContents with the **virtual** keyword:

```
class Locksmith {  
    protected virtual void ReturnContents  
        (string safeContents, SafeOwner owner)
```

Now run your program again. Here's the output we've been looking for:

I'm stealing the jewels! I stole: precious jewels



Sharpen your pencil

Draw a line from each of the following descriptions to the keyword it describes.

1. A method that can only be **accessed by an instance of the same class**
2. A method that **a subclass can replace** with a method of the same name
3. A method that can be **accessed by an instance of any other class**
4. A method that **hides another method in the superclass** with the same name
5. A method that **replaces a method in the superclass**
6. A method that can only be **accessed by a member of the class or its subclass**

virtual

new

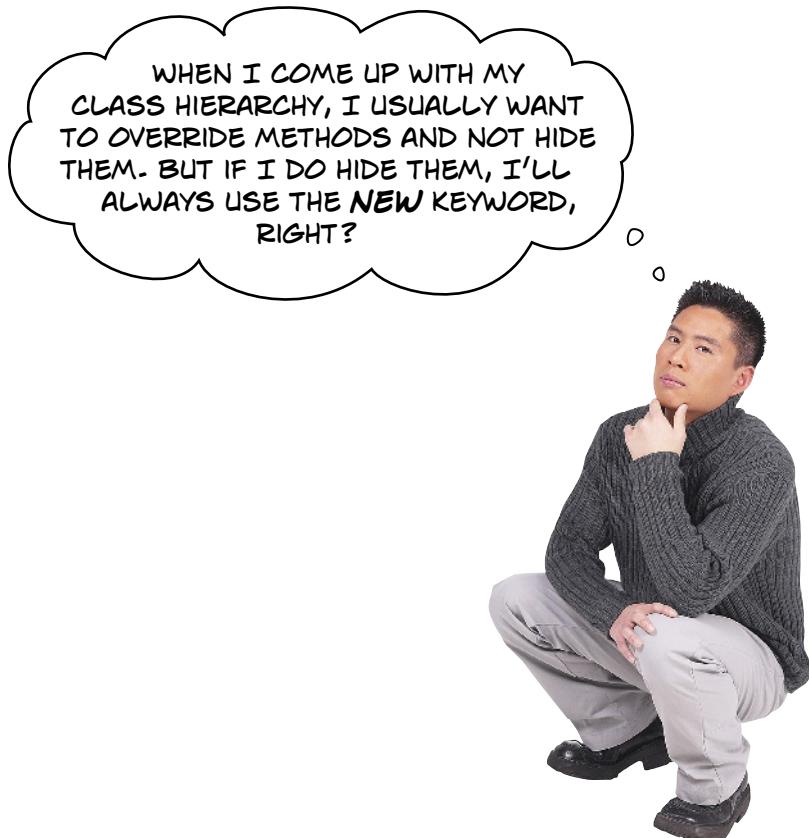
override

protected

private

public

1. **private** 2. **virtual** 3. **public** 4. **new** 5. **override** 6. **protected**



WHEN I COME UP WITH MY CLASS HIERARCHY, I USUALLY WANT TO OVERRIDE METHODS AND NOT HIDE THEM. BUT IF I DO HIDE THEM, I'LL ALWAYS USE THE `NEW` KEYWORD, RIGHT?

Exactly. Most of the time you want to override methods, but hiding them is an option.

When you're working with a subclass that extends a base class, you're much more likely to use overriding than you are to use hiding. So when you see that compiler warning about hiding a method, pay attention to it! Make sure you really want to hide the method, and didn't just forget to use the `virtual` and `override` keywords. If you always use the `virtual`, `override`, and `new` keywords correctly, you'll never run into a problem like this again!

If you want to override a method in a base class, always mark it with the `virtual` keyword, and always use the `override` keyword any time you want to override the method in a subclass. If you don't, you'll end up accidentally hiding methods instead.

A subclass can access its base class using the **base** keyword

Even when you override a method or property in your base class, sometimes you'll still want to access it. Luckily, we can use **base**, which lets us access any member of the base class.

- All animals eat, so the **Vertebrate** class has an **Eat** method that takes a **Food** object as its parameter.

```
class Vertebrate {
    public virtual void Eat(Food morsel) {
        Swallow(morsel);
        Digest();
    }
}
```



Vertebrate
NumberOfLegs

Chameleon
NumberOfLegs
Color
TongueLength

Chameleon
Eat
ChangeColor
GripBranch
CatchWithTongue

- Chameleons eat by catching food with their tongues. So the **Chameleon** class inherits from **Vertebrate** but overrides **Eat**.

```
class Chameleon : Vertebrate {
    public override void Eat(Food morsel) {
        CatchWithTongue(morsel);
        Swallow(morsel);
        Digest();
    }
}
```

This is exactly the same as code in the base class. Do we really need to have two duplicate copies of the same code?

The **Chameleon.Eat** method needs to call **CatchWithTongue**, but after that it's identical to the **Eat** method in the **Vertebrate** base class that it overrides.

- Instead of duplicating the code, we can use the **base** keyword to call the method that was overridden. Now we have access to both the old and the new version of **Eat**.

```
class Chameleon : Vertebrate {
    public override void Eat(Food morsel) {
        CatchWithTongue(morsel);
        base.Eat(morsel);
    }
}
```

We can't just write "Eat(morsel)" because that would call **Chameleon.Eat**. We need to use the "base" keyword to access **Vertebrate.Eat**.

This updated version of the method in the base class uses the **base** keyword to call the Eat method in the base class.

Now we don't have any duplicated code—so if we ever need to change the way all vertebrates eat, chameleons will get the changes automatically.

When a base class has a constructor, your subclass needs to call it

Let's go back to the code you wrote with the Bird, Pigeon, Ostrich, and Egg classes. We want to add a BrokenEgg class that extends Egg, and make 25% of the eggs that the Pigeon lays broken. **Replace the new statement** in Pigeon.LayEgg with this **if/else** that creates a new instance of either Egg or BrokenEgg:

```
if (Bird.Randomizer.Next(4) == 0)
    eggs[i] = new BrokenEgg(Bird.Randomizer.NextDouble() * 2 + 1, "white");
else
    eggs[i] = new Egg(Bird.Randomizer.NextDouble() * 2 + 1, "white");
```

← Add this!

Now we just need a BrokenEgg class that extends Egg. Let's make it identical to the Egg class, except that it has a constructor that writes a message to the console letting us know that an egg is broken:

```
class BrokenEgg : Egg
{
    public BrokenEgg()
    {
        Console.WriteLine("A bird laid a broken egg");
    }
}
```

Go ahead and **make those two changes** to your Egg program.

Uh-oh—looks like those new lines of code caused compiler errors:

- ★ The first error is on the line where you create a new BrokenEgg: *CS1729 – 'BrokenEgg' does not contain a constructor that takes 2 arguments*
- ★ The second error is in the BrokenEgg constructor: *CS7036 – There is no argument given that corresponds to the required formal parameter 'size' of 'Egg.Egg(double, string)'*

This is another great opportunity to **read those errors** and figure out what went wrong. The first error is pretty clear: the statement that creates a BrokenEgg instance is trying to pass two arguments to the constructor, but the BrokenEgg class has a parameterless constructor. So go ahead and **add parameters to the constructor**:

```
public BrokenEgg(double size, string color)
```

That takes care of the first error—now the Main method compiles just fine. What about the other error?

Let's break down what that error says:

- ★ It's complaining about *Egg.Egg(double, string)*—this refers to the Egg class constructor.
- ★ It says something about *parameter 'size'*, which the Egg class needs in order to set its Size property.
- ★ But there is *no argument given*, because it's not enough to just modify the BrokenEgg constructor to take arguments that match the parameter. It also needs to **call that base class constructor**.

Modify the BrokenEgg class to **use the base keyword to call the base class constructor**:

```
public BrokenEgg(double size, string color) : base(size, color)
```

Now your code compiles. Try running it—now when a Pigeon lays an egg, about a quarter of them will print a message about being broken when they're instantiated (but after that, the rest of the output is the same as before).



It's easy to go back to an old project.

You can get the IDE to load a previous project by choosing *Recent Projects and Solutions* (Windows) or *Recent Solutions* (Mac) from the File menu.

A subclass and base class can have different constructors

When we modified BrokenEgg to call the base class constructor, we made its constructor match the one in the Egg base class. What if we want all broken eggs to have a size of zero and a color that starts with the word “broken”? **Modify the statement that instantiates BrokenEgg** so it only takes the color argument:

```
if (Bird.Randomizer.Next(4) == 0)
    eggs[i] = new BrokenEgg("white");
else
    eggs[i] = new Egg(Bird.Randomizer.NextDouble() * 2 + 1, "white");
```

← **Modify this!**

When you make that change you’ll get the “required formal parameter” compiler error again—which makes sense, because the BrokenEgg constructor has two parameters, but you’re only passing it one argument.

Fix your code **by modifying the BrokenEgg constructor to take one parameter**:

```
class BrokenEgg : Egg
{
    public BrokenEgg(string color) : base(0, $"broken {color}")
    {
        Console.WriteLine("A bird laid a broken egg");
    }
}
```

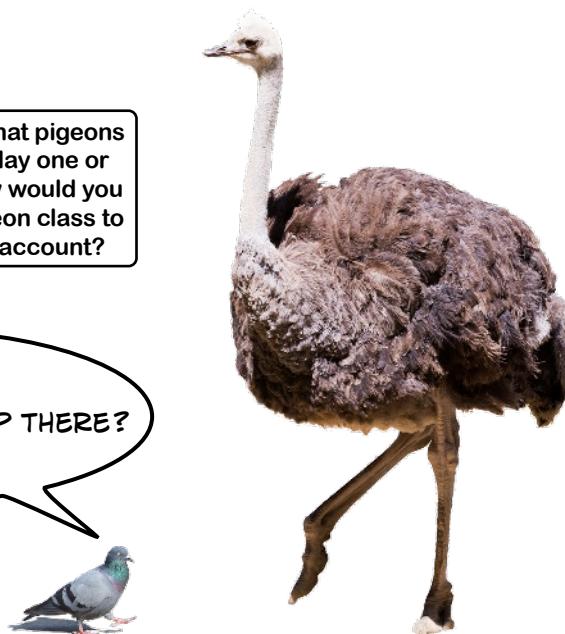
Now run your program again. The BrokenEgg constructor will still write its message to the console during the for loop in the Pigeon constructor, but now it will also cause the Egg to initialize its Size and Color fields. When the foreach loop in the Main method writes egg.Description to the console, it writes this message for each broken egg:

Press P for pigeon, O for ostrich:

```
p
How many eggs should it lay? 7
A bird laid a broken egg
A bird laid a broken egg
A bird laid a broken egg
A 2.4cm white egg
A 0.0cm broken White egg
A 3.0cm white egg
A 1.4cm white egg
A 0.0cm broken White egg
A 0.0cm broken White egg
A 2.7cm white egg
```

Did you know that pigeons typically only lay one or two eggs? How would you modify the Pigeon class to take this into account?

The subclass constructor can have any number of parameters, and it can even be parameterless. It just needs to use the base keyword to pass the correct number of arguments to the base class constructor.





It's time to finish the job for Owen

The first thing you did in this chapter was modify the damage calculator you built for Owen to roll for damage for either a sword or an arrow. It worked, and your `SwordDamage` and `ArrowDamage` classes were well-encapsulated. But aside from a few lines of code, ***those two classes were identical***. You've learned that having code repeated in different classes is inefficient and error-prone, especially if you want to keep extending the program to add more classes for different kinds of weapons. Now you have a new tool to solve this problem: **inheritance**. So it's time for you to finish the damage calculator app. You'll do it in two steps: first you'll design the new class model on paper, and then you'll implement it in code.

Building your class model on paper before you write code helps you understand your problem better so you can solve it more effectively.

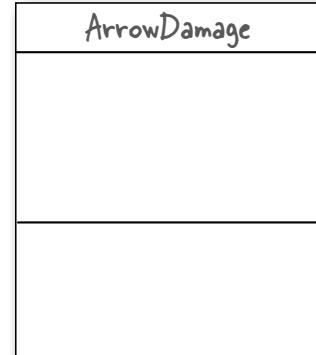
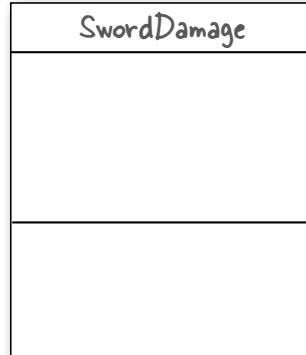
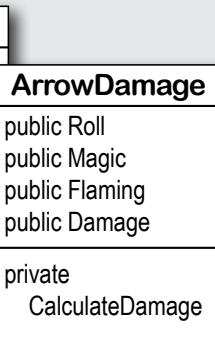
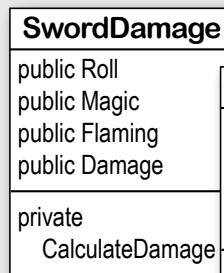
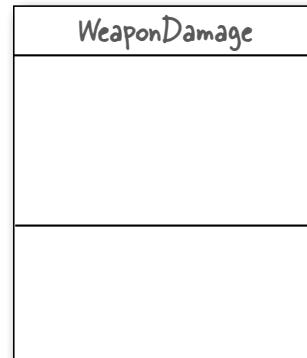


Great code starts in your head, not an IDE. So let's take the time to design the class model on paper *before* we start writing code.

We've started you out by filling in the class names. Your job is to add the members to all three classes, and draw the arrows between the boxes.

For reference, we included diagrams for the `SwordDamage` and `ArrowDamage` classes that you built earlier. We included the private `CalculateDamage` method for each class. Make sure to include all public, private, and protected class members when you fill in the class diagram. Write the access modifier (`public`, `private`, or `protected`) next to each class member.

The `SwordDamage` and `ArrowDamage` classes looked like this at the start of the chapter. They're well-encapsulated, but most of the code in `SwordDamage` is duplicated in `ArrowDamage`.

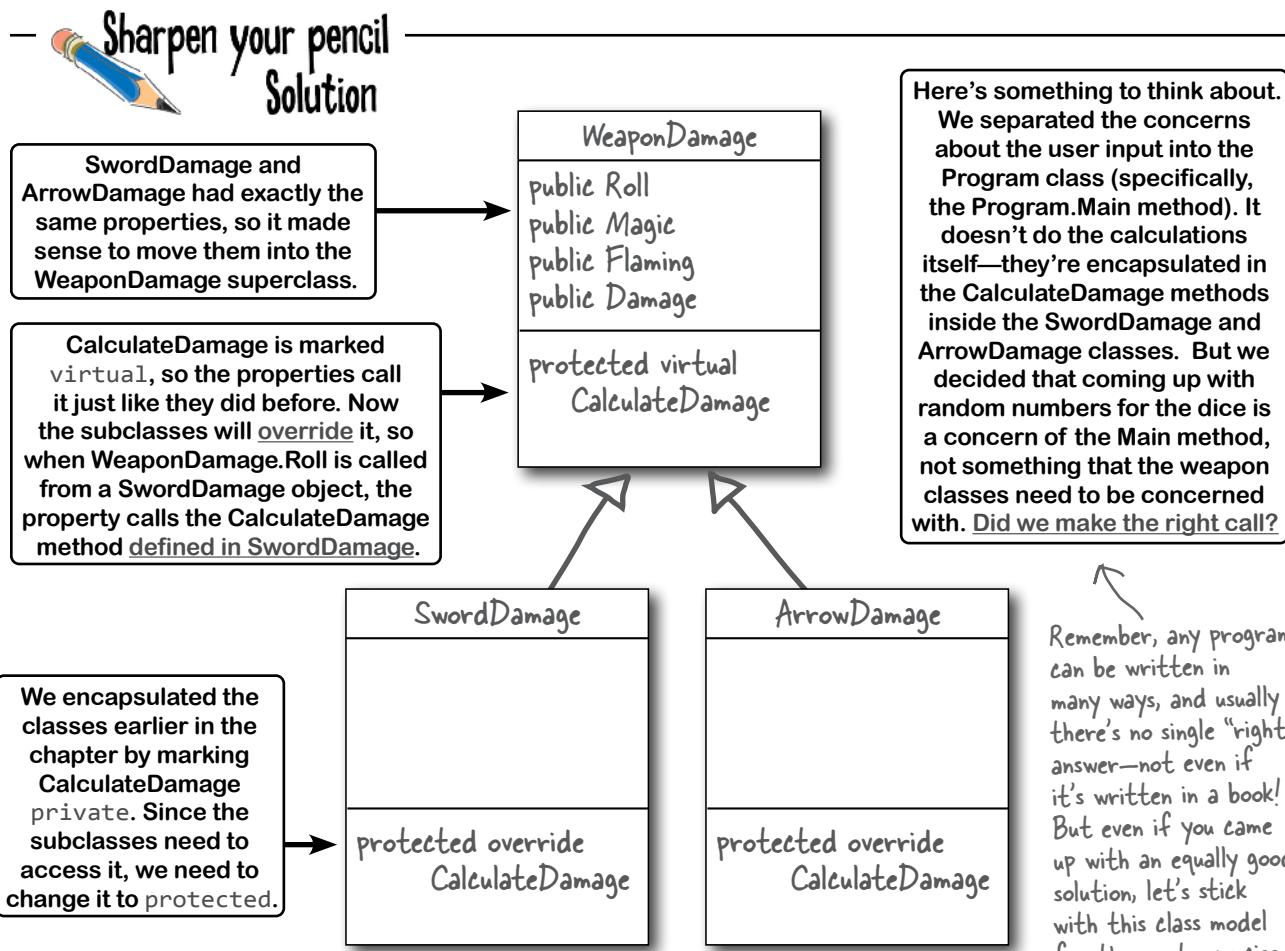


When your classes overlap as little as possible, that's an important design principle called separation of concerns

If you design your classes well today, they'll be easier to modify later. Imagine if you had a dozen different classes to calculate damage for different weapons. What if you wanted to change Magic from a bool to an int, so you could have weapons with enchantment bonuses (like a +3 magic mace or +1 magic dagger)? With inheritance, you'd just have to change the Magic property in the superclass. Of course, you'd have to modify the CalculateDamage method for each class—but it would be a lot less work, and there's no danger of accidentally forgetting to modify one of the classes. (That happens in professional software development *all the time!*)

This is an example of **separation of concerns**, because each class has only the code that concerns one specific part of the problem that your program solves. Code that only concerns swords goes in SwordDamage, code only for arrows goes in ArrowDamage, and code that's shared between them goes in WeaponDamage.

When you're designing classes, separation of concerns is one of the first things you should think about. If one class seems to be doing two different things, try to figure out if you can split it into two classes.





Exercise

Now that you've **designed** the class model, you're ready to write the code to **implement** it. That's a great habit to get into—design your classes first, then turn them into code.

Here's what you'll do to finish the job for Owen. You can reopen the project that you created at the beginning of the chapter, or you can create an entirely new one and copy the relevant parts into it. If your code is very different from the exercise solution earlier in the chapter, you might want to start with the solution code. You can download the code from <https://github.com/head-first-csharp/fourth-edition> if you don't want to type it in.

- Don't make any changes to the Main method.** It will use the new SwordDamage and ArrowDamage classes exactly like it did at the beginning of the chapter.
- Implement the WeaponDamage class.** Add a new WeaponDamage class and make it match the class diagram in the "Sharpen your pencil" solution. Here are a few things to consider:
 - The properties in WeaponDamage are *almost* identical to the properties in the SwordDamage and ArrowDamage classes at the beginning of the chapter. There's just a single keyword that's different.
 - Don't put any code in the CalculateDamage class (you can include a comment: `/* the subclass overrides this */`). It needs to be virtual, it can't be private—otherwise you'll get a compiler error:

`CS0621 'WeaponDamage.CalculateDamage()': virtual or abstract members cannot be private`
 - Add a constructor that sets the starting roll.
- Implement the SwordDamage class.** Here are a few things you need to think about:
 - The constructor has a single parameter, which it passes to the base class constructor.
 - C# always calls the most specific method. That means you'll need to override CalculateDamage and make it do the sword damage calculation.
 - It's worth taking a minute to think about how CalculateDamage works. The Roll, Magic, or Flaming setters call CalculateDamage to make sure the Damage field is updated automatically. Since C# always calls the most specific method, they'll call `SwordDamage.CalculateDamage` even though they're part of the `WeaponDamage` superclass.
- Implement the ArrowDamage class.** It works exactly like SwordDamage, except that its CalculateDamage method does the arrow calculation and not the sword calculation.



WE CAN MAKE SOME PRETTY BIG
CHANGES TO THE WAY THE CLASSES WORK
**WITHOUT MODIFYING THE MAIN METHOD THAT
CALLS THOSE CLASSES.**

When your classes are well-encapsulated, it makes your code much easier to modify.

If you know a professional developer, ask them the most annoying thing they've had to do at work in the last year. There's a good chance they'll talk about having to make a change to a class, but to do that they had to change these two other classes, which required three other changes, and it was hard just to keep track of all the changes. Designing your classes with encapsulation in mind is a great way to avoid ending up in that situation.



Exercise Solution

Here's the code for the WeaponDamage class. The properties are **almost** identical to the properties in the old sword and arrow classes. It also has a constructor to set the starting roll, and a CalculateDamage method for the subclasses to override.

```
class WeaponDamage
{
    public int Damage { get; protected set; }

    private int roll;
    public int Roll
    {
        get { return roll; }
        set
        {
            roll = value;
            CalculateDamage();
        }
    }

    private bool magic;
    public bool Magic
    {
        get { return magic; }
        set
        {
            magic = value;
            CalculateDamage();
        }
    }

    private bool flaming;
    public bool Flaming
    {
        get { return flaming; }
        set
        {
            flaming = value;
            CalculateDamage();
        }
    }

    protected virtual void CalculateDamage() { /* the subclass overrides this */ }

    public WeaponDamage(int startingRoll)
    {
        roll = startingRoll;
        CalculateDamage();
    }
}
```

WeaponDamage
public Roll
public Magic
public Flaming
public Damage
protected virtual
CalculateDamage

The Damage property's get accessor needs to be marked protected. That way the subclasses have access to update it, but no other class can set it. It's still protected from other classes accidentally setting it, so the subclasses will still be well-encapsulated.

The properties still call the CalculateDamage method, which keeps the Damage property updated. Even though they're defined in the superclass, when they're inherited by a subclass they call the CalculateDamage method defined in that subclass.

This is just like how the JewelThief worked when you had it override the method in LockSmith to steal the jewels from the safe instead of returning them.

The CalculateDamage method itself is empty—we're taking advantage of the fact that C# always calls the most specific method. Now that a SwordDamage class extends WeaponDamage, when its inherited Flaming property's set accessor calls CalculateDamage it executes the most specific version of that method, so it calls SwordDamage.CalculateDamage instead.

Use the debugger to really understand how these classes work



One of the most important ideas in this chapter is that when you extend a class, you can override its methods to make pretty significant changes to the way it behaves. Use the debugger to really understand how that works:

- ★ **Set breakpoints** on the lines in the Roll, Magic, and Flaming setters that call CalculateDamage.
- ★ Add a Console.WriteLine statement to WeaponDamage.CalculateDamage. *This statement will never get called.*
- ★ Run your program. When it hits any of the breakpoints, use **Step Into** to enter the CalculateDamage method. ***It will step into the subclass***—the WeaponDamage.CalculateDamage method is never called.

The SwordDamage class extends WeaponDamage and overrides its CalculateDamage method to implement the sword damage calculation. Here's the code:

```
class SwordDamage : WeaponDamage
{
    public const int BASE_DAMAGE = 3;
    public const int FLAME_DAMAGE = 2;

    public SwordDamage(int startingRoll) : base(startingRoll) { }

    protected override void CalculateDamage()
    {
        decimal magicMultiplier = 1M;
        if (Magic) magicMultiplier = 1.75M;

        Damage = BASE_DAMAGE;
        Damage = (int)(Roll * magicMultiplier) + BASE_DAMAGE;
        if (Flaming) Damage += FLAME_DAMAGE;
    }
}
```

All the constructor needs to do is use the base keyword to call the superclass's constructor, using its startingRoll parameter as the argument.



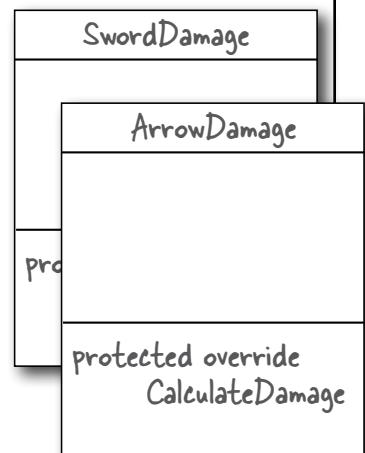
Exercise Solution

And here's the code for the ArrowDamage class. It works just like the SwordDamage class, except it does the calculation for arrows instead:

```
class ArrowDamage : WeaponDamage
{
    private const decimal BASE_MULTIPLIER = 0.35M;
    private const decimal MAGIC_MULTIPLIER = 2.5M;
    private const decimal FLAME_DAMAGE = 1.25M;

    public ArrowDamage(int startingRoll) : base(startingRoll) { }

    protected override void CalculateDamage()
    {
        decimal baseDamage = Roll * BASE_MULTIPLIER;
        if (Magic) baseDamage *= MAGIC_MULTIPLIER;
        if (Flaming) Damage = (int)Math.Ceiling(baseDamage + FLAME_DAMAGE);
        else Damage = (int)Math.Ceiling(baseDamage);
    }
}
```



We're about to talk about an important element of game design:
dynamics. It's actually such an important concept that it goes beyond
game design. In fact, you can find dynamics in almost any kind of app.



Dynamics

Game design... and beyond

The **dynamics** of a game describe how the mechanics combine and cooperate to drive the gameplay. Any time you have game mechanics, they lead to dynamics. That's not limited to video games—all games have mechanics, and dynamics arise out of those mechanics.

- We've already seen a **good example of mechanics**: in Owen's role-playing game, he uses formulas (the ones you built into your damage classes) to calculate damage for various weapons. That's a good starting point to think about how a change in those mechanics would affect dynamics.
- What happens if you change the mechanics of the arrow formula so that it multiplies the base damage by 10? That's a small change in mechanics, but it leads to a **huge change in the dynamics** of the game. Suddenly, arrows are much more powerful than swords. Players will stop using swords and start shooting arrows, even at close range—that's a change in dynamics.
- Once the players start **behaving differently**, Owen will need to change his campaigns. For example, some battles designed to be difficult may suddenly become too easy for the players. That makes the players change again.

Take a minute to think about all of that. A tiny change to the rules leads to a huge change in the way the players behave. A *small change* in mechanics caused a *very large change* in dynamics. Owen didn't make those changes to gameplay directly; they were follow-on effects from his small rule change. In technical terms, the change in dynamics **emerged** from the change in mechanics.

- If you haven't come across the idea of **emergence** before it may seem a little odd, so let's look at a concrete example from a classic video game.
- The **mechanics of Space Invaders** are simple. Aliens march back and forth and fire shots down; if a shot hits the player, they lose a life. The player moves the ship left and right and fires shots up. If a shot hits an alien, it's destroyed. A mothership occasionally flies across the top of the screen for more points. Shields slowly get eaten away by shots. Different aliens add different scores. The aliens march faster as the game goes on. And that's pretty much it.
- The **dynamics of Space Invaders** are more complex. The game starts off very easy—most players can get much of the first wave—but it quickly gets harder and harder. The only thing that changes is the speed at which the invaders march. Somehow as the invaders get faster and faster, it changes the entire game. The **tempo**—how fast the game feels—drastically changes.
- Some players try to shoot the aliens starting at the edge of the formation, because the gap at the side of the formation slows down their descent. That's not written anywhere in the code, which just has simple rules for how the invaders march. That's a dynamic, and it's **emergent** because it's a side effect of how the mechanics combine—specifically, the mechanics of how the player shots work combined with the rules for how the invaders march. None of that is programmed into the code of the game. It's not part of the mechanics. It's all dynamics.



Dynamics can feel like a really abstract concept at first! We'll spend more time on it later in the chapter—but for now, keep all of this stuff about dynamics in mind when you're doing the next project. See if you can spot how dynamics come into play as you're coding it.



YOU KNOW WHAT? I'VE HAD IT UP TO HERE WITH GAMES. MATCHING GAMES, 3D GAMES, NUMBER GAMES, CARD AND PAINTBALL GUN CLASSES THAT GO INTO GAMES, CLASS MODELS FOR GAMES, GAME DESIGN...IT SEEMS LIKE WE'VE BEEN DOING NOTHING BUT GAMES.

LOOK, WE ALL KNOW THAT C# DEVELOPERS CAN MAKE A LOT OF MONEY ON THE JOB MARKET. I WANT TO LEARN C# SO I CAN USE IT FOR A JOB. CAN'T WE JUST HAVE ONE PROJECT WHERE WE DEVELOP A **SERIOUS BUSINESS APPLICATION?**

Video games are serious business.

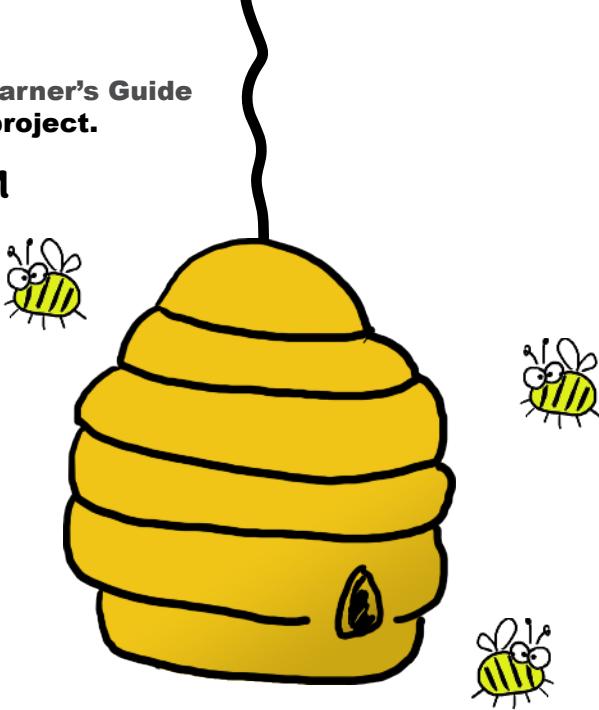
The video game industry is growing globally every year, and employs hundreds of thousands of people all over the world, and it's a business that a talented game designer can break into! There's an entire ecosystem of **independent game developers** who build and sell games, either as individuals or on small teams.

But you're right—C# is a serious language, and it's used for all sorts of serious, non-gaming applications. In fact, while C# is a favorite language among game developers, it's also one of the most common languages found in businesses across many different industries.

So for this next project, let's get some practice with inheritance by building a **serious business application**.

Build a beehive management system

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



① The queen assigns jobs to her workers.

There are three different jobs that the workers can do.

Nectar collector bees fly out and bring nectar back to the hive. **Honey manufacturer** bees turn that nectar into honey, which bees eat to keep working. Finally, the queen is constantly laying eggs, and **egg care** bees make sure they become workers.

② When the jobs are all assigned, it's time to work.

Once the queen's done assigning the work, she'll tell the bees to work the next shift by clicking the "Work the next shift" button in her Beehive Management System app, which generates a shift report that tells her how many bees are assigned to each job and the status of the nectar and honey in the **honey vault**.

The screenshot shows a window titled "Beehive Management System". The left side is labeled "Job Assignments" and contains a dropdown menu set to "Nectar Collector" and a button "Assign this job to a bee". Below these is a dashed-line box containing the text "Work the next shift". The right side is labeled "Queen's Report" and displays a "Vault report" with the following data:
16.0 units of honey
1.9 units of nectar
LOW NECTAR - ADD A NECTAR COLLECTOR
Egg count: 3.9
Unassigned workers: 0.9
1 Nectar Collector bee
2 Honey Manufacturer bees
1 Egg Care bee
TOTAL WORKERS: 4

③ Help the queen grow her hive.

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

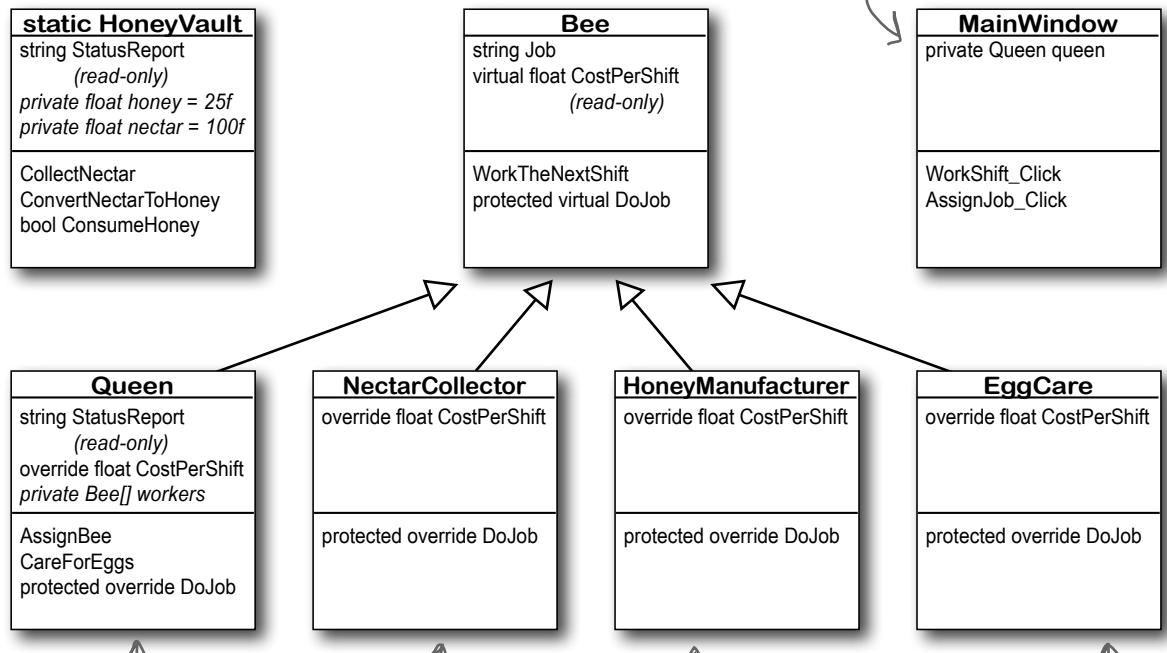
The beehive management system class model

Here are the classes that you'll build for the beehive management system. There's an inheritance model with a base class and four subclasses, a static class to manage the honey and nectar that drive the hive business, and the MainWindow class with the code-behind for the main window.

HoneyVault is a static class that keeps track of the honey and nectar in the hive. Bees use the `ConsumeHoney` method, which checks if there's enough honey to do their jobs, and if so subtracts the amount requested.

Bee is the base class for all of the bee classes. Its `WorkTheNextShift` method calls the Honey Vault's `ConsumeHoney` method, and if it returns true calls `DoJob`.

The code-behind for the main window just does a few things. It creates an instance of Queen, and has Click event handlers for the buttons to call her `WorkTheNextShift` and `AssignBee` methods and display the status report.



This Bee subclass uses an array to keep track of the workers and overrides `DoJob` to call their `WorkTheNextShift` methods.

This Bee subclass overrides `DoJob` to call the `HoneyVault` method to collect nectar.

This Bee subclass overrides `DoJob` to call the `HoneyVault` method to convert nectar to honey.

This Bee subclass keeps a reference to the Queen, and overrides `DoJob` to call the Queen's `CareForEggs` method.



This class model is just the start. We'll give more details so you can write the code.

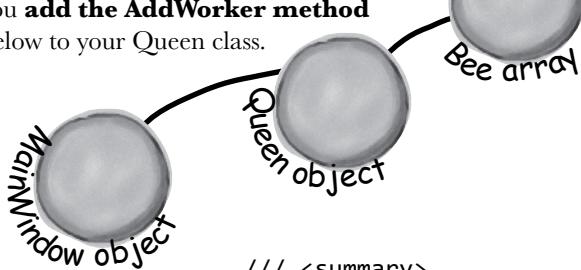
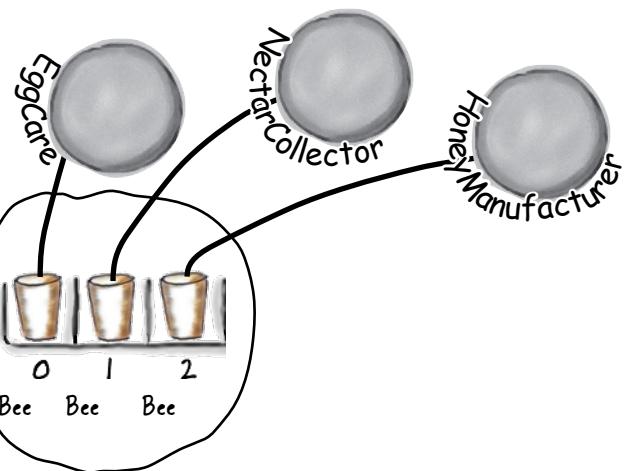
Examine this class model really carefully. It has a lot of information about the app you're about to build. Next, we'll give you all of the details you need to write the code for these classes.

The Queen class: how she manages the worker bees

When you **press the button to work the next shift**, the button's Click event handler calls the Queen object's WorkTheNextShift method, which is inherited from the Bee base class. Here's what happens next:

- ★ Bee.WorkTheNextShift calls HoneyVault.ConsumeHoney(HoneyConsumed), using the CostPerShift property (which each subclass overrides with a different value) to determine how much honey she needs to work.
- ★ Bee.WorkTheNextShift then calls DoJob, which the Queen also overrides.
- ★ Queen.DoJob adds 0.45 eggs to her private eggs field (using a const called EGGS_PER_SHIFT). The EggCare bee will call her CareForEggs method, which decreases eggs and increases unassignedWorkers.
- ★ Then it uses a foreach loop to call each worker's WorkTheNextShift method.
- ★ It consumes honey for each unassigned worker. The HONEY_PER_UNASSIGNED_WORKER const tracks how much each one consumes per shift.
- ★ Finally, it calls its UpdateStatusReport method.

When you **press the button to assign a job** to a bee, the event handler calls the Queen object's AssignBee method, which takes a string with the job name (you'll get that name from jobSelector.text). It uses a **switch** statement to create a new instance of the appropriate Bee subclass and pass it to AddWorker, so make sure you **add the AddWorker method** below to your Queen class.



The length of an Array instance can't be changed during its lifetime. That's why C# has this useful static [Array.Resize method](#). It doesn't actually resize the array. Instead, it creates a new one and copies the contents of the old one into it. Notice how it uses the `ref` keyword—we'll learn more about that later in the book.

You'll need this AddWorker method to add a new worker to the Queen's worker array. It calls `Array.Resize` to expand the array, then adds the new worker Bee to it.

```

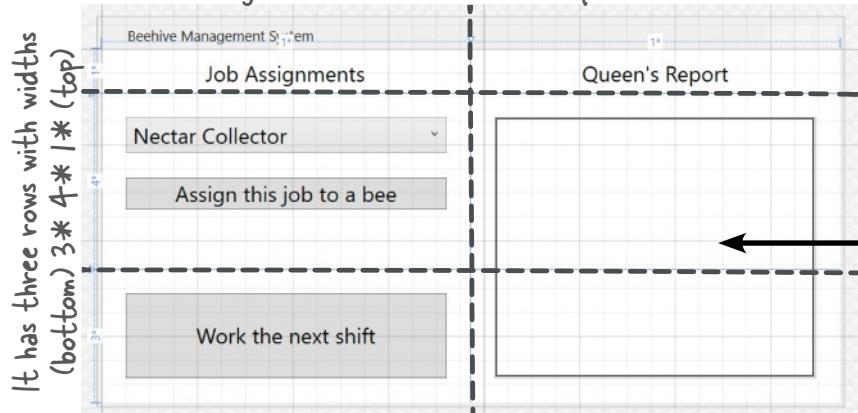
/// <summary>
/// Expand the workers array by one slot and add a Bee reference.
/// </summary>
/// <param name="worker">Worker to add to the workers array.</param>
private void AddWorker(Bee worker)
{
    if (unassignedWorkers >= 1)
    {
        unassignedWorkers--;
        Array.Resize(ref workers, workers.Length + 1);
        workers[workers.Length - 1] = worker;
    }
}

```

The UI: add the XAML for the main window

Create a **new WPF app called BeehiveManagementSystem**. The main window is laid out with a grid, with `Title="Beehive Management System" Height="325" Width="625"`. It uses the same Label, StackPanel, and Button controls you've used in previous chapters, and introduces two new controls. The dropdown list under Job Assignments is a **ComboBox** control, which lets users choose from a list of options. The status report under Queen's Report is displayed in a **TextBox** control.

The grid has two columns with equal widths



This is a **TextBox** control. Normally a **TextBox** is used to get input from the user—but we'll set its `IsReadOnly` property to "True" to make it read-only. We're using it instead of the **TextBlock** you've used in previous projects for two reasons. First, it draws a box around its border, which looks nice. Second, it lets you select and copy text, which is really useful for a status report in a business application.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="1*"/>
        <RowDefinition Height="4*"/>
        <RowDefinition Height="3*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Label Content="Job Assignments" FontSize="18" Margin="20,0"
          HorizontalAlignment="Center" VerticalAlignment="Bottom"/>

    <StackPanel Grid.Row="1" VerticalAlignment="Top" Margin="20">
        <ComboBox x:Name="jobSelector" FontSize="18" SelectedIndex="0" Margin="0,0,0,20">
            <ListBoxItem Content="Nectar Collector"/>
            <ListBoxItem Content="Honey Manufacturer"/>
            <ListBoxItem Content="Egg Care"/>
        </ComboBox>
        <Button Content="Assign this job to a bee" FontSize="18px" Click="AssignJob_Click" />
    </StackPanel>

    <Button Grid.Row="2" Content="Work the next shift" FontSize="18px"
           Click="WorkShift_Click" Margin="20"/>

    <Label Content="Queen's Report" Grid.Column="1" FontSize="18" Margin="20,0"
          VerticalAlignment="Bottom" HorizontalAlignment="Center"/>

    <TextBox
        x:Name="statusReport" IsReadOnly="True"
        Grid.Row="1" Grid.RowSpan="2" Grid.Column="1" Margin="20"/>
</Grid>
```

The dropdown list is a **ComboBox** control. It's a container control (like **Grid**) with controls between its opening and closing tags. In this case, it contains three **ListBoxItem** controls, one for each item the user can select. You can expand Common in the Properties window and use the use the **...** button next to Items to add them (choose **ListBoxItem** from the dropdown), but it's actually easier to just type the items into the XAML code by hand. Make sure the content for each item exactly matches this code.

These **ListBoxItem** controls determine the items displayed to the user in the **ComboBox** list.

Give the **TextBox** a name (`x:Name`) so you can set its `Text` property in the code-behind.

let's get to bee-siness

Don't get overwhelmed or intimidated by the length of this exercise! Just break it down into small steps. Once you start working on it, you'll see it's all review of things you've learned.



Long Exercise

Build the Beehive Management System. The purpose of the system is to **maximize the number of workers assigned to jobs in the hive**, and keep the hive running as long as possible until the honey runs out.

Rules of the hive

Workers can be assigned one of three jobs: nectar collectors add nectar to the honey vault, honey manufacturers convert the nectar into honey, and egg care bees turn eggs into workers who can be assigned to jobs. During each shift, the Queen lays eggs (just under two shifts per egg). The Queen updates her status report at the end of the shift. It shows the honey vault status and the number of eggs, unassigned workers, and bees assigned to each job.

Start by building the static HoneyVault class

- The HoneyVault class is a good starting point because it has no **dependencies**—it doesn't call methods or use properties or fields from any other class. Start by creating a new class called HoneyVault. Make it **static**, then look at the class diagram and add the class members.
- HoneyVault has **two constants** (`NECTAR_CONVERSION_RATIO = .19f` and `LOW_LEVEL_WARNING = 10f`) that are used in the methods. Its private honey field is initialized to `25f` and its private nectar field is initialized to `100f`.
- The **ConvertNectarToHoney method** converts nectar to honey. It takes a float parameter called `amount`, subtracts that amount from its nectar field, and adds `amount × NECTAR_CONVERSION_RATIO` to the honey field. (If the amount passed to the method is less than the nectar left in the vault, it converts all of the remaining nectar.)
- The **ConsumeHoney method** is how the bees use honey to do their jobs. It takes a parameter, `amount`. If it's greater than the honey field it subtracts `amount` from honey and returns true; otherwise it returns false.
- The **CollectNectar method** is called by the NectarCollector bee each shift. It takes a parameter, `amount`. If `amount` is greater than zero, it adds it to the honey field.
- The **StatusReport property** only has a get accessor that returns a string with separate lines with the amount of honey and the amount of nectar in the vault. If the honey is below `LOW_LEVEL_WARNING`, it adds a warning ("LOW HONEY - ADD A HONEY MANUFACTURER"). It does the same for the nectar field.

Create the Bee class and start building the Queen, HoneyManufacturer, NectarCollector, and EggCare classes

- Create the Bee base class. Its **constructor** takes a string, which it uses to set the **read-only Job property**. Each Bee subclass passes a string to the base constructor—"Queen", "Nectar Collector", "Honey Manufacturer", or "Egg Care"—so the Queen class has this code: `public Queen() : base("Queen")`
- The virtual read-only **CostPerShift property** lets each Bee subclass define the amount of honey it consumes each shift. The **WorkTheNextShift method** passes `HoneyConsumed` to the `HoneyVault.ConsumeHoney` method. If `ConsumeHoney` returns true there's enough honey left in the hive, so `WorkTheNextShift` then calls `DoJob`.
- Create empty** HoneyManufacturer, NectarCollector, and EggCare classes that just extend Bee—you'll need them to build the Queen class. You'll **finish the Queen class first**, then come back and finish the other Bee subclasses.
- Each Bee subclass **overrides the DoJob method** with code to do its job, and **overrides the CostPerShift property** with the amount of honey it consumes each shift.
- Here are all of the **values for the read-only Bee.CostPerShift property** for each Bee subclass:
`Queen.CostPerShift` returns `2.15f`, `NectarCollector.CostPerShift` returns `1.95f`, `HoneyManufacturer.CostPerShift` returns `1.7f`, and `EggCare.CostPerShift` returns `1.35f`.

Every single part of this exercise is something you've seen before. You CAN do this!

Long Exercise



This is a long exercise, but that's OK! Just build it class by class. Finish building the Queen class first. Once you're done, you'll go back to the other Bee subclasses.

- The Queen class has a **private Bee[] field** called workers. It starts off as an empty array. We gave you the AddWorker method to add Bee references to it.
- Her **AssignBee method** takes a parameter with a job name (like "Egg Care"). It has `switch (job)` with cases that call AddWorker. For example, if job is "Egg Care" then it calls AddWorker(new EggCare(this)).
- There are two **private float fields** called eggs and unassignedWorkers to keep track of the number of eggs (which she adds to each shift) and the number of workers waiting to be assigned.
- She overrides the **DoJob method** to add eggs, tell the worker bees to work, and feed honey to the unassigned workers waiting for work. The EGGS_PER_SHIFT constant (set to 0.45f) is added to the eggs field. She uses a foreach loop to call each worker's WorkTheNextShift method. Then she calls HoneyVault.ConsumeHoney, passing it the constant HONEY_PER_UNASSIGNED_WORKER (set to 0.5f) × workers.Length.
- She starts off with three unassigned workers—her **constructor** calls the AssignBee method three times to create three worker bees, one of each type.
- The EggCare bees call the Queen's **CareForEggs method**. It takes a float parameter called eggsToConvert. If the eggs field is \geq eggsToConvert, it subtracts eggsToConvert from eggs and adds it to unassignedWorkers.
- Look carefully at the status reports in the screenshot—her private **UpdateStatusReport method** generates it (using HoneyVault.StatusReport). She calls UpdateStatusReport at the end of her DoJob and AssignBee methods.

Finish building the other Bee subclasses

- The **NectarCollector class** has a const NECTAR_COLLECTED_PER_SHIFT = 33.25f. Its **DoJob method** passes that const to HoneyVault.CollectNectar.
- The **HoneyManufacturer class** has a const NECTAR_PROCESSED_PER_SHIFT = 33.15f, and its DoJob method passes that const to HoneyVault.ConvertNectarToHoney.
- The **EggCare class** has a const CARE_PROGRESS_PER_SHIFT = 0.15f, and its DoJob method passes that const to queen.CareForEggs, using a private Queen reference that's **initialized in the EggCare constructor**.

Build the code-behind for the main window

- We gave you the XAML for the **main window**. Your job is to add the code-behind. It has a private Queen field called queen that's initialized in the constructor, and event handlers for the buttons and combo box.
- Hook up the **event handlers**. The "assign job" button calls queen.AssignBee(jobSelector.Text). The "Work the next shift" button calls queen.WorkTheNextShift. They both set statusReport.Text equal to queen.StatusReport.

Some more details about how the Beehive Management System works

- The goal is to get the TOTAL WORKERS line in the status report (which lists the total number of assigned workers) to go as high as possible—and that all depends on **which workers you add and when you add them**. Workers drain honey: if you've got too many of one kind of worker, the honey starts to go down. As you run the program, watch the honey and nectar numbers. After the first few shifts, you'll get a low honey warning (so add a honey manufacturer); after a few more, you'll get a low nectar warning (so add a nectar collector)—after that, you need to figure out how to staff the hive. How high can you get TOTAL WORKERS to go before the honey runs out?



Long Exercise Solution

This project is big, and it has **a lot of different parts**. If you run into trouble, just take it piece by piece. None of it is magic—you already have the tools to understand every part of it.

Here's the code for the **static HoneyVault class**:

```
static class HoneyVault
{
    public const float NECTAR_CONVERSION_RATIO = .19f;
    public const float LOW_LEVEL_WARNING = 10f;
    private static float honey = 25f;
    private static float nectar = 100f;

    public static void CollectNectar(float amount)
    {
        if (amount > 0f) nectar += amount; ← The NectarCollector bees do their jobs by calling the CollectNectar method to add nectar to the hive.
    }

    public static void ConvertNectarToHoney(float amount)
    {
        float nectarToConvert = amount;
        if (nectarToConvert > nectar) nectarToConvert = nectar;
        nectar -= nectarToConvert;
        honey += nectarToConvert * NECTAR_CONVERSION_RATIO; } ← The HoneyManufacturer bees do their jobs by calling ConvertNectarToHoney, which reduces the nectar and increases the honey in the vault.

    public static bool ConsumeHoney(float amount)
    {
        if (honey >= amount)
        {
            honey -= amount;
            return true; ← Every bee tries to consume a specific amount of honey during each shift. The ConsumeHoney method only returns true if there's enough honey for the bee to do its job.
        }
        return false;
    }

    public static string StatusReport
    {
        get
        {
            string status = $"{honey:0.0} units of honey\n" +
                $"{nectar:0.0} units of nectar";
            string warnings = "";
            if (honey < LOW_LEVEL_WARNING) warnings +=
                "\nLOW HONEY - ADD A HONEY MANUFACTURER";
            if (nectar < LOW_LEVEL_WARNING) warnings +=
                "\nLOW NECTAR - ADD A NECTAR COLLECTOR";
            return status + warnings;
        }
    }
}
```

The constants in the HoneyVault class are really important. Try making the nectar conversion ratio bigger—that adds lots of honey to the vault each shift. Try making it smaller—now the honey disappears almost immediately.

} The HoneyManufacturer bees do their jobs by calling ConvertNectarToHoney, which reduces the nectar and increases the honey in the vault.

It's OK if your code doesn't exactly match our code!
There are many different ways that you can solve this problem—and the bigger the program is, the more ways there are to write it. If your code works, then you did the exercise correctly! But take a few minutes to compare your solution with ours, and take the time to try and figure out why we made the decisions that we did.

Try using the View menu to show the Class View in the IDE (it will be docked in the Solution Explorer window). It's a useful tool for exploring your class hierarchy. Try expanding a class in the Class View window, then expand the Base Types folder to see its hierarchy. Use the tabs at the bottom of the window to switch between the Class View and Solution Explorer.



The behavior of this program is driven by the way the different classes interact with each other—especially the ones in the Bee class hierarchy. And at the top of that hierarchy is the **Bee superclass** that all of the other Bee classes extend:

```
class Bee
{
    public virtual float CostPerShift { get; }

    public string Job { get; private set; }

    public Bee(string job) {
        Job = job;
    }

    public void WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
        {
            DoJob();
        }
    }

    protected virtual void DoJob() { /* the subclass overrides this */ }
}
```

The Bee constructor takes a single parameter, which it uses to set its read-only Job property. The Queen uses that property when she generates the status report to figure out what subclass a specific bee is.

The **NectarCollector class** collects nectar each shift and adds it to the vault:

```
class NectarCollector : Bee
{
    public const float NECTAR_COLLECTED_PER_SHIFT = 33.25f;
    public override float CostPerShift { get { return 1.95f; } }
    public NectarCollector() : base("Nectar Collector") { }

    protected override void DoJob()
    {
        HoneyVault.CollectNectar(NECTAR_COLLECTED_PER_SHIFT);
    }
}
```

The NectarCollector and HoneyManufacturer classes have constants that determine how much nectar is collected and how much of it is converted to honey during each shift. Try changing them—the program is a lot less sensitive to changes to these constants than it is when you change the HoneyVault conversion ratio.

The **HoneyManufacturer class** converts the nectar in the honey vault into honey:

```
class HoneyManufacturer : Bee
{
    public const float NECTAR_PROCESSED_PER_SHIFT = 33.15f;
    public override float CostPerShift { get { return 1.7f; } }
    public HoneyManufacturer() : base("Honey Manufacturer") { }

    protected override void DoJob()
    {
        HoneyVault.ConvertNectarToHoney(NECTAR_PROCESSED_PER_SHIFT);
    }
}
```



Long Exercise Solution

Each of the Bee subclasses has a different job, but they have **shared behaviors**—even the Queen. They all work during each shift, but only do their jobs if there's enough honey.

The Queen class manages the workers and generates the status reports:

```
class Queen : Bee
{
    public const float EGGS_PER_SHIFT = 0.45f;
    public const float HONEY_PER_UNASSIGNED_WORKER = 0.5f;

    private Bee[] workers = new Bee[0];
    private float eggs = 0;
    private float unassignedWorkers = 3;

    public string StatusReport { get; private set; }
    public override float CostPerShift { get { return 2.15f; } }

    public Queen() : base("Queen") {
        AssignBee("Nectar Collector");
        AssignBee("Honey Manufacturer");
        AssignBee("Egg Care");
    }

    private void AddWorker(Bee worker) {
        if (unassignedWorkers >= 1)
        {
            unassignedWorkers--;
            Array.Resize(ref workers, workers.Length + 1);
            workers[workers.Length - 1] = worker;
        }
    }

    private void UpdateStatusReport()
    {
        StatusReport = $"Vault report:\n{HoneyVault.StatusReport}\n" +
        $"\\nEgg count: {eggs:0.0}\\nUnassigned workers: {unassignedWorkers:0.0}\\n" +
        $"\\n{WorkerStatus("Nectar Collector")}\\n{WorkerStatus("Honey Manufacturer")}" +
        $"\\n{WorkerStatus("Egg Care")}\\nTOTAL WORKERS: {workers.Length}";
    }

    public void CareForEggs(float eggsToConvert)
    {
        if (eggs >= eggsToConvert)
        {
            eggs -= eggsToConvert;
            unassignedWorkers += eggsToConvert;
        }
    }
}
```

The constants in the Queen class are really important because they determine how the program behaves over the course of many shifts. If she lays too many eggs, they eat more honey, but also speed up progress. If unassigned workers consume more honey, it adds more pressure to assign workers quickly.

The Queen starts things off by assigning one bee of each type in her constructor.

We gave you this AddWorker method. It resizes the array and adds a Bee object to the end. Have you noticed that sometimes the status report lists the unassigned workers as 1.0 but you can't add a worker? Add a breakpoint to the first line of AddWorker—you'll see unassignedWorkers is equal to 0.9999999999.... Can you think of how to fix that?

You had to look really closely at the status report in the screenshot to figure out what to include here.

The EggCare bees call the CareForEggs method to convert eggs into unassigned workers.

Long Exercise Solution



The Queen class drives all of the work in the program—she keeps track of the instances of the worker Bee objects, creates new ones when they need to be assigned to their jobs, and tells them to start working their shifts:

```

private string WorkerStatus(string job)
{
    int count = 0;
    foreach (Bee worker in workers)
        if (worker.Job == job) count++;
    string s = "s";
    if (count == 1) s = "";
    return $"{count} {job} bee{s}";
}

public void AssignBee(string job)
{
    switch (job)
    {
        case "Nectar Collector":
            AddWorker(new NectarCollector());
            break;
        case "Honey Manufacturer":
            AddWorker(new HoneyManufacturer());
            break;
        case "Egg Care":
            AddWorker(new EggCare(this));
            break;
    }
    UpdateStatusReport();
}

protected override void DoJob()
{
    eggs += EGGS_PER_SHIFT;
    foreach (Bee worker in workers)
    {
        worker.WorkTheNextShift();
    }
    HoneyVault.ConsumeHoney(unassignedWorkers * HONEY_PER_UNASSIGNED_WORKER);
    UpdateStatusReport();
}

```

The private WorkerStatus method uses a foreach loop to count the number of bees in the workers array that match a specific job. Notice how it uses the "s" variable to use the plural "bees" unless there's just one bee.

The AssignBee method uses a switch statement to determine which type of worker to add. The strings in the case statements need to match the Content properties of each ListBoxItem in the ComboBox exactly, otherwise none of the cases will match.

The Queen does her job by adding eggs, telling each worker to work the next shift, and then making sure each of the unassigned workers consumes honey. She updates the status report after every bee assignment and shift to make sure it's always up to date.

The Queen is not a micromanager. She lets the worker Bee objects do their jobs and consume their own honey.

That's a good example of separation of concerns: queen-related behavior is encapsulated in the Queen class, and the Bee class contains only the behavior common to all bees.



Long Exercise Solution

The constants at the top of each of the Bee subclasses are really important. We came up with the values for those constants through trial and error: we tweaked one of the numbers, then ran the program to see what effect it had. We tried to come up with a good balance between the classes. Do you think we did a good job? Can you do better? We bet you can!

The EggCare class uses a reference to the Queen object to call her CareForEggs method to turn eggs into workers:

```
class EggCare : Bee
{
    public const float CARE_PROGRESS_PER_SHIFT = 0.15f; ←
    public override float CostPerShift { get { return 1.35f; } }

    private Queen queen;

    public EggCare(Queen queen) : base("Egg Care")
    {
        this.queen = queen;
    }

    protected override void DoJob()
    {
        queen.CareForEggs(CARE_PROGRESS_PER_SHIFT);
    }
}
```

The EggCare bee's constant determines how rapidly the eggs are turned into unassigned workers. More workers can be good for the hive, but they also consume more honey. The challenge is getting the right balance of different worker types.

Here's the code-behind for the main window. It doesn't do much—all of the intelligence is in the other classes:

```
public partial class MainWindow : Window
{
    private Queen queen = new Queen();

    public MainWindow()
    {
        InitializeComponent();
        statusReport.Text = queen.StatusReport;
    }

    private void WorkShift_Click(object sender, RoutedEventArgs e)
    {
        queen.WorkTheNextShift();
        statusReport.Text = queen.StatusReport;
    }

    private void AssignJob_Click(object sender, RoutedEventArgs e)
    {
        queen.AssignBee(jobSelector.Text); ←
        statusReport.Text = queen.StatusReport;
    }
}
```

The code-behind updates the status report TextBox in the constructor after the buttons are clicked to make sure the latest report is always displayed.

The "assign job" button passes the text from the selected ComboBox item directly to Queen.AssignBee, so it's really important that the cases in the switch statement match the ComboBox items exactly.

If you run into trouble writing the code, it's absolutely OK to look at the solution!



Really take a minute and think about this, because it gets to the heart of what dynamics are about. Do you see any way to use some of these ideas in other kinds of programs, and not just games?

HEY, WAIT A MINUTE. THIS...THIS ISN'T A SERIOUS BUSINESS APPLICATION.
IT'S A GAME!

YOU GUYS REALLY SUCK.

OK, you got us. Yes, you're right. This is a game.

Specifically, it's a **resource management game**, or a game where the mechanics are focused on collecting, monitoring, and using resources. If you've played a simulation game like SimCity or strategy game like Civilization, you'll recognize resource management as a big part of the game, where you need resources like money, metal, fuel, wood, or water to run a city or build an empire.

Resource management games are a great way to experiment with the relationship between **mechanics, dynamics, and aesthetics**:

- ★ The **mechanics** are simple: the player assigns workers and then initiates the next shift. Then each bee either adds nectar, reduces nectar/increases honey, or reduces eggs/increases workers. The egg count increases, and the report is displayed.
- ★ The **aesthetics** are more complex. Players feel stress as the honey or nectar levels fall and the low level warning is displayed. They feel excitement when they make a choice, and satisfaction when it affects the game—and then stress again, as the numbers stop increasing and start decreasing again.
- ★ The game is driven by the **dynamics**. There's nothing in the code that makes the honey or nectar scarce—they're just consumed by the bees and eggs.

BRAIN POWER

A small change in HoneyVault.NECTAR_CONVERSION_RATIO can make the game much easier or much harder by making the honey drain slowly or quickly. What other numbers affect gameplay? What do you think is driving those relationships?

Feedback drives your Beehive Management game

Let's take a few minutes and really understand how this game works. The nectar conversion ratio has a big impact on your game. If you change the constants, it can make really big differences in gameplay. If it takes just a little honey to convert an egg to a worker, the game gets really easy. If it takes a lot, the game gets much harder. But if you go through the classes, you won't find a difficulty setting. There's no Difficulty field on any of them. Your Queen doesn't get special power-ups to help make the game easier, or tough enemies or boss battles to make it more difficult. In other words, there's **no code that explicitly creates a relationship** between the number of eggs or workers and the difficulty of the game. So what's going on?

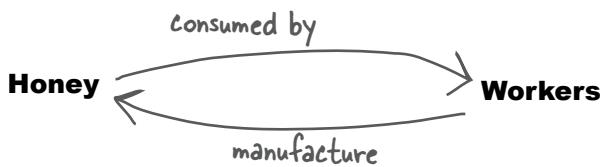
When you point a camera at a screen displaying its video output, you create a feedback loop that can cause these weird patterns.



You've probably played with **feedback** before. Start a video call between your phone and your computer. Hold the phone near the computer speaker and you'll hear noisy echoes. Point the camera at the computer screen and you'll see a picture of the screen inside the picture of the screen inside the picture of the screen, and it will turn into a crazy pattern if you tilt the phone. This is feedback: you're taking the live video or audio output and *feeding it right back* into the input. There's nothing in the code of the video call app that specifically generates those crazy sounds or images. Instead, they **emerge** from the feedback.

Workers and honey are in a feedback loop

Your Beehive Management game is based on a series of **feedback loops**: lots of little cycles where parts of the game interact with each other. For example, honey manufacturers add honey to the vault, which is consumed by honey manufacturers, who make more honey.

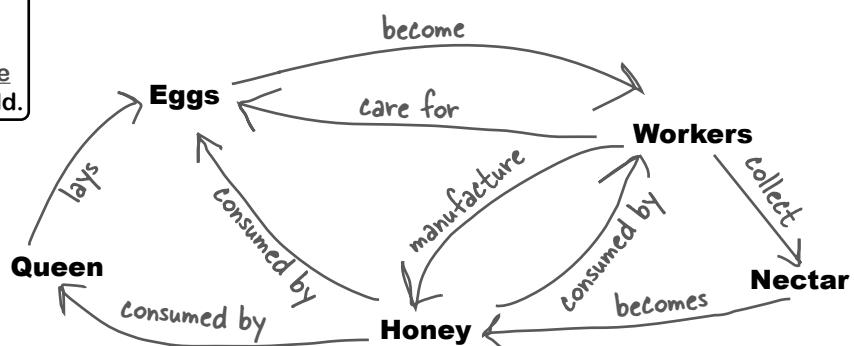


The feedback loop between the workers and honey is just one small part of the whole system that drives the game. See if you can spot it in the bigger picture below.

And that's just one feedback loop. There are many different feedback loops in your game, and they make the whole game more complex, more interesting, and (hopefully!) more fun.

A series of feedback loops drive the dynamics of your game. The code you build won't explicitly manage these feedback loops. They **emerge** out of the mechanics that you'll build.

And this same concept is actually really important in a lot of real-world business applications, not just games. Everything you're learning here, you can use on the job as a professional software developer.



Mechanics, Aesthetics, and Dynamics Up Close



Feedback loops...equilibrium...making your code do something indirectly by creating a system...does all this have your head spinning a little? Here's another opportunity to *use game design to explore a larger programming concept*.

You've learned about mechanics, dynamics, and aesthetics—now it's time to bring them all together. The **Mechanics-Dynamics-Aesthetics framework**, or **MDA framework**, is a formal tool ("formal" just means it's written down) that's used by researchers and academics to analyze and understand games. It defines the relationship between mechanics, dynamics, and aesthetics, and gives us a way to talk about how they create feedback loops to influence each other.

The MDA framework was developed by Robin Hunicke, Marc LeBlanc, and Robert Zubek, and published in a 2004 paper called "MDA: A Formal Approach to Game Design and Game Research"—it's quite readable, without a ton of academic jargon. (Remember back in Chapter 5, when we talked about how aesthetics includes challenge, narrative, sensation, fantasy, and expression? That came from this paper.) Take a few minutes and look through it, it's actually a great read: <http://bit.ly/mda-paper>.

The goal of the MDA framework is to give us a formal way to think about and analyze video games. This may sound like something that's only important in an academic setting, like a college course on game design. But it's actually really valuable to us as everyday game developers, because it can help us understand how people perceive the games we create, and give us a deeper insight into **what makes those games fun**.

Game designers had been using the terms mechanics, dynamics, and aesthetics informally, but the paper really gave them a solid definition, and established the relationship between them.



One thing that the MDA framework tackles is the **difference in perspective** between gamers and game designers. Players, first and foremost, want the game to be fun—but we've already seen how "fun" can differ wildly from player to player. Game designers, on the other hand, typically see a game through the lens of its mechanics, because they spend their time writing code, designing levels, creating graphics, and tinkering with the mechanical aspects of the game.

All developers (not just game developers!) can use the MDA framework to get a handle on feedback loops

Let's use the MDA framework to analyze a classic game, Space Invaders, so we can better understand feedback loops.

- Start with mechanics of the game: the player's ship moves left and right and fires shots up; the invaders march in formation and fire shots down; the shields block shots. The fewer enemies there are on screen, the faster they go.
- Players figure out strategies: shoot where the invaders will be, pick off enemies on the sides of the formation, hide behind the shields. The code for the game doesn't have an `if/else` or `switch` statement for these strategies; they emerge as the player figures out the game. Players learn the rules, then start to understand the system, which helps them better take advantage of the rules. In other words, ***the mechanics and dynamics form a feedback loop***.
- The invaders get faster, the marching sounds speed up, and the player gets a rush of adrenaline. The game gets more exciting—and in turn, the player has to make decisions more quickly, makes mistakes, and changes strategy, which has an effect on the system. ***The dynamics and aesthetics form another feedback loop***.
- None of this happened by accident. The speed of the invaders, the rate at which they increase, the sounds, the graphics...these were all carefully balanced by the game's creator, Tomohiro Nishikado, who spent over a year designing it, drawing inspiration from earlier games, H. G. Wells, even his own dreams to create a classic game.

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

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

1 Add a using line to the top of your MainWindow.xaml.cs file.

We'll be using a DispatcherTimer to force the game to work the next shift every second and a half. DispatcherTimer is in the System.Windows.Threading namespace, so you'll need to add this **using** line to the top of your *MainWindow.xaml.cs* file:

```
using System.Windows.Threading;
```

You used a DispatcherTimer in Chapter 1 to add a timer to your animal matching game. This code is very similar to the code you used in Chapter 1. Take a few minutes and flip back to that project to remind yourself how the DispatcherTimer works.

2 Add a private field with a DispatcherTimer reference.

Now you'll need to create a new DispatcherTimer. Put it in a private field at the top of the MainWindow class:

```
private DispatcherTimer timer = new DispatcherTimer();
```

3 Make the timer call the WorkShift button's Click event handler method.

We want the timer to keep the game moving forward, so if the player doesn't click the button quickly enough it will automatically trigger the next shift. Start by adding this code:

```
public MainWindow()
{
    InitializeComponent();
    statusReport.Text = queen.StatusReport;
    timer.Tick += Timer_Tick;
    timer.Interval = TimeSpan.FromSeconds(1.5);
    timer.Start();
}

private void Timer_Tick(object sender, EventArgs e)
{
    WorkShift_Click(this, new RoutedEventArgs());
```

As soon as you type `+=` Visual Studio prompts you to create the `Timer_Tick` event handler. Press Tab to have the IDE create the method for you.

The Timer calls the `Tick` event handler every 1.5 seconds, which in turn calls the `WorkShift` button's event handler.

Now run your game. A new shift starts every 1.5 seconds, whether or not you click the button. This is a small change to the mechanics, but it **dramatically changes the dynamics of the game**, which leads to a huge difference in aesthetics. It's up to you to decide if the game is better as a turn-based or real-time simulation.



IT TOOK JUST A FEW LINES OF CODE TO ADD THE TIMER, BUT THAT COMPLETELY CHANGED THE GAME. IS THAT BECAUSE IT HAD A BIG IMPACT ON THE RELATIONSHIP BETWEEN MECHANICS, DYNAMICS, AND AESTHETICS?

Yes! The timer changed the mechanics, which altered the dynamics, which in turn impacted the aesthetics.

Let's take a minute and think about that feedback loop. The change in mechanics (a timer that automatically clicks the "Work the next shift" button every 1.5 seconds) creates a totally new dynamic: a window when players must make decisions, or else the game makes the decision for them. That increases the pressure, which gives some players a satisfying shot of adrenaline, but just causes stress in other players—the aesthetics changed, which makes the game more fun for some people but less fun for others.

But you only added half a dozen lines of code to your game, and none of them included "make this decision or else" logic. That's an example of behavior that **emerged** from how the timer and the button work together.

There's a feedback loop here, too. As players feels more stress, they make worse decisions, changing the game... aesthetics feeds back into mechanics.

THIS WHOLE DISCUSSION OF FEEDBACK LOOPS SEEMS PRETTY IMPORTANT—ESPECIALLY THE PART ABOUT HOW BEHAVIOR EMERGES.

Feedback loops and emergence are important programming concepts.

Try experimenting with these feedback loops. Add more eggs per shift or start the hive with more honey, for example, and the game gets easier. Go ahead, give it a try! You can change the entire feel of the game just by making small changes to a few constants.

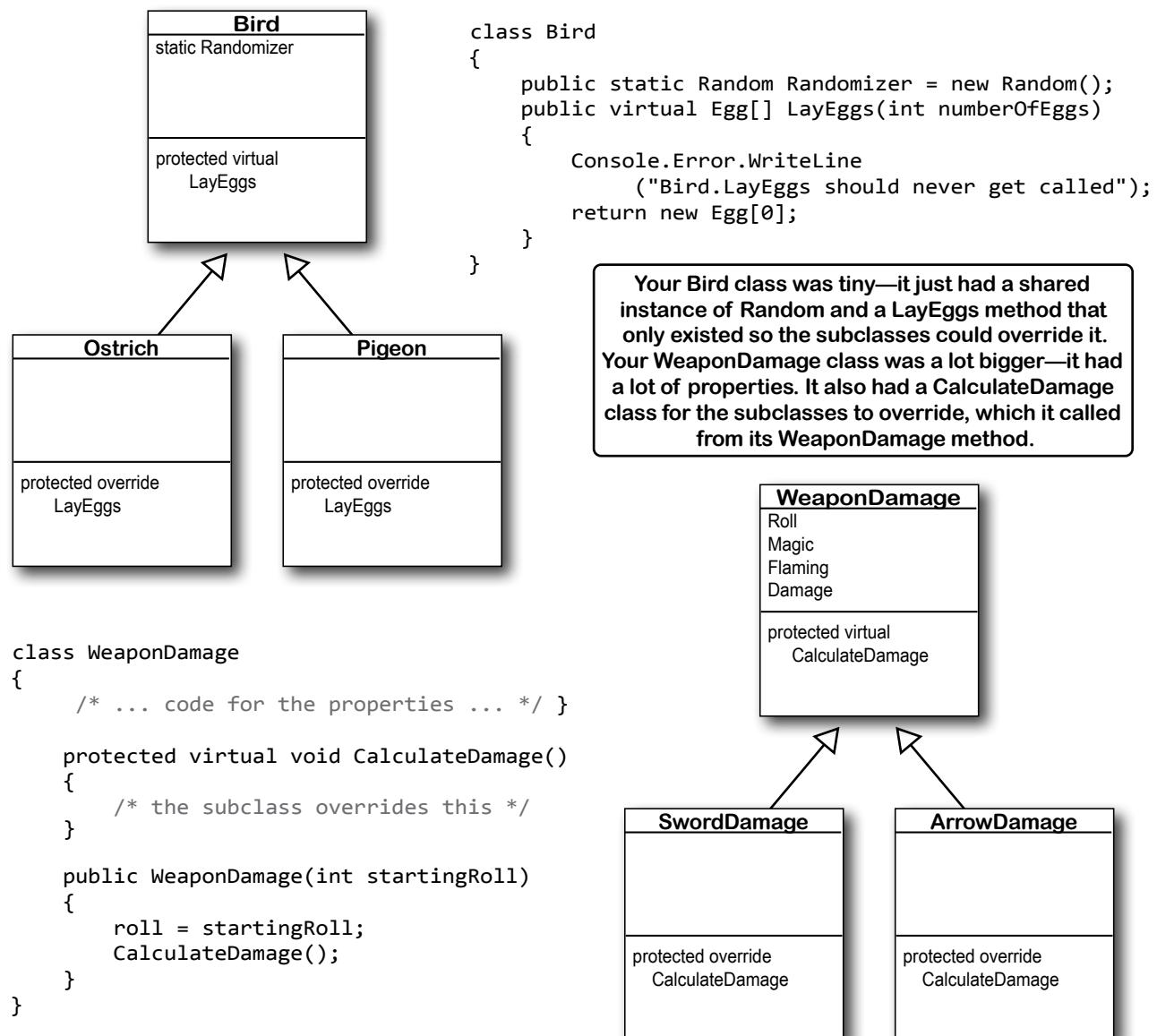
We designed this project to give you practice with inheritance, but *also* to let you explore and experiment with **emergent** behavior. That's behavior that comes not just from what your objects do individually, but also out of **the way objects interact with each other**. The constants in the game (like the nectar conversion ratio) are an important part of that emergent interaction. When we created this exercise, we started out by setting those constants to some initial values, then we tweaked them by making tiny adjustments until we ended up with a system that's not quite in **equilibrium**—a state where everything is perfectly balanced—so the player needs to keep making decisions in order to make the game last as long as possible. That's all driven by the feedback loops between the eggs, workers, nectar, honey, and queen.



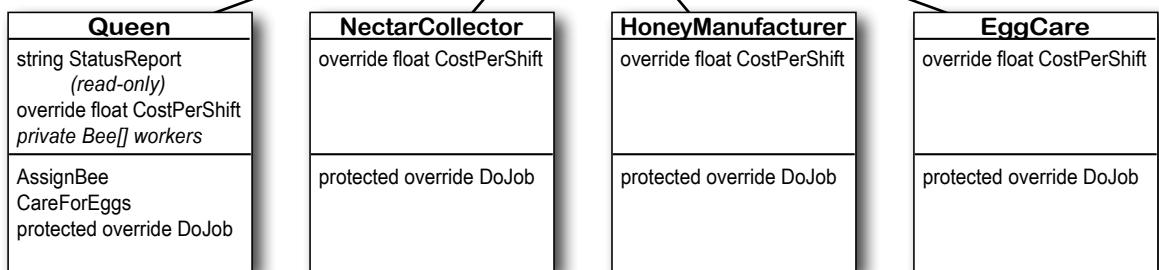
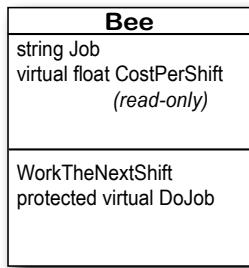
Some classes should never be instantiated

Remember our zoo simulator class hierarchy? You'll definitely end up instantiating a bunch of Hippos, Dogs, and Lions. What about the Canine and Feline classes? How about the Animal class? It turns out that **there are some classes that just don't need to be instantiated**...and, in fact, *don't make any sense* if they are.

Does that sound weird? Actually, it happens all the time—in fact, you created several classes earlier in the chapter that should never be instantiated.



The Bee class isn't instantiated anywhere in the Beehive Management System code. It's not clear what would happen if you tried to instantiate it, since it never sets its cost per shift.



```

class Bee
{
    public virtual float CostPerShift { get; }

    public string Job { get; private set; }

    public Bee(string job)
    {
        Job = job;
    }

    public void WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
        {
            DoJob();
        }
    }

    protected virtual void DoJob() { /* the subclass overrides this */ }
}
  
```

The Bee class had a WorkTheNextShift method that consumed honey and then did whatever job the bee was supposed to do—so it expected the subclass to override the DoJob method to actually do the job.



So what happens when you instantiate the Bird, WeaponDamage, or Bee classes? Does it ever make sense to do it? Do all of their methods even work?

An abstract class is an intentionally incomplete class

It's really common to have a class with "placeholder" members that it expects the subclasses to implement. It could be at the top of the hierarchy (like your Bee, WeaponDamage, or Bird classes) or in the middle (like Feline or Canine in the zoo simulator class model). They take advantage of the fact that C# always calls the most specific method, like how WeaponDamage calls the CalculateDamage method that's only implemented in SwordDamage or ArrowDamage, or how Bee.WorkTheNextShift depends on the subclasses to implement the DoJob method.

C# has a tool that's built specifically for this: an **abstract class**. It's a class that's intentionally incomplete, with empty class members that serve as placeholders for the subclasses to implement. To make a class abstract, **add the abstract keyword to the class declaration**. Here's what you need to know about abstract classes.



An abstract class works just like a normal class.

You define an abstract class just like a normal one. It has fields and methods, and it can inherit from other classes, too, exactly like a normal class. There's almost nothing new to learn.



An abstract class can have incomplete "placeholder" members.

An abstract class can include declarations of properties and methods that must be implemented by inheriting classes. A method that has a declaration but no statements or method body is called an **abstract method**, and a property that only declares its accessors but doesn't define them is called an **abstract property**. Subclasses that extend it must implement all abstract methods and properties unless they're also abstract.



Only abstract classes can have abstract members.

If you put an abstract method or property into a class, then you'll have to mark that class abstract or your code won't compile. You'll learn more about how to mark a class abstract in a minute.



An abstract class can't be instantiated.

The opposite of abstract is **concrete**. A concrete method is one that has a body, and all the classes you've been working with so far are concrete classes. The biggest difference between an **abstract** class and a **concrete** class is that you can't use `new` to create an instance of an abstract class. If you do, C# will give you an error when you try to compile your code.

Try it now! **Create a new console app**, add an empty abstract class, and try to instantiate it:

```
abstract class MyAbstractClass { }

class Program
{
    MyAbstractClass myInstance = new MyAbstractClass();
}
```

The compiler will give you an error, and won't let you build your code:

CS0144 Cannot create an instance of the abstract class or interface 'MyAbstractClass'

The compiler won't let you instantiate an abstract class because abstract classes are not meant to be instantiated.





WAIT, WHAT? A CLASS THAT I
CAN'T INstantiate? WHY WOULD I EVEN WANT
SOMETHING LIKE THAT?

Because you want to provide some of the code, but still require that subclasses fill in the rest of the code.

Sometimes **bad things happen** when you create objects that should never be instantiated. The class at the top of your class diagram usually has some fields that it expects its subclasses to set. An Animal class may have a calculation that depends on a Boolean called HasTail or Vertebrate, but there's no way for it to set that itself. **Here's a quick example of a class that's problematic when instantiated...**

```
class PlanetMission
{
    protected float fuelPerKm;
    protected long kmPerHour;
    protected long kmToPlanet;

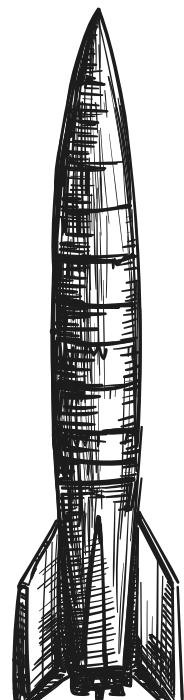
    public string MissionInfo()
    {
        long fuel = (long)(kmToPlanet * fuelPerKm);
        long time = kmToPlanet / kmPerHour;
        return $"We'll burn {fuel} units of fuel in {time} hours";
    }
}

class Mars : PlanetMission
{
    public Mars()
    {
        kmToPlanet = 92000000;
        fuelPerKm = 1.73f;
        kmPerHour = 37000;
    }
}

class Venus : PlanetMission
{
    public Venus()
    {
        kmToPlanet = 41000000;
        fuelPerKm = 2.11f;
        kmPerHour = 29500;
    }
}

class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine(new Venus().MissionInfo());
        Console.WriteLine(new Mars().MissionInfo());
        Console.WriteLine(new PlanetMission().MissionInfo());
    }
}
```

Do this!



Before you run this code, can you figure out what it will print to the console?

abstract classes can help avoid this exception

Like we said, some classes should never be instantiated

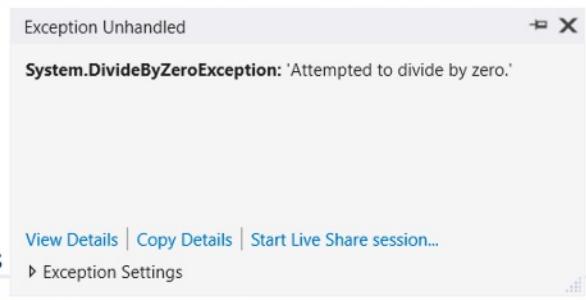
Try running the PlanetMission console app. Did it do what you expected? It printed two lines to the console:

```
We'll burn 86509992 units of fuel in 1389 hours  
We'll burn 159160000 units of fuel in 2486 hours
```

But then it threw an exception.

The problems all started when you created an instance of the PlanetMission class. Its FuelNeeded method expects the fields to be set by the subclass. When they aren't, they get their default values—zero. When C# tries to divide a number by zero...

```
class PlanetMission  
{  
    protected float fuelPerKm;  
    protected long kmPerHour;  
    protected long kmToPlanet;  
  
    public string MissionInfo()  
    {  
        long fuel = (long)(kmToPlanet * fuelPerKm);  
        long time = kmToPlanet / kmPerHour; ✖  
        return $"We'll burn {fuel} units of fuel in {time} hours";  
    }  
}
```



Solution: use an abstract class

When you mark a class **abstract**, C# won't let you write code to instantiate it. So how does that fix this problem? It's like the old saying goes—prevention is better than cure. Add the **abstract** keyword to the PlanetMission class declaration:

```
abstract class PlanetMission  
{  
    // The rest of the class stays the same  
}
```

As soon as you make the change, the compiler gives you an error:

```
✖ CS0144 Cannot create an instance of the abstract class or interface 'PlanetMission'
```

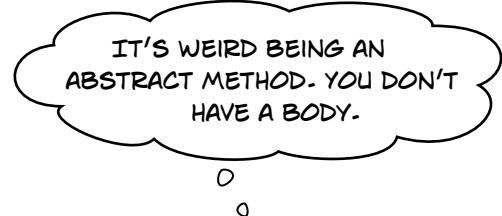
Your code won't compile at all—and no compiled code means no exception. This is really similar to the way you used the **private** keyword in Chapter 5, or **virtual** and **override** earlier in this chapter. Making some members private doesn't change the behavior. It just prevents your code from building if you break the encapsulation. The **abstract** keyword works the same way: you'll never get an exception instantiating an abstract class because the C# compiler *won't let you instantiate one in the first place*.

When you add the abstract keyword to a class declaration, the compiler gives you an error any time you try to create an instance of that class.

An abstract method doesn't have a body

The Bird class that you built was never meant to be instantiated. That's why it uses `Console.Error` to write an error message if a program tries to instantiate it and call its `LayEggs` method:

```
class Bird
{
    public static Random Randomizer = new Random();
    public virtual Egg[] LayEggs(int numberOfEggs)
    {
        Console.Error.WriteLine
            ("Bird.LayEggs should never get called");
        return new Egg[0];
    }
}
```



Since we don't ever want to instantiate the `Bird` class, let's add the `abstract` keyword to its declaration. But that's not enough—not only should this class never be instantiated, but we want to **require** that every subclass that extends `Bird` **must override the `LayEggs` method**.

And that's exactly what happens when you add the `abstract` keyword to a class member. An **abstract method** only has a class declaration but **no method body** that must be implemented by any subclass that extends the abstract class. The **body** of a method is the code between the curly braces that comes after the declaration—and it's something abstract methods can't have.

Go back to your `Bird` project from earlier and **replace the `Bird` class** with this abstract class:

```
abstract class Bird
{
    public static Random Randomizer = new Random();
    public abstract Egg[] LayEggs(int numberOfEggs);
}
```



Your program still runs exactly like it did before! But try adding this line to the `Main` method:

```
Bird abstractBird = new Bird();
```

and you'll get a compiler error:

CS0144 Cannot create an instance of the abstract class or interface 'Bird'

Try to add a body to the `LayEggs` method:

```
public abstract Egg[] LayEggs(int numberOfEggs)
{
    return new Egg[0];
}
```

and you'll get a different compiler error:

CS0500 'Bird.LayEggs(int)' cannot declare a body because it is marked abstract

If an abstract class has virtual members, every subclass must override all of those members.

properties can be abstract

Abstract properties work just like abstract methods

Let's go back to the Bee class from our earlier example. We already know that we don't want the class to be instantiated, so let's modify it to turn it into an abstract class. We can do that just by adding the **abstract** modifier to the class declaration, and changing the DoJob method to an abstract method without a body:

```
abstract class Bee
{
    /* the rest of the class stays the same */
    protected abstract void DoJob();
}
```

But there's one other virtual member—and it's not a method. It's the CostPerShift property, which the Bee.WorkTheNextShift method calls to figure out how much honey the bee will require this shift:

```
public virtual float CostPerShift { get; }
```

We learned in Chapter 5 that properties are really just methods that are called like fields. Use the **abstract keyword to create an abstract property** just like you do with a method:

```
public abstract float CostPerShift { get; }
```

Abstract properties can have a get accessor, a set accessor, or both get and set accessors. Setters and getters in abstract properties **can't have method bodies**. Their declarations look just like automatic properties—but they're not, because they don't have any implementation at all. Like abstract methods, abstract properties are placeholders for properties that must be implemented by any subclass that extends their class.

Here's the whole abstract Bee class, complete with abstract method and property:

```
abstract class Bee
{
    public abstract float CostPerShift { get; }
    public string Job { get; private set; }

    public Bee(string job)
    {
        Job = job;
    }

    public void WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
        {
            DoJob();
        }
    }

    protected abstract void DoJob();
}
```

Replace
this!

Replace the Bee class in your Beehive Management System app with this new abstract one. It will still work! But now if you try to instantiate the Bee class with `new Bee()`; you'll get a compiler error. Even more importantly, **you'll get an error if you extend Bee but forget to implement CostPerShift**.



Exercise

It's time to get some practice with abstract classes—and you don't have to look far to find good candidates for classes to make abstract.

Earlier in the chapter you modified your SwordDamage and ArrowDamage classes to extend a new class called WeaponDamage. Make the WeaponDamage class abstract. There's a good candidate for an abstract method in WeaponDamage—make that abstract as well.

there are no Dumb Questions

Q: When I mark a class abstract, does that change the way it behaves? Do the methods or properties work differently than they do in a concrete class?

A: No, abstract classes work exactly like any other kind of class. When you add the `abstract` keyword to the class declaration, it causes the C# compiler to do two things: prevent you from using the class in a new statement, and allow you to include abstract members.

Q: Some of the abstract classes you showed me are public, others are protected. Does that make a difference? And does the order of those keywords in the class declaration matter?

A: Abstract methods can have any access modifier. If you make an abstract method private, then the classes that implement that abstract method also need to make it private. The keyword order doesn't matter. `protected abstract void DoJob();` and `abstract protected void DoJob();` do exactly the same thing.

Q: I'm confused about the way you're using the word "implement" or "implementation." What do you mean when you're talking about implementing an abstract method?

A: When you use the `abstract` keyword to declare an abstract method or property, we say that you're **defining** the abstract member. Later on, when you add a complete method or property with the same declaration to a concrete class, we say that you're **implementing** the member. So you define abstract methods or properties in an abstract class, and implement them in concrete classes that extend it.

Q: I'm still having trouble with the idea that the `abstract` keyword keeps my code from compiling if I try to instantiate an instance of an abstract class. I already have trouble finding and fixing all of the compiler errors. Why do I want to make it even harder to get my code to build?

A: Sometimes when you're first learning to code, those "CS" compiler errors can be a little frustrating. Everyone has spent time trying to track down a missing comma, period, or quotation mark to try to clear out the Errors List. So why would you ever use a keyword like `abstract` or `private` that puts even more restrictions on your code and makes those compiler errors even more common? It seems a little counterintuitive. If you never use the `abstract` keyword, you'll never see a "Cannot create an instance of the abstract class" compiler error. So why ever use it?

The reason you use keywords like `abstract` or `private` that keep your code from building in certain cases is that it's a lot easier to fix a "Cannot create an instance of the abstract class" compiler error than it is to track down the error that it prevents. If you have a class that should never be instantiated, it's because the bug you get when you create an instance of it instead of a subclass can be subtle and difficult to find. Adding `abstract` to the base class causes your code to **fail fast** with an error that's easier to fix.

Bugs caused by instantiating a base class that should never be instantiated can be subtle and hard to find. Making it abstract makes your code fail fast if you try to create an instance of it.



Exercise Solution

THANKS FOR REFACTORING THIS CLASS! I BET YOU PREVENTED SOME ANNOYING BUGS IN THE FUTURE. NOW I CAN THINK ABOUT MY GAME, AND NOT CODE. GREAT JOB!

The WeaponDamage class should never be instantiated—the only reason it exists is so that the SwordDamage and ArrowDamage classes can inherit its properties and methods. So it makes sense to mark this class abstract. Have a look at its CalculateDamage method:

```
protected virtual void CalculateDamage() {  
    /* the subclass overrides this */  
}
```

This method is a great candidate to convert to an abstract class, because it only exists so that subclasses will override it with their own implementations that update the Damage property. Here's everything that you needed to change in the WeaponDamage class:

```
abstract class WeaponDamage  
{  
    /* the Damage, Roll, Flaming, and Magic properties  
     * stay the same */  
  
    protected abstract void CalculateDamage();  
  
    public WeaponDamage(int startingRoll)  
    {  
        roll = startingRoll;  
        CalculateDamage();  
    }  
}
```



Was this the first time you've read through the code you wrote for previous exercises?

It may feel a little weird to go back to code you wrote before—but that's actually something a lot of developers do, and it's a habit you should get used to. Did you find things that you would do differently the second time around? Are there improvements or changes that you might make? It's always a good idea to take the time to refactor your code. That's exactly what you did in this exercise: you changed the structure of the code without modifying its behavior. **That's refactoring.**



INHERITANCE IS REALLY USEFUL. I CAN DEFINE A METHOD ONCE IN A BASE CLASS, AND IT AUTOMATICALLY APPEARS IN EACH SUBCLASS. WHAT IF I WANT TO DO THAT FOR METHODS IN TWO DIFFERENT CLASSES? IS THERE A WAY FOR ONE SUBCLASS TO EXTEND TWO BASE CLASSES?

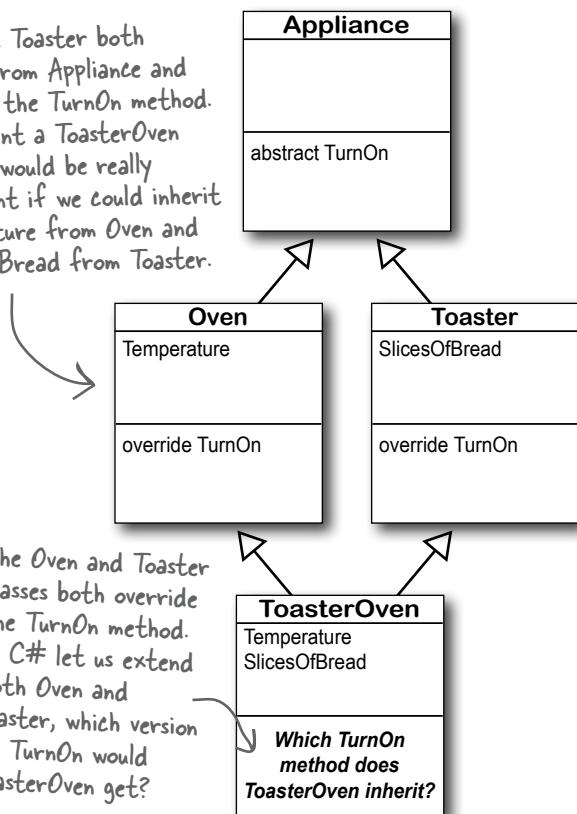
That sounds great! But there's a problem.

If C# let you inherit from more than one base class, it would open up a whole can of worms. When a language lets one subclass inherit from two base classes, it's called **multiple inheritance**. If C# supported multiple inheritance, you would end up in a big fat class conundrum called...

That's its real name!
Some developers just call it the "diamond problem."

The Deadly Diamond of Death

Oven and Toaster both inherit from Appliance and override the TurnOn method. If we want a ToasterOven class, it would be really convenient if we could inherit Temperature from Oven and SlicesOfBread from Toaster.



What would happen in a CRAZY world where C# allowed multiple inheritance? Let's play a little game of "what if" and find out.

What if...you had a class called Appliance that had an abstract method called TurnOn?

And what if...it had two subclasses: Oven with a Temperature property, and Toaster with a SlicesOfBread property?

And what if...you wanted to create a ToasterOven class that inherited both Temperature and SlicesOfBread?

And what if...C# supported multiple inheritance, so you could do that?

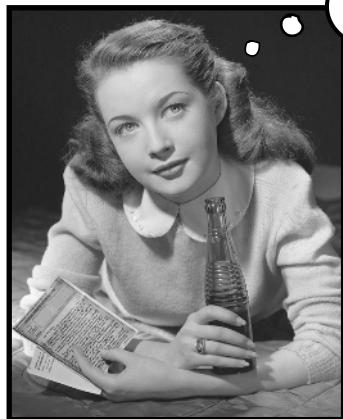
Then there's only one more question...

Which TurnOn does ToasterOven inherit?

Does it get the version from Oven? Or does it get the version from Toaster?

There's no way to know!

And that's why C# doesn't allow multiple inheritance.



WOULDN'T IT BE DREAMY IF THERE WERE SOMETHING LIKE AN ABSTRACT CLASS, BUT THAT GETS AROUND THE DIAMOND PROBLEM SO THAT C# CAN EXTEND MORE THAN ONE OF THEM AT A TIME?

o o

BUT IT'S PROBABLY NOTHING BUT A FANTASY...

BULLET POINTS

- A subclass can override members it inherits, replacing them with new methods or properties with the same names.
- To override a method or property, add the **virtual keyword** to the base class, then add the **override keyword** to the member with the same name in the subclass.
- The **protected keyword** is an access modifier that makes a member public only to its subclasses, but private to every other class.
- When a subclass overrides a method in its base class, the **more specific version** defined in the subclass is always called—even if the base class is calling it.
- If a subclass just adds a method with the same name as a method in its base class, it only **hides** the base class method instead of overriding it. Use the **new keyword** when you're hiding methods.
- The **dynamics** of a game describe how the mechanics combine and cooperate to drive the gameplay.
- A subclass can access its base class using the **base keyword**. When a base class has a constructor, your subclass needs to use the **base keyword** to call it.
- A subclass and base class can have **different constructors**. The subclass can choose what values to pass to the base class constructor.
- Build your **class model on paper** before you write code to help you understand and solve your problem.
- When your classes overlap as little as possible, that's an important design principle called **separation of concerns**.
- **Emergent behavior** occurs when objects interact with each other, beyond the logic directly coded into them.
- **Abstract classes** are intentionally incomplete classes that cannot be instantiated.
- Adding the **abstract keyword** to a method or property and leaving out the body makes it abstract. Any concrete subclass of the abstract class must implement it.
- **Refactoring** means reading code you already wrote and making improvements without modifying its behavior.
- C# does not allow for multiple inheritance because of the **diamond problem**: it can't determine which version of a member inherited from two base classes to use.

Unity Lab #3

GameObject Instances

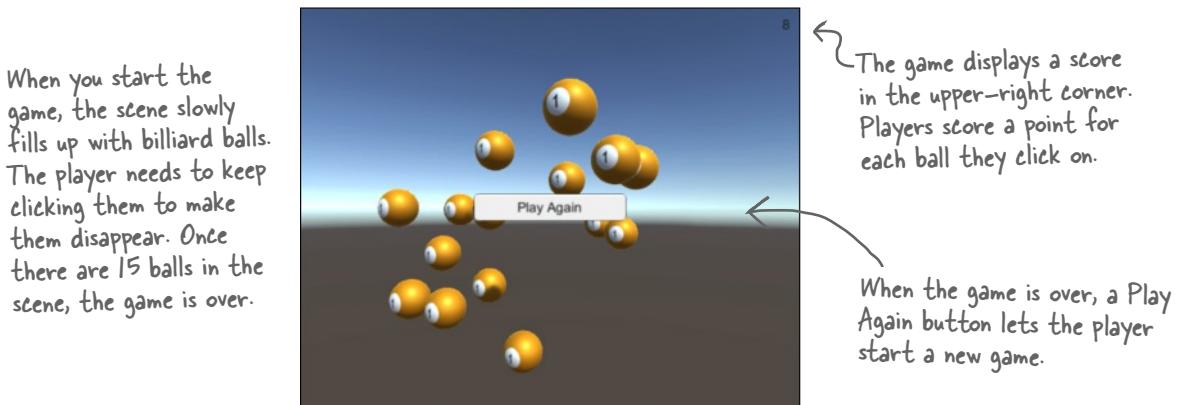
C# is an object-oriented language, and since these Head First C# Unity Labs are all **about getting practice writing C# code**, it makes sense that these labs will focus on creating objects.

You've been creating objects in C# since you learned about the **new** keyword in Chapter 3. In this Unity Lab you'll **create instances of a Unity GameObject** and use them in a complete, working game. This is a great jumping-off point for writing Unity games in C#.

The goal of the next two Unity Labs is to **create a simple game** using the familiar billiard ball from the last lab. In this lab, you'll build on what you learned about C# objects and instances to start building the game. You'll use a **prefab**—Unity's tool for creating instances of GameObjects—to create lots of instances of a GameObject, and you'll use scripts to make your GameObjects fly around your game's 3D space.

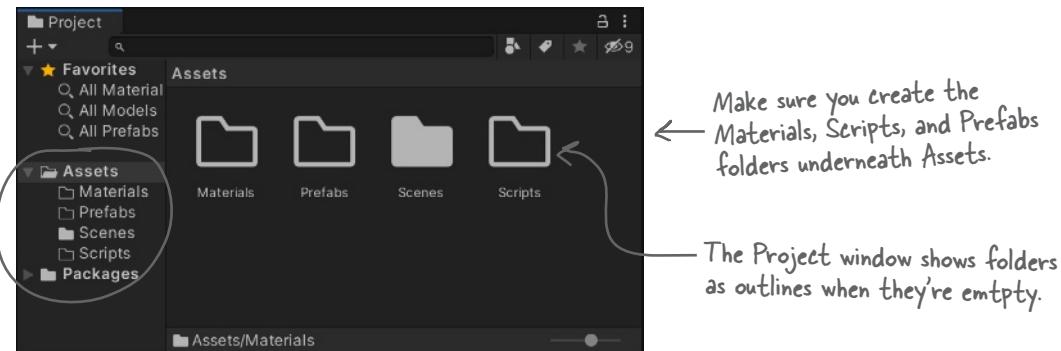
Let's build a game in Unity!

Unity is all about building games. So in the next two Unity Labs, you'll use what you've learned about C# to build a simple game. Here's the game that you're going to create:



So let's get started. The first thing you'll do is get your Unity project set up. This time we'll keep the files a little more organized, so you'll create separate folders for your materials and scripts—and one more folder for prefabs (which you'll learn about later in the lab):

1. Before you begin, close any Unity project that you have open. Also close Visual Studio—you'll let Unity open it for you.
2. **Create a new Unity project** using the 3D template, just like you did for the previous Unity Labs. Give it a name to help you remember which labs it goes with (“Unity Labs 3 and 4”).
3. Choose the Wide layout so your screen matches the screenshots.
4. Create a folder for your materials underneath the Assets folder. **Right-click on the Assets folder** in the Project window and choose Create >> Folder. Name it **Materials**.
5. Create another folder under Assets named **Scripts**.
6. Create one more folder under Assets named **Prefabs**.



Unity Lab #3

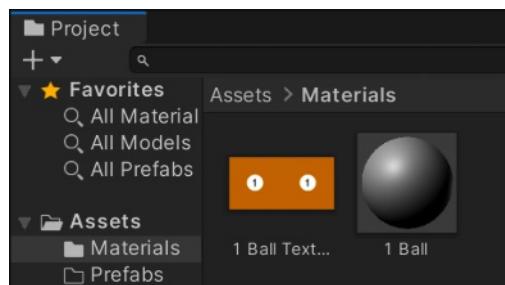
GameObject Instances

Create a new material inside the Materials folder

Double-click on your new Materials folder to open it. You'll create a new material here.

Go to <https://github.com/head-first-csharp/fourth-edition> and click on the Billiard Ball Textures link (just like you did in the first Unity lab) and download the texture file **1 Ball Texture.png** into a folder on your computer, then drag it into your Materials folder—just like you did with the downloaded file in the first Unity Lab, except this time drag it into the Materials folder you just created instead of the parent Assets folder.

Now you can create the new material. Right-click on the Materials folder in the Project window and **choose Create >> Material**. Name your new material **1 Ball**. You should see it appear in the Materials folder in the Project window.



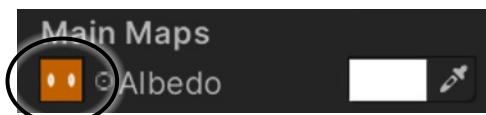
In the previous Unity Labs we used a **texture**, or a bitmap image file that Unity can wrap around GameObjects. When you dragged the texture onto a sphere, Unity automatically created a **material**, which is what Unity uses to keep track of information about how a GameObject should be rendered that can have a reference to a texture. This time you're creating the material manually. Just like last time, you may need to click the Download button on the GitHub page to download the texture PNG file.

Make sure the 1 Ball material is selected in the Materials window, so it shows up in the Inspector. Click on the *1 Ball Texture* file and **drag it into the box to the left of the Albedo label**.

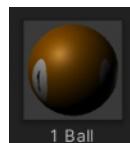


Select the 1 Ball material in the Project window so you can see its properties, then drag the texture map onto the box to the left of the Albedo label.

You should now see a tiny little picture of the 1 Ball texture in the box to the left of Albedo in the Inspector.



Now your material looks like a billiard ball when wrapped around a sphere.



GameObjects reflect light from their surfaces.

Behind the Scenes



When you see an object in a Unity game with a color or texture map, you're seeing the surface of a GameObject reflecting light from the scene, and the **albedo** controls the color of that surface. Albedo is a term from physics (specifically astronomy) that means the color that's reflected by an object. You can learn more about albedo from the Unity Manual. Choose "Unity Manual" from the Help menu to open the manual in a browser and search for "albedo"—there's a manual page that explains albedo color and transparency.

Spawn a billiard ball at a random point in the scene

Create a new Sphere GameObject with a script called OneBallBehaviour:

- ★ Choose 3D Object >> Sphere from the GameObject menu to **create a sphere**.
- ★ Drag your new **1 Ball material** onto it to make it look like a billiard ball.
- ★ Next, **right-click on the Scripts folder** that you created in the Project window and **create a new C# script** named OneBallBehaviour.
- ★ **Drag the script onto the Sphere** in the Hierarchy window. Select the sphere and make sure a Script component called “One Ball Behaviour” shows up in the Inspector window.

Double-click on your new script to edit it in Visual Studio. **Add exactly the same code** that you used in BallBehaviour in the first Unity Lab, then **comment out the Debug.DrawRay line** in the Update method.

Your OneBallBehaviour script should now look like this:

```
public class OneBallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;

    // Start is called before the first frame update
    void Start()
    {
        Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
        transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
        // Debug.DrawRay(Vector3.zero, axis, Color.yellow); ← You won't need this line, so
    }
}

// Update is called once per frame
void Update()
{
}
```

We won't include the using lines in the script code, but assume they're there.

When you add a Start method to a GameObject, Unity calls that method every time a new instance of that object is added to the scene. If the Start method is in a script attached to a GameObject that appears in the Hierarchy window, that method will get called as soon as the game starts.

Unity often instantiates a GameObject some time before it's added to the scene. It only calls the Start method when the GameObject is actually added to the scene.

Now modify the Start method to move the sphere to a random position when it's created. You'll do this by setting **transform.position**, which changes the position of the GameObject in the scene. Here's the code to position your ball at a random point—**add it to the Start** method of your OneBallBehaviour script:

```
// Start is called before the first frame update
void Start()
{
    transform.position = new Vector3(3 - Random.value * 6,
                                    3 - Random.value * 6, 3 - Random.value * 6);
}
```

Remember, the Play button does not save your game! Make sure you save early and save often.

Use the Play button in Unity to run your game. A ball should now be circling the Y axis at a random point. Stop and start the game a few times. The ball should spawn at a different point in the scene each time.

Unity Lab #3

GameObject Instances

Use the debugger to understand Random.value

You've used the Random class in the .NET System namespace a few times already. You used it to scatter the animals in the animal matching game in Chapter 1 and to pick random cards in Chapter 3. This Random class is different—try hovering over the Random keyword in Visual Studio.

These classes are both called Random, but if you hover over them in Visual Studio to see the tooltip, you'll see that the one you used earlier is in the System namespace. Now you're using the Random class in the UnityEngine namespace.
cl {

```
// Start is called before the first frame update
void Start()
{
    transform.position = new Vector3(3 - Random.value * 6,
                                    3 - Random.value * 6, 3 - Random.value * 6);
}

static Random random = new Random();
public st Represents a pseudo-random number generator, which is a device that produces a sequence of numbers that meet certain statistical requirements for randomness.

string[] pickedCards = new string[numberOfCards];
```

class UnityEngine.Random
Class for generating random data.

This is from the code you wrote earlier to pick random cards.

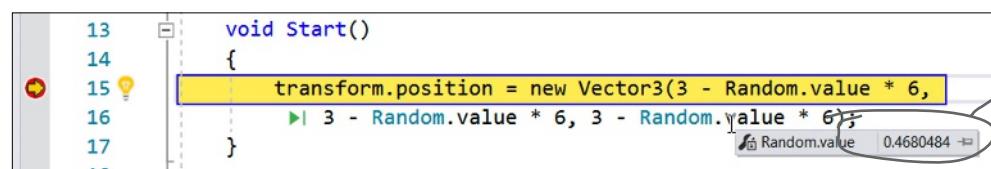
You can see from the code that this new Random class is different from the one you used before. Earlier you called Random.Next to get a random value, and that value was a whole number. This new code uses **Random.value**, but that's not a method—it's actually a property.

Use the Visual Studio debugger to see the kinds of values that this new Random class gives you. Click the “Attach to Unity” button (▶ Attach to Unity) in Windows, (▶ Debug > Attach to Unity) in macOS to attach Visual Studio to Unity. Then **add a breakpoint** to the line you added to the Start method.

Unity may prompt you to enable debugging, just like in the last Unity Lab.

Now go back to Unity and **start your game**. It should break as soon as you press the Play button.

Hover your cursor over Random.value—make sure it's over **value**. Visual Studio will show you its value in a tooltip:



Keep Visual Studio attached to Unity and restart your game a few times. You'll get a new random number between 0 and 1 each time you restart it.

Keep Visual Studio attached to Unity, then go back to the Unity editor and **stop your game** (in the Unity editor, not in Visual Studio). Start your game again. Do it a few more times. You'll get a different random value each time. That's how UnityEngine.Random works: it gives you a new random value between 0 and 1 each time you access its value property.

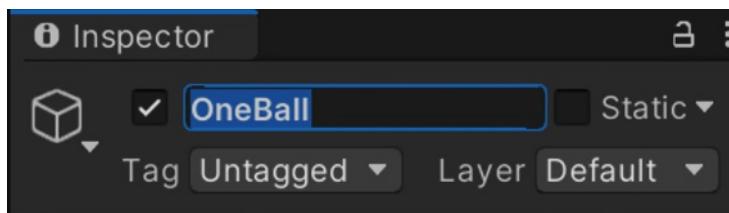
Press Continue (▶ Continue) to resume your game. It should keep running—the breakpoint was only in the Start method, which is just called once for each GameObject instance, so it won't break again. Then go back to Unity and stop the game.

You can't edit scripts in Visual Studio while it's attached to Unity, so click the square Stop Debugging button to detach the Visual Studio debugger from Unity.

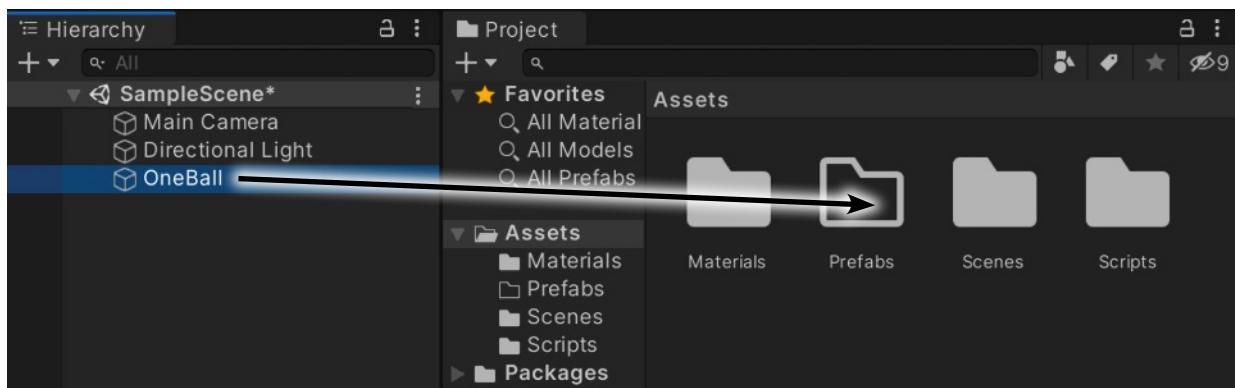
Turn your GameObject into a prefab

In Unity, a **prefab** is a GameObject that you can instantiate in your scene. Over the past few chapters you've been working with object instances, and creating objects by instantiating classes. Unity lets you take advantage of objects and instances, so you can build games that reuse the same GameObjects over and over again. Let's turn your 1 ball GameObject into a prefab.

GameObjects have names.. Change the name of your GameObject to *OneBall*. Start by **selecting your sphere**, by clicking on it in the Hierarchy window or in the scene. Then use the Inspector window to **change its name to OneBall**.



Now you can turn your GameObject into a prefab. **Drag OneBall from the Hierarchy window into the Prefabs folder**.



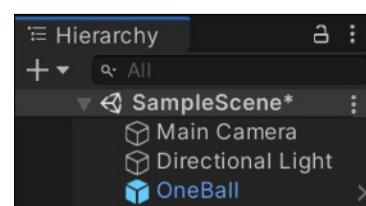
OneBall should now appear in your Prefabs folder. Notice that ***OneBall is now blue in the Hierarchy window***. This indicates that it's now a prefab—Unity turned it blue to tell you that an instance of a prefab is in your hierarchy. That's fine for some games, but for this game we want all of the instances of the balls to be created by scripts.

Right-click on OneBall in the Hierarchy window **and delete the OneBall GameObject from the scene**. You should now only see it in the Project window, and not in the Hierarchy window or the scene.

Have you been saving your scene as you go? Save early, save often!



You can also rename a GameObject by right-clicking on it in the Hierarchy window and choosing Rename.



When a GameObject is blue in the Hierarchy window, Unity is telling you it's a prefab instance.

Unity Lab #3

GameObject Instances

Create a script to control the game

The game needs a way to add balls to the scene (and eventually keep track of the score, and whether or not the game is over).

Right-click on the Scripts folder in the Project window and **create a new script called GameController**. Your new script will use two methods available in any GameObject script:

- ★ **The Instantiate method creates a new instance of a GameObject.**

When you're instantiating GameObjects in Unity, you don't typically use the `new` keyword like you saw in Chapter 2. Instead, you'll use the `Instantiate` method, which you'll call from the `AddABall` method.

- ★ **The InvokeRepeating method calls another method in the script over and over again.**

In this case, it will wait one and a half seconds, then call the `AddABall` method once a second for the rest of the game.

What's the type of the second argument that you're passing to `InvokeRepeating`?

Here's the source code for it:

```
public class GameController : MonoBehaviour
{
    public GameObject OneBallPrefab;

    void Start()
    {
        InvokeRepeating("AddABall", 1.5F, 1);
    }

    void AddABall()
    {
        Instantiate(OneBallPrefab);
    }
}
```

Unity's `InvokeRepeating` method calls another method over and over again. Its first parameter is a string with the name of the method to call ("invoke" just means calling a method).

This is a method called `AddABall`. All it does is create a new instance of a prefab.

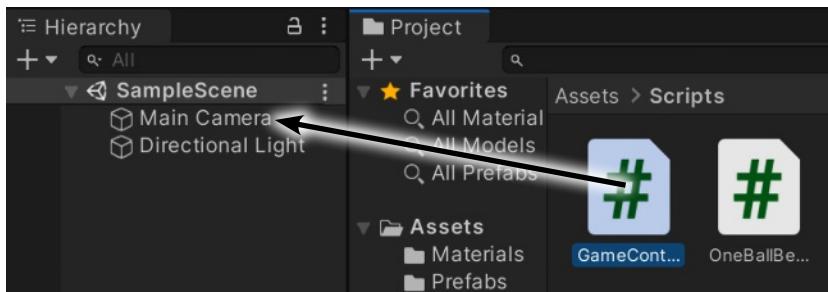
You're passing the `OneBallPrefab` field as a parameter to the `Instantiate` method, which Unity will use to create an instance of your prefab.



Unity will only run scripts that are attached to GameObjects in a scene. The `GameController` script will create instances of our `OneBall` prefab, but we need to attach it to something. Luckily, we already know that a camera is just a GameObject with a Camera component (and also an AudioListener). The Main Camera will always be available in the scene. So...what do you think you'll do with your new `GameController` script?

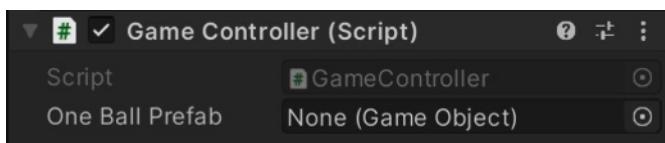
Attach the script to the Main Camera

Your new GameController script needs to be attached to a GameObject to run. Luckily, the Main Camera is just another GameObject—it happens to be one with a Camera component and an AudioListener component—so let's attach your new script to it. **Drag your GameController** script out of the Scripts folder in the Project window and **onto the Main Camera** in the Hierarchy window.



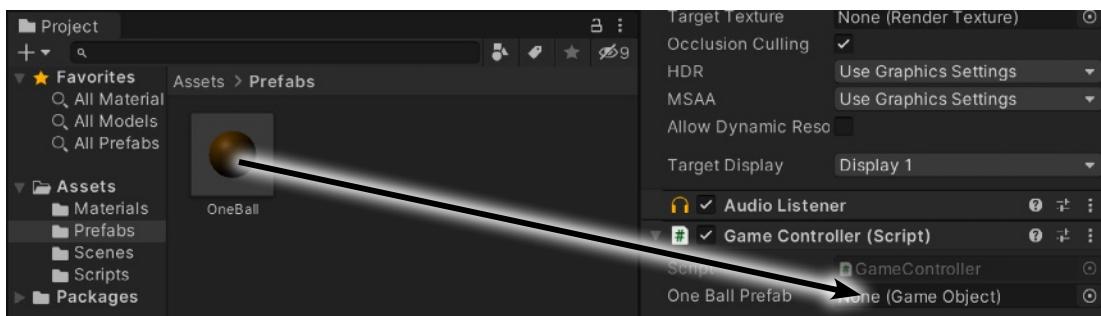
You learned all about public versus private fields in Chapter 5. When a script class has a public field, the Unity editor shows that field in the Script component in the Inspector. It adds spaces between uppercase letters to make its name easier to read.

Look in the Inspector—you'll see a component for the script, exactly like you would for any other GameObject. The script has a **public field called OneBallPrefab**, so Unity displays it in the Script component.



Here's the OneBallPrefab field in your GameController class. Unity added spaces before uppercase letters to make it easier to read (just like we saw in the last lab).

The OneBallPrefab field still says None, so we need to set it. **Drag OneBall out of the Prefabs folder and onto the box next to the One Ball Prefab label.**



Now the GameController's OneBallPrefab field contains a **reference** to the OneBall prefab:



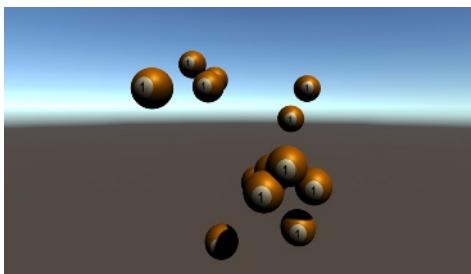
Go back to the code and **look closely the AddABall method**. It calls the Instantiate method, passing it the OneBallPrefab field as an argument. You just set that field so that it contains your prefab. So every time GameController calls its AddABall method, it will **create a new instance of the OneBall prefab**.

Unity Lab #3

GameObject Instances

Press Play to run your code

Your game is all ready to run. The GameController script attached to the Main Camera will wait 1.5 seconds, then instantiate a OneBall prefab every second. Each instantiated OneBall's Start method will move it to a random position in the scene, and its Update method will rotate it around the Y axis every 2 seconds using OneBallBehaviour fields (just like in the last Lab). Watch as the play area slowly fills up with rotating balls:

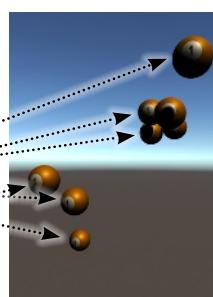
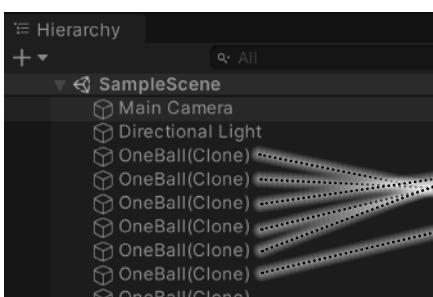


Unity calls every GameObject's Update method before each frame. That's called the update loop.

When you instantiate GameObjects in your code, they show up in the Hierarchy window when you run your game.

Watch the live instances in the Hierarchy window

Each of the balls flying around the scene is an instance of the OneBall prefab. Each of the instances has its own instance of the OneBallBehaviour class. You can use the Hierarchy window to track all of the OneBall instances—as each one is created, a “OneBall(Clone)” entry is added to the Hierarchy.



We've included some coding exercises in the Unity Labs. They're just like the exercises in the rest of the book—and remember, it's not cheating to peek at the solution.



Click on any of the OneBall(Clone) items to view it in the Inspector. You'll see its Transform values change as it rotates, just like in the last lab.



Figure out how to add a BallNumber field to your OneBallBehaviour script, so that when you click on a OneBall instance in the Hierarchy and check its One Ball Behaviour (Script) component, under the X Rotation, Y Rotation, Z Rotation, and Degrees Per Second labels it has a Ball Number field:

Ball Number

11

The first instance of OneBall should have its Ball Number field set to 1. The second instance should have it set to 2, the third 3, etc. Here's a hint: you'll need a way to keep track of the count that's **shared by all of the OneBall instances**. You'll modify the Start method to increment it, then use it to set the BallNumber field.

Use the Inspector to work with GameObject instances

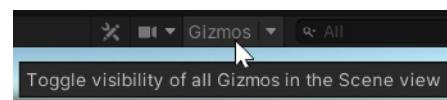
Run your game. Once a few balls have been instantiated, click the Pause button—the Unity editor will jump back to the Scene view. Click one of the OneBall instances in the Hierarchy window to select it. The Unity editor will outline it in the Scene window to show you which object you selected. Go to the Transform component in the Inspector window and **set its Z scale value to 4** to make the ball stretch.



Start your simulation again—now you can track which ball you’re modifying. Try changing its DegreesPerSecond, XRotation, YRotation, and ZRotation fields like you did in the last lab.

While the game is running, switch between the Game and Scene views. You can use the Gizmos in the Scene view **while the game is running**, even for GameObject instances that were created using the Instantiate method (rather than added to the Hierarchy window).

Try clicking the Gizmos button at the top of the toolbar to toggle them on and off. You can turn on the Gizmos in the Game view, and you can turn them off in the Scene view.



Exercise Solution

You can add a BallNumber field to the OneBallBehaviour script by keeping track of the total number of balls added so far in a static field (which we called BallCount). Each time a new ball is instantiated, Unity calls its Start method, so you can increment the static BallCount field and assign its value to that instance’s BallNumber field.

```
static int BallCount = 0;
public int BallNumber;

void Start()
{
    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);

    BallCount++;
    BallNumber = BallCount;
}
```

All of the OneBall instances share a single static BallCount field, so the first instance’s Start method increments it to 1, the second instance increments BallCount to 2, the third increments it to 3, etc.

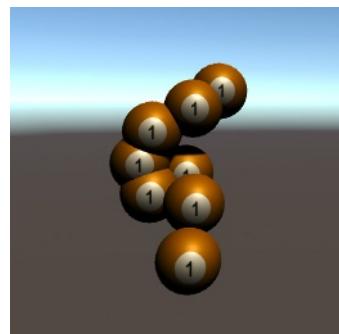
Unity Lab #3

GameObject Instances

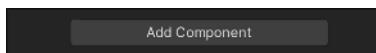
Use physics to keep balls from overlapping

Did you notice that occasionally some of the balls overlap each other?

Unity has a powerful **physics engine** that you can use to make your GameObjects behave like they're real, solid bodies—and one thing that solid shapes don't do is overlap each other. To prevent that overlap, you just need to tell Unity that your OneBall prefab is a solid object.



Stop your game, then **click on the OneBall prefab in the Project window** to select it. Then go to the Inspector and scroll all the way down to the bottom to the Add Component button:



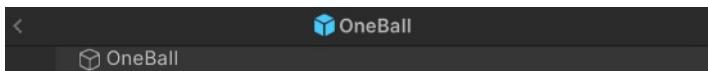
Click the button to pop up the Component window. **Choose Physics** to view the physics components, then **select Rigidbody** to add the component.



While you're running physics experiments, here's one Galileo would appreciate. Try checking the Use Gravity box while your game is running. New balls that get created will start falling, occasionally hitting another ball and knocking it out of the way.

Run your game again—now you won't see balls overlap. Occasionally one ball will get created on top of another one. When that happens, the new ball will knock the old one out of the way.

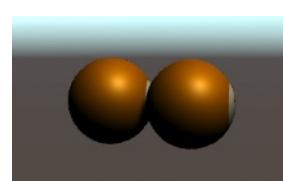
Let's run a little physics experiment to prove that the balls really are rigid now. Start your game, then pause it as soon as there are more than two balls created. Go to the Hierarchy window. If it looks like this:



then you're editing the prefab—click the back caret () in the top-right corner of the Hierarchy window to get back to the scene (you may need to expand SampleScene again).

- ★ Hold down the Shift key, click the first OneBall instance in the Hierarchy window, and then click the second one so the first two OneBall instances are selected.
- ★ You'll see dashes () in the Position boxes in the Transform panel. **Set the Position to (0, 0, 0)** to set both OneBall instances' positions at the same time.
- ★ Use Shift-click to select any other instances of OneBall, right-click, and **choose Delete** to delete them from the scene so only the two overlapping balls are left.
- ★ Unpause your game—the balls can't overlap now, so instead they'll be rotating next to each other.

You can use the Hierarchy window to delete GameObjects from your scene while the game is running.



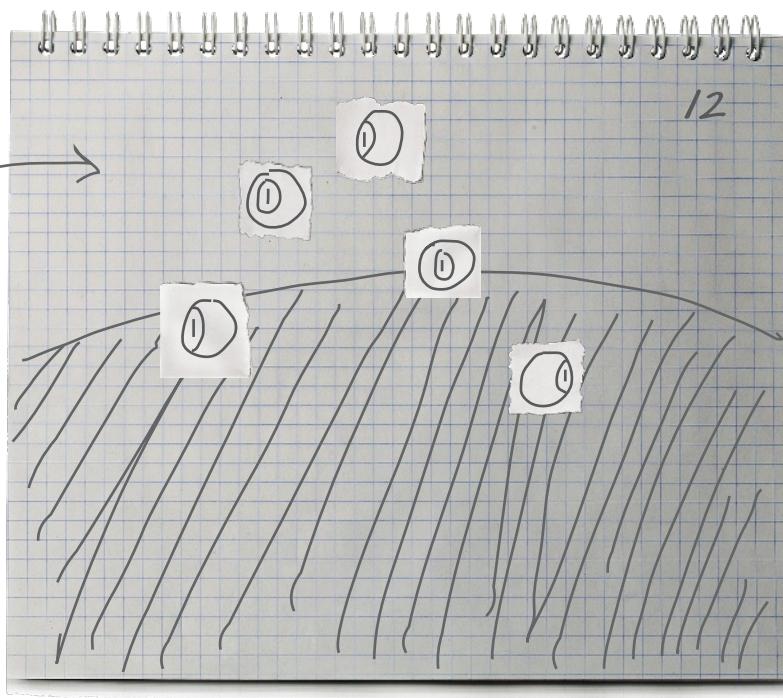
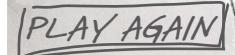
Stop the game in Unity and Visual Studio and save your scene. Save early, save often!

Get creative!

You're halfway done with the game! You'll finish it in the next Unity Lab. In the meantime, this is a great opportunity to practice your **paper prototyping** skills. We gave you a description of the game at the beginning of this Unity Lab. Try creating a paper prototype of the game. Can you come up with ways to make it more interesting?

Draw the background of the Unity scene on a piece of paper, then draw billiard balls on scraps of paper.

Here's the "Play Again" button that's displayed when the game is over.



Can you come up with a way to use the 8 ball from the first two Unity Labs to make the game more fun?



BULLET POINTS

- **Albedo** is a physics term that means the color that's reflected by an object. Unity can use texture maps for the albedo in a material.
- Unity has its own **Random class** in the `UnityEngine` namespace. The static `Random.value` method returns a random number between 0 and 1.
- A **prefab** is a `GameObject` that you can instantiate in your scene. You can turn any `GameObject` into a prefab.
- The **Instantiate method** creates a new instance of a `GameObject`. The `Destroy` method destroys it. Instances are created and destroyed at the end of the update loop.
- The **InvokeRepeating method** calls another method in the script over and over again.
- Unity calls every `GameObject`'s `Update` method before each frame. That's called the **update loop**.
- You can **inspect the live instances** of your prefabs by clicking on them in the Hierarchy window.
- When you add a **Rigidbody** component to a `GameObject`, Unity's physics engine makes it act like a real, solid, physical object.
- The `Rigidbody` component lets you turn **gravity** on or off for a `GameObject`.

7 interfaces, casting, and “is”



Making classes keep their promises



OK, OK,
I KNOW I EXTENDED THE
IBOOKIECUSTOMER INTERFACE! BUT I'M
A LITTLE SHORT, AND CAN'T IMPLEMENT
THE PAYMONEY METHOD UNTIL
FRIDAY.



YOU'VE
GOT TWO DAYS BEFORE
I SEND SOME **GOON** OBJECTS
TO COME BY AND IMPLEMENT
YOUR **WALKSWITHALIMP**
METHOD.

Need an object to do a specific job? Use an interface.

Sometimes you need to group your objects together based on the **things they can do** rather than the classes they inherit from—and that's where **interfaces** come in. You can use an interface to define a **specific job**. Any instance of a class that **implements** the interface is *guaranteed to do that job*, no matter what other classes it's related to. To make it all work, any class that implements an interface must promise to **fulfill all of its obligations**...or the compiler will break its kneecaps, see?

add a new subclass to the Bee class hierarchy

The beehive is under attack!

An enemy hive is trying to take over the Queen's territory, and keeps sending enemy bees to attack her workers. So she's added a new elite Bee subclass called HiveDefender to defend the hive.

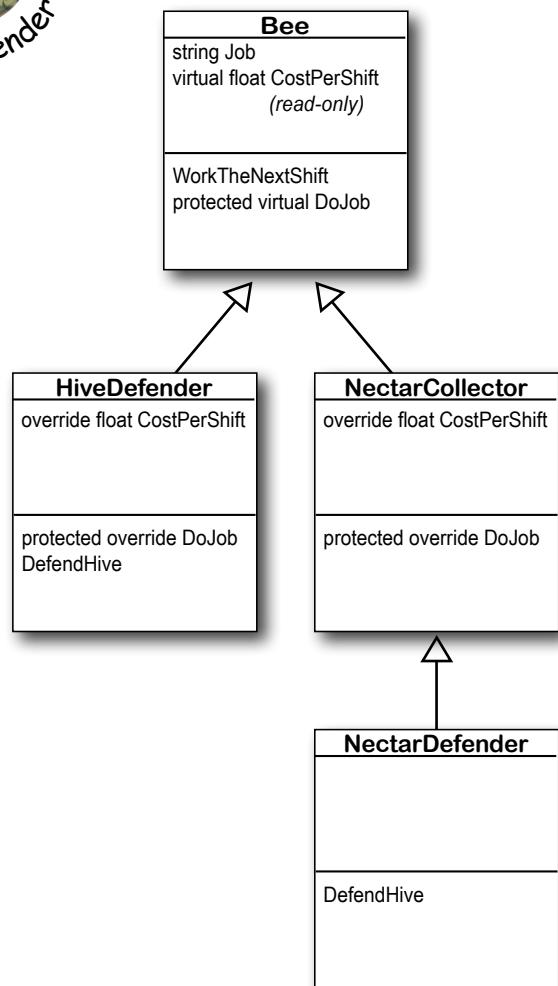


So we need a `DefendHive` method, because enemies can attack at any time

We can add a HiveDefender subclass to the Bee class hierarchy by extending the Bee class, overriding its CostPerShift with the amount of honey each defender consumes every shift, and overriding the DoJob method to fly out to the enemy hive and attack the enemy bees.

But enemy bees can attack at any time. We want defenders to be able to defend the hive *whether or not they're currently doing their normal jobs*.

So in addition to DoJob, we'll add a DefendHive method to any Bee that can defend the hive—not just the elite HiveDefender workers, but any of their sisters who can take up arms and protect their Queen. The Queen will call her workers' DefendHive methods any time she sees that her hive is under attack.



We can use casting to call the DefendHive method...

When you coded the Queen.DoJob method, you used a foreach loop to get each Bee reference in the `workers` array, then you used that reference to call worker.DoJob. If the hive is under attack, the Queen will want to call her defenders' DefendHive methods. So we'll give her a HiveUnderAttack method that gets called any time the hive is being attacked by enemy bees, and she'll use a foreach loop to order her workers to defend the hive until all of the attackers are gone.

But there's a problem. The Queen can use the Bee references to call DoJob because each subclass overrides Bee.DoJob, but she can't use a Bee reference to call the DefendHive method, because that method isn't part of the Bee class. So how does she call DefendHive?

Since DefendHive is only defined in each subclass, we'll need to use **casting** to convert the Bee reference to the correct subclass in order to call its DefendHive method.

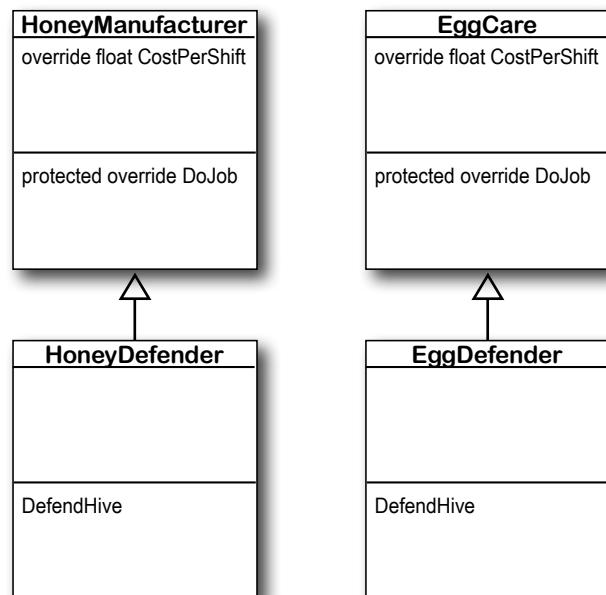
```
public void HiveUnderAttack() {
    foreach (Bee worker in workers) {
        if (EnemyHive.AttackingBees > 0) {
            if (worker.Job == "Hive Defender") {
                HiveDefender defender = (HiveDefender) worker;
                defender.DefendHive();
            } else if (worker.Job == "Nectar Defender") {
                NectarDefender defender = (NectarDefender) defender;
                defender.DefendHive();
            }
        }
    }
}
```

The honey manufacturer and egg care bees want to help defend the hive, too. Does the Queen need to have an if/else for each subclass?

...but what if we add more Bee subclasses that can defend?

Some honey manufacturer and egg care bees want to step up and defend the hive, too. That means we'll need to add more `else` blocks to her HiveUnderAttack method.

This is getting complex. The Queen.DoJob method is nice and simple—a very short foreach loop that takes advantage of the Bee class model to call the specific version of the DoJob method that was implemented in the subclass. We can't do that with DefendHive because it's not part of the Bee class—and we don't want to add it, because not all bees can defend the hive. **Is there a better way to have unrelated classes do the same job?**



An interface defines methods and properties that a class must implement...

An **interface** works just like an abstract class: you use abstract methods, and then you use the colon (:) to make a class implement that interface.

So if we wanted to add defenders to the hive, we could have an interface called IDefend. Here's what that looks like. It uses the **interface keyword** to define the interface, and it includes a single member, an abstract method called Defend. All members in an interface are public and abstract by default, so C# keeps things simple by having you **leave off the public and abstract keywords**:

```
interface IDefend
{
    void Defend();
}
```

This interface has one member, a public abstract method called Defend. This works just like the abstract methods you saw in Chapter 6.

Any class that implements the IDefend interface **must include a Defend method** whose declaration matches the one in the interface. If it doesn't, the compiler will give an error.

...but there's no limit to the number of interfaces a class can implement

We just said that you use a colon (:) to make a class implement an interface. What if that class is already using a colon to extend a base class? No problem! **A class can implement many different interfaces, even if it already extends a base class**:

```
class NectarDefender : NectarCollector, IDefend
{
    void Defend() {
        /* Code to defend the hive */
    }
}
```

Since the Defend method is part of the IDefend interface, the NectarDefender class must implement it or it won't compile.

When a class implements an interface, it must include all of the methods and properties listed inside the interface or the code won't build.

Now we have a class that can act like a NectarCollector, but can also defend the hive. NectarCollector extends Bee, so if you **use it from a Bee reference** it acts like a Bee:

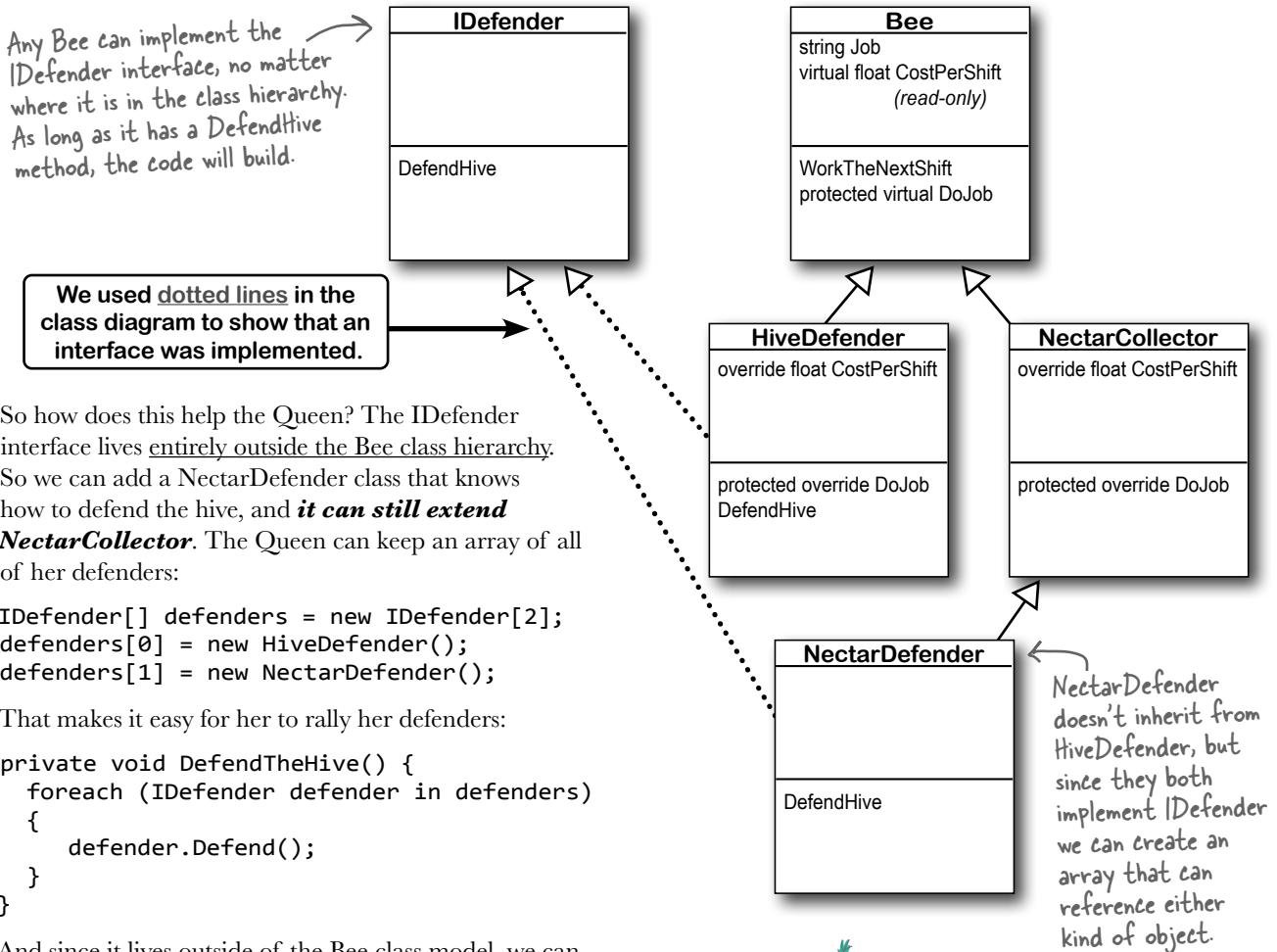
```
Bee worker = new NectarCollector();
Console.WriteLine(worker.Job);
worker.WorkTheNextShift();
```

But if you **use it from an IDefend reference**, it acts like a hive defender:

```
IDefend defender = new NectarCollector();
defender.Defend();
```

Interfaces let unrelated classes do the same job

Interfaces can be a really powerful tool to help you design C# code that's easy to understand and build. Start by thinking about **specific jobs that classes need to do**, because that's what interfaces are all about.



NOW THAT I
KNOW YOU CAN DEFEND
THE HIVE, WE'LL ALL BE
A LOT SAFER!



Relax
We're going to give you a lot
of examples of interfaces.

Still a little puzzled about how interfaces work and why you would use them? Don't worry—that's normal! The syntax is pretty straightforward, but there's **a lot of subtlety**. So we'll spend more time on interfaces...and we'll give you plenty of examples, and lots of practice.

Get a little practice using interfaces

The best way to understand interfaces is to start using them. Go ahead and **create a new Console App** project.



- 1 Add the Main method.** Here's the code for a class called TallGuy, along with code for the Main method that instantiates it using an object initializer and calls its TalkAboutYourself method. There's nothing new here—we'll use it in a minute:

```
class TallGuy {
    public string Name;
    public int Height;

    public void TalkAboutYourself() {
        Console.WriteLine($"My name is {Name} and I'm {Height} inches tall.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        TallGuy tallGuy = new TallGuy() { Height = 76, Name = "Jimmy" };
        tallGuy.TalkAboutYourself();
    }
}
```

- 2 Add an interface.** We're going to make TallGuy implement an interface. Add a new IClown interface to your project: right-click on the project in the Solution Explorer, **select Add >> New Item...** (**Windows**) or **Add >> New File...** (**Mac**), and choose **Interface**. Make sure it's called *IClown.cs*. The IDE will create an interface that includes the interface declaration. Add a Honk method:

```
interface IClown
{
    void Honk();
}
```

You don't need to add "public" or "abstract" inside the interface, because it automatically makes every property and method public and abstract.

- 3 Try coding the rest of the IClown interface.** Before you go on to the next step, see if you can create the rest of the IClown interface, and modify the TallGuy class to implement this interface. In addition to the void method called Honk that doesn't take any parameters, your IClown interface should also have a read-only string property called FunnyThingIHave that has a get accessor but no set accessor.

Interface names start with I

Whenever you create an interface, you should make its name start with an uppercase I. There's no rule that says you need to do it, but it makes your code a lot easier to understand. You can see for yourself just how much easier that can make your life. Just go into the IDE to any blank line inside any method and type "I"—IntelliSense shows .NET interfaces.

- 4** Here’s the **IClown** interface. Did you get it right? It’s OK if you put the Honk method first—the order of the members doesn’t matter in an interface, just like it doesn’t matter in a class.

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();
}
```

The IClown interface requires any class that implements it to have a void method called Honk and a string property called FunnyThingIHave that has a get accessor.

- 5** Modify the **TallGuy** class so that it implements **IClown**. Remember, the colon operator is always followed by the base class to inherit from (if any), and then a list of interfaces to implement, all separated by commas. Since there’s no base class and only one interface to implement, the declaration looks like this:

```
class TallGuy : IClown
```

Then make sure the rest of the class is the same, including the two fields and the method. Select Build Solution from the Build menu in the IDE to compile and build the program. You’ll see two errors:

CS0535 'TallGuy' does not implement interface member 'IClown.FunnyThingIHave'
 CS0535 'TallGuy' does not implement interface member 'IClown.Honk()'

- 6** Fix the errors by adding the missing interface members. The errors will go away as soon as you add all of the methods and properties defined in the interface. So go ahead and implement the interface. Add a read-only string property called FunnyThingIHave with a get accessor that always returns the string “big shoes”. Then add a Honk method that writes “Honk honk!” to the console.

Here’s what it’ll look like:

```
public string FunnyThingIHave {
    get { return "big shoes"; }
}

public void Honk() {
    Console.WriteLine("Honk honk!");
}
```

Any class that implements the IClown interface must have a void method called Honk and a string property called FunnyThingIHave that has a get accessor. The FunnyThingIHave property is allowed to have a set accessor, too. The interface doesn’t specify it, so it doesn’t matter either way.

- 7** Now your code will compile. Update your Main method so that it prints the TallGuy object’s FunnyThingIHave property and then calls its Honk method:

```
static void Main(string[] args) {
    TallGuy tallGuy = new TallGuy() { Height = 76, Name = "Jimmy" };
    tallGuy.TalkAboutYourself();
    Console.WriteLine($"The tall guy has {tallGuy.FunnyThingIHave}");
    tallGuy.Honk();
}
```



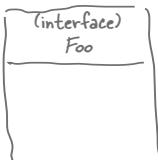
If you're given...

1)

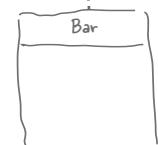
```
interface Foo { }  
class Bar : Foo { }
```

We did the first
one for you.

1)



2)



2)

```
interface Vinn { }  
abstract class Vout : Vinn { }
```

3)

3)

```
abstract class Muffie : Whuffie { }  
class Fluffie : Muffie { }  
interface Whuffie { }
```

4)

4)

```
class Zoop { }  
class Boop : Zoop { }  
class Goop : Boop { }
```

You'll need a little
more room for #5.

5)

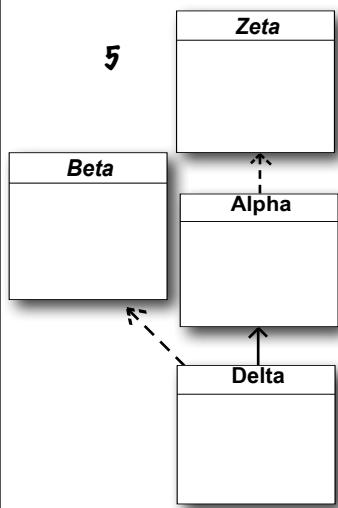
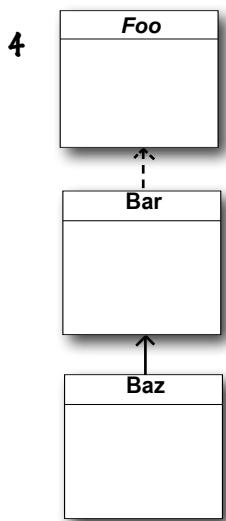
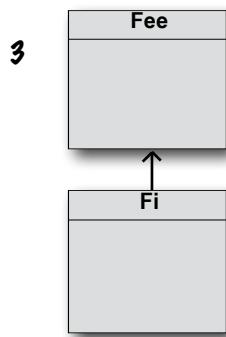
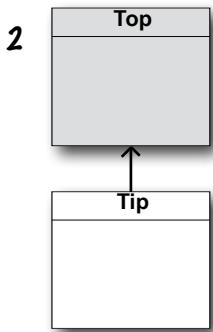
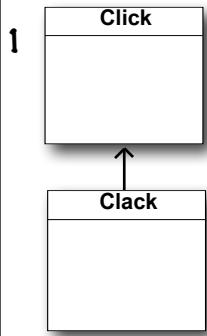
```
class Gamma : Delta, Epsilon { }  
interface Epsilon { }  
interface Beta { }  
class Alpha : Gamma, Beta { }  
class Delta { }
```

What's the picture?

Here's your chance to demonstrate your artistic abilities. On the left, you'll find sets of class and interface declarations. Your job is to draw the associated class diagrams on the right. Don't forget to use a dashed line for implementing an interface and a solid line for inheriting from a class.



If you're given...



KEY

extends

implements

Clack

class

Clack

interface

Clack

abstract class

On the left, you'll find sets of class diagrams. Your job is to turn these into valid C# declarations. **We did number 1 for you.** Notice how the class declarations are just a pair of curly braces {}? That's because these classes don't have any members. (But they're still valid classes that build!)

What's the declaration?

1) public class Click { }

public class Clack : Click { }

2)

3)

4)

5)

Fireside Chats



Tonight's talk: **An abstract class and an interface butt heads over the pressing question, "Who's more important?"**

Abstract Class:

I think it's obvious who's more important between the two of us. Programmers need me to get their jobs done. Let's face it. You don't even come close.

You can't really think you're more important than me. You don't even use real inheritance—you only get implemented.

Better? You're nuts. I'm much more flexible than you. Sure, I can't be instantiated—but then, neither can you. Unlike you, I have the **awesome power** of inheritance. The poor saps that extend you can't take advantage of **virtual** and **override** at all!

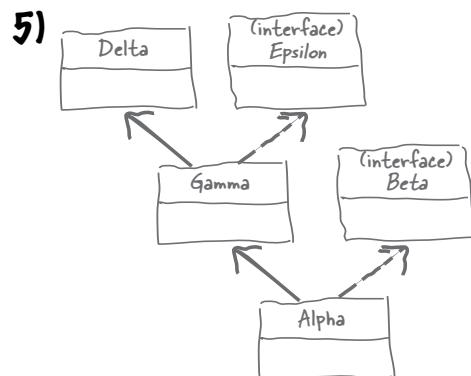
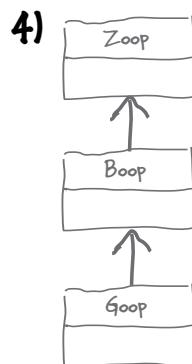
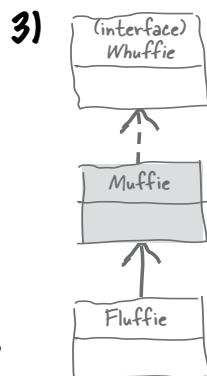
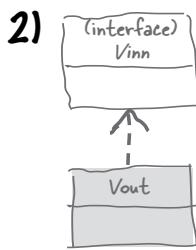
Interface:

Nice. This oughta be good.

Great, here we go again. “Interfaces don’t use real inheritance.” “Interfaces only implement.” That’s just plain ignorant. Implementation is as good as inheritance. In fact, it’s better!

Yeah? What if you want a class that inherits from you **and** your buddy? **You can't inherit from two classes.** You have to choose which class to inherit from. That’s just plain rude! There’s no limit to the number of interfaces a class can implement. Talk about flexible! With me, a programmer can make a class do anything.

Sharpen your pencil Solution



What's the picture ?

Abstract Class:

You might be overstating your power a little bit.

And you think that's a good thing? Ha! When you use me and my subclasses, you know exactly what's going on inside all of us. I can handle any behavior that all of my subclasses need, and they just need to inherit it. Transparency is a powerful thing, kiddo!

Really? I doubt that—programmers always care what's in their properties and methods.

Yeah, sure, tell a coder they can't code.



2) abstract class Top { }
class Tip : Top { }

4) interface Foo { }
class Bar : Foo { }
class Baz : Bar { }

What's the declaration?

Interface:

Really, now? Well, let's think about just how powerful I can be for developers that use me. I'm all about the job—when they get an interface reference, they don't need to know anything about what's going on inside that object at all.

Nine times out of 10, a programmer wants to make sure an object has certain properties and methods, but doesn't really care how they're implemented.

OK, sure. Eventually. But think about how many times you've seen a programmer write a method that takes an object that just needs to have a certain method, and it doesn't really matter right at that very moment exactly how the method's built—just that it's there. So bang! The programmer just needs to use an interface. Problem solved!

Ugh, you're **so frustrating!**

3) abstract class Fee { }
abstract class Fi : Fee { }

5) interface Zeta { }
class Alpha : Zeta { }
interface Beta { }
class Delta : Alpha, Beta { }

Delta inherits
from Alpha and
implements Beta.

You can't instantiate an interface, but you can reference an interface

Say you need an object that has a Defend method so you can use it in a loop to defend the hive. Any object that implemented the IDefender interface would do. It could be a HiveDefender object, a NectarDefender object, or even a HelpfulLadyBug object. As long as it implements the IDefender interface, that guarantees that it has a Defend method. You just need to call it.

That's where **interface references** come in. You can use one to refer to an object that implements the interface you need and you'll always be sure that it has the right methods for your purpose—even if you don't know much else about it.

If you try to instantiate an interface, your code won't build

You can create an array of IWorker references, but you can't instantiate an interface. What you *can* do is point those references at new instances of classes that implement IWorker. Now you can have an array that holds many different kinds of objects!

If you try to instantiate an interface, the compiler will complain.

```
IDefender barb = new IDefender(); ← THIS WILL NOT COMPILE
```

You can't use the `new` keyword with an interface, which makes sense—the methods and properties don't have any implementation. If you could create an object from an interface, how would it know how to behave?

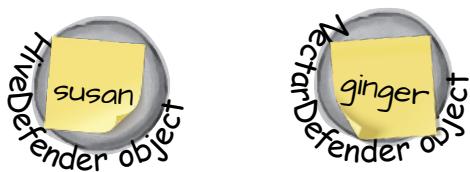
Use the interface to reference an object you already have

So you can't instantiate an interface...but you **can use the interface to make a reference variable**, and use it to reference an object that **implements** the interface.

Remember how you could pass a Tiger reference to any method that expects an Animal, because Tiger extends Animal? Well, this is the same thing—you can use an instance of a class that implements IDefender in any method or statement that expects an IDefender.

```
IDefender susan = new HiveDefender();  
IDefender ginger = new NectarDefender();
```

These are ordinary `new` statements, just like you've been using for most of the book. The only difference is that you're **using a variable of type IDefender** to reference them.



Even though this object can do more, when you use an interface reference you only have access to the methods in the interface.

You used the interface to declare the "susan" and "ginger" variables, but they're normal references that work just like any other object references.



Pool Puzzle

Your **job** is to take code snippets from the pool and place them into the blank lines in the code and output. You may use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a set of classes that will compile and run and produce the output listed.

```

____ INose {
    _____;
    string Face { get; }
}

abstract class _____ : _____ {
    private string face;
    public virtual string Face {
        _____ { _____ _____ ; }
    }

    public abstract int Ear();

    public Picasso(string face)
    {
        _____ = face;
    }
}

class _____ : _____ {
    public Clowns() : base("Clowns") { }

    public override int Ear() {
        return 7;
    }
}

```

```

class _____ : _____ {
    public Acts() : base("Acts") { }
    public override _____ {
        return 5;
    }
}

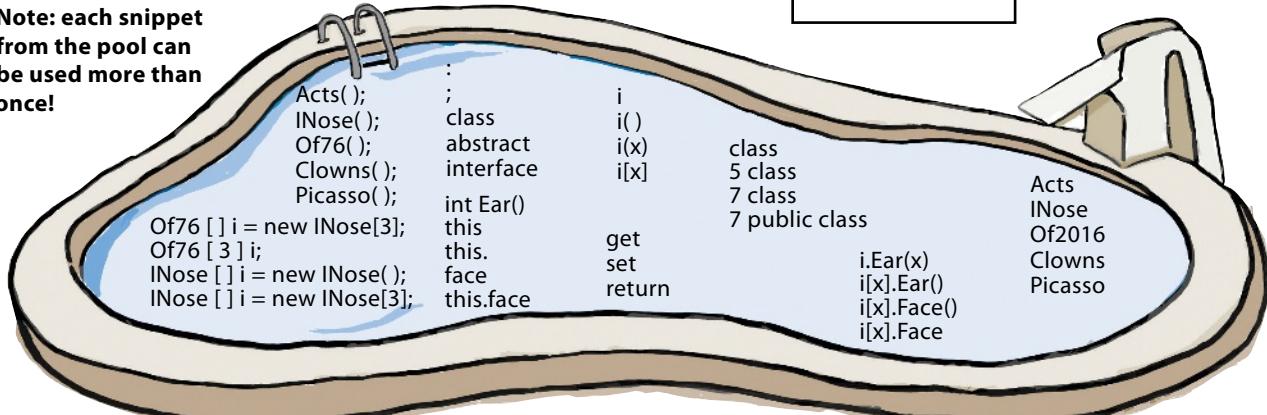
class _____ : _____ {
    public override string Face {
        get { return "Of2016"; }
    }
}

public static void Main(string[] args) {
    string result = "";
    INose[] i = new INose[3];
    i[0] = new Acts();
    i[1] = new Clowns();
    i[2] = new Of2016();
    for (int x = 0; x < 3; x++) {
        result +=
            $"{{_____.Face}} {{_____.Ear()}}\n";
    }
    Console.WriteLine(result);
    Console.ReadKey();
}

```

Here's the entry point—this is a complete C# program.

Note: each snippet from the pool can be used more than once!



Output

5 Acts
7 Clowns
7 Of2016



Pool Puzzle Solution

Your **job** is to take code snippets from the pool and place them into the blank lines in the code and output. You may use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a set of classes that will compile and run and produce the output listed.

Output

5 Acts
7 Clowns
7 Of2016

Face is a get accessor that returns the value of the face property. Both of them are defined in Picasso and inherited into the subclasses.

```
interface INose {
    int Ear();
    string Face { get; } ←
}
```

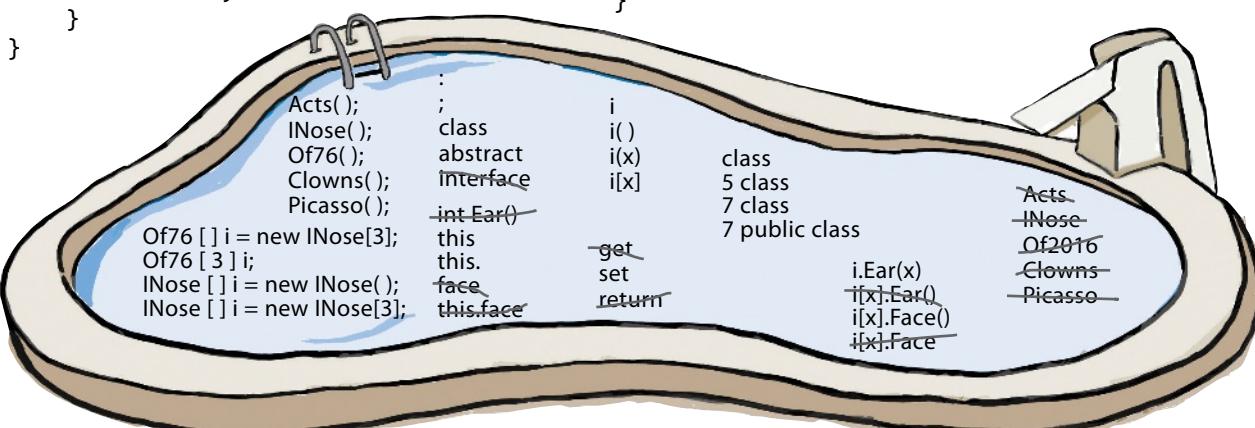
```
abstract class Picasso : INose {
    private string face;
    public virtual string Face {
        get { return face; }
    }
    public abstract int Ear();
    public Picasso(string face)
    {
        this.face = face;
    }
}
```

```
class Clowns : Picasso {
    public Clowns() : base("Clowns") { }
    public override int Ear()
    {
        return 7;
    }
}
```

Here's where the Acts class calls the constructor in Picasso, which it inherits from. It passes "Acts" into the constructor, which gets stored in the Face property.

```
class Acts : Picasso {
    public Acts() : base("Acts") { }
    public override int Ear()
    {
        return 5;
    }
}
```

```
class Of2016 : Clowns {
    public override string Face {
        get { return "Of2016"; }
    }
    public static void Main(string[] args) {
        string result = "";
        INose[] i = new INose[3];
        i[0] = new Acts();
        i[1] = new Clowns();
        i[2] = new Of2016();
        for (int x = 0; x < 3; x++) {
            result +=
                $"{i[x].Ear()} {i[x].Face}\n";
        }
        Console.WriteLine(result);
        Console.ReadKey();
    }
}
```



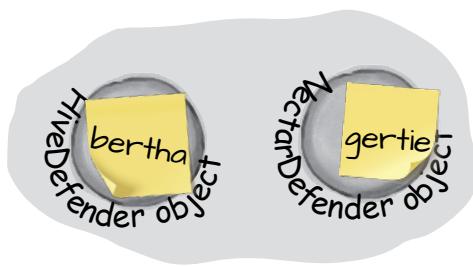
Interface references are ordinary object references

You already know all about how objects live in the heap. When you work with an interface reference, it's just another way to refer to the same objects you've already been using. Let's take a closer look at how interfaces would be used to reference objects in the heap.

1 We'll start by creating objects as usual.

Here's code to create some bees: it creates an instance of HiveDefender and an instance of NectarDefender—and both of those classes implement the IDefender interface.

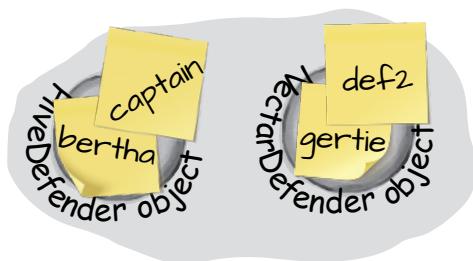
```
HiveDefender bertha = new HiveDefender();
NectarDefender gertie = new NectarDefender();
```



2 Next we'll add IDefender references.

You can use interface references just like you use any other reference type. These two statements use interfaces to create **new references to existing objects**. You can only point an interface reference at an instance of a class that implements it.

```
IDefender def2 = gertie;
IDefender captain = bertha;
```

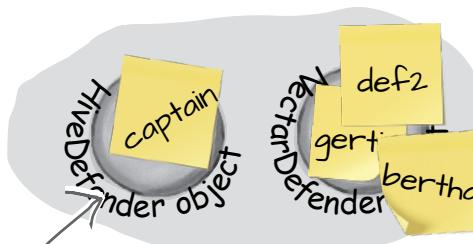


3 An interface reference will keep an object alive.

When there aren't any references pointing to an object, it disappears. There's no rule that says those references all have to be the same type! An interface reference is just as good as any other object reference when it comes to keeping track of objects so they don't get garbage-collected.

```
bertha = gertie; ← Now bertha points to the NectarDefender.
// the captain reference still points to the
// HiveDefender object
```

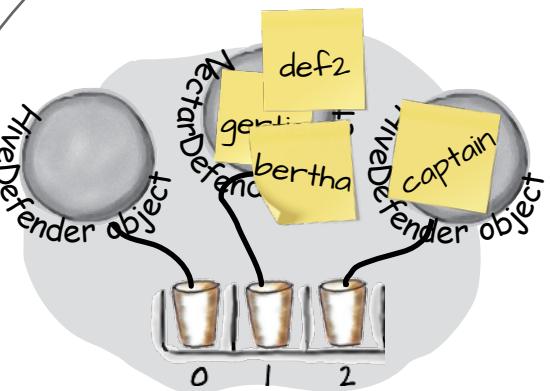
This object didn't disappear from the heap because "captain" still references it.



4 Use an interface like any other type.

You can create a new object with a `new` statement and assign it straight to an interface reference variable in a single line of code. You can **use interfaces to create arrays** that can reference any object that implements the interface.

```
IDefender[] defenders = new IDefender[3];
defenders[0] = new HiveDefender();
defenders[1] = bertha;
defenders[2] = captain;
```

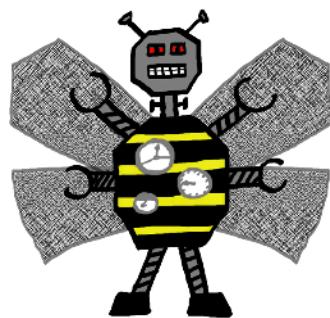


The RoboBee 4000 can do a worker bee's job without using valuable honey

Bee-siness was booming last quarter, and the Queen had enough spare budget to buy the latest in hive technology: the RoboBee 4000. It can do the work of three different bees, and best of all it doesn't consume any honey! It's not exactly environmentally friendly, though—it runs on gas. So how can we use interfaces to integrate RoboBee into the hive's day-to-day business?

```
class Robot
{
    public void ConsumeGas() {
        // Not environmentally friendly
    }
}

class RoboBee4000 : Robot, IWorker
{
    public string Job {
        get { return "Egg Care"; }
    }
    public void WorkTheNextShift()
    {
        // Do the work of three bees!
    }
}
```

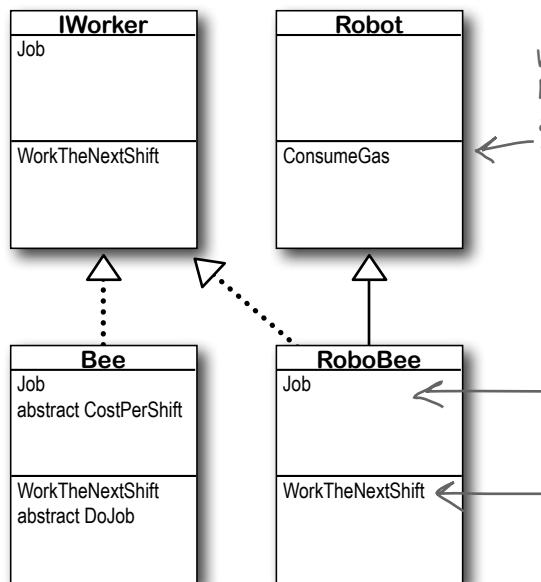


Let's take a close look at the class diagram to see how we can use an interface to integrate a RoboBee class into the Beehive Management System. Remember, we're using dotted lines to show that an object implements an interface.

We can create an IWorker interface that has the two members that have to do with doing work in the hive.

The Bee class implements the IWorker interface, while the RoboBee class inherits from Robot and implements IWorker. That means it's a robot, but can do the job of worker bees.

We'll start with a basic Robot class—everyone knows all robots run on gasoline, so it has a ConsumeGas method.



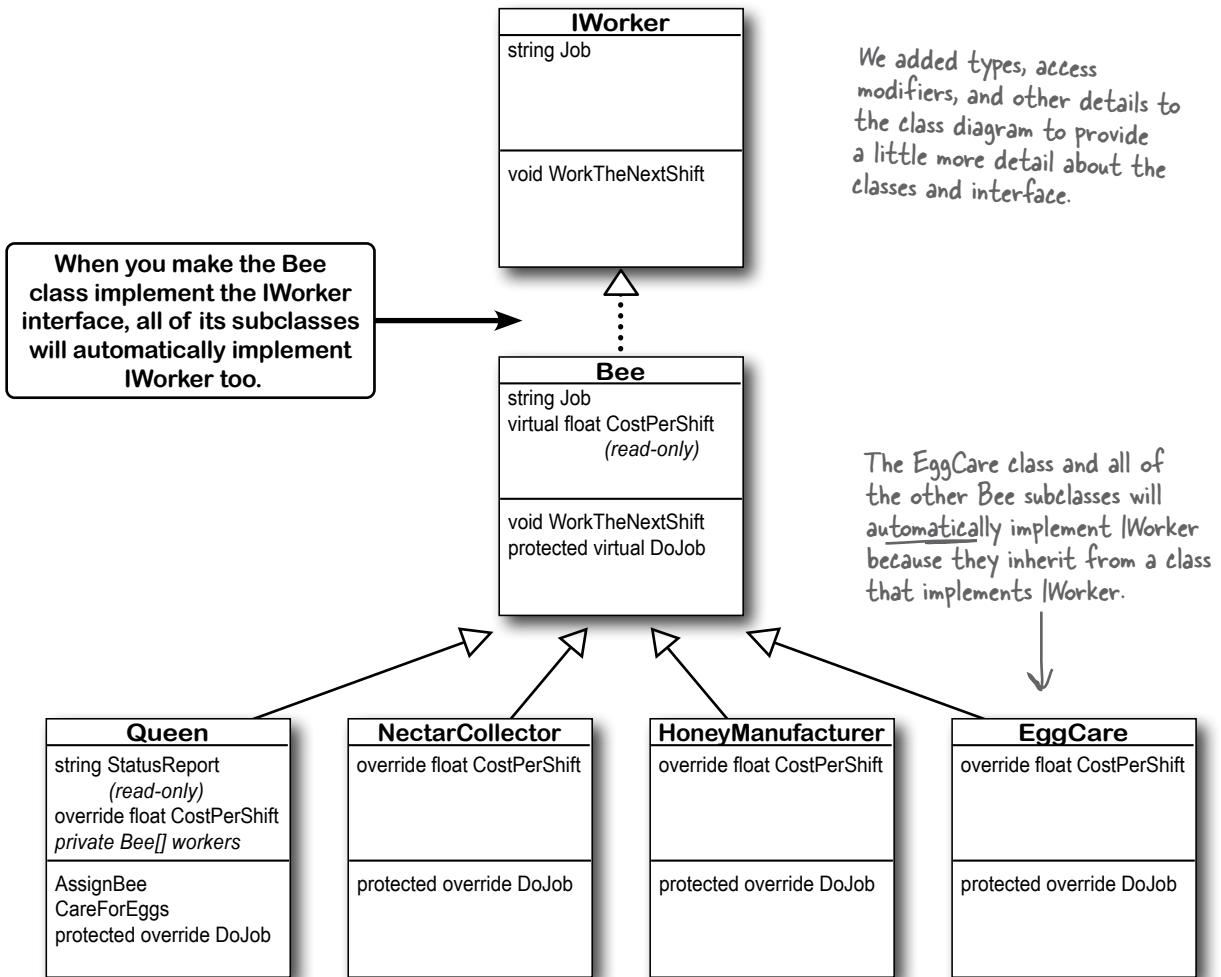
The RoboBee class implements both of the IWorker interface members. We don't have a choice in this—if the RoboBee class doesn't implement everything in the IWorker interface, the code won't compile.

Now all we need to do is modify the Beehive Management System to use the IWorker interface instead of the abstract Bee class any time it needs to reference a worker.

**Exercise**

Modify the Beehive Management System to use the IWorker interface instead of the abstract Bee class any time it needs to reference a worker.

Your job is to add the IWorker interface to your project, then refactor your code to make the Bee class implement it and modify the Queen class so it only uses IWorker references. Here's what the updated class diagram looks like:



Here's what you'll need to do:

- Add the IWorker interface to your Beehive Management System project.
- Modify the Bee class to implement IWorker.
- Modify the Queen class to replace any Bee reference with an IWorker reference.

If that doesn't sound like much code, that's because it's not. After you add the interface, you only need to change one line of code in the Bee class and three lines of code in the Queen class.



Exercise Solution

Your job was to modify the Beehive Management System to use the **IWorker** interface instead of the abstract **Bee** class any time it needs to reference a worker. You had to add the **IWorker** interface, then modify the **Bee** and **Queen** classes. This didn't take a lot of code—that's because using interfaces doesn't require a lot of extra code.

First you added the **IWorker** interface to your project

```
interface IWorker
{
    string Job { get; }
    void WorkTheNextShift();
}
```

Then you modified **Bee** to implement the **IWorker** interface

```
abstract class Bee : IWorker
{
    /* The rest of the class stays the same */
}
```

And finally, you modified **Queen** to use **IWorker** references instead of **Bee** references

```
class Queen : Bee
{
    private IWorker[] workers = new IWorker[0];

    private void AddWorker(IWorker worker)
    {
        if (unassignedWorkers >= 1)
        {
            unassignedWorkers--;
            Array.Resize(ref workers, workers.Length + 1);
            workers[workers.Length - 1] = worker;
        }
    }

    private string WorkerStatus(string job)
    {
        int count = 0;
        foreach (IWorker worker in workers)
            if (worker.Job == job) count++;
        string s = "s";
        if (count == 1) s = "";
        return $"{count} {job} bee{s}";
    }

    /* Everything else in the Queen class stays the same */
}
```

Any class can implement ANY interface as long as it keeps the promise of implementing the interface's methods and properties.

Try modifying **WorkerStatus to change the **IWorker** in the foreach loop back to a **Bee**:**
foreach (**Bee** worker in workers)
Then run your code—it works just fine! Now try changing it to a **NectarCollector. This time you get a **System.InvalidCastException**. Why do you think that happens?**

there are no
Dumb Questions

Q: When I put a property in an interface, it looks just like an automatic property. Does that mean I can only use automatic properties when I implement an interface?

A: No, not at all. It's true that a property inside an interface looks very similar to an automatic property—like the Job property in the IWorker interface on the next page—but they're definitely not automatic properties. You could implement the Job property like this:

```
public Job {
    get; private set;
}
```

You need that private set, because automatic properties require you to have both a set and a get (even if they're private). Alternatively, you could implement it like this:

```
public Job {
    get {
        return "Egg Care";
    }
}
```

and the compiler will be perfectly happy with that. You can also add a setter—the interface requires a get, but it doesn't say you can't have a set, too. (If you use an automatic property to implement it, you can decide for yourself whether you want the set to be private or public.)

Q: Isn't it weird that there are no access modifiers in my interfaces? Should I be marking methods and properties public?

A: You don't need the access modifiers because everything in an interface is automatically public by default. If you have an interface that has this:

```
void Honk();
```

It says that you need a public void method called Honk, but it doesn't say what that method needs to do. It can do anything at all—no matter what it does, the code will compile as long as some method is there with the right signature.

Does that look familiar? That's because we've seen it before—in abstract classes, back in Chapter 6. When you declare methods or properties in an interface without bodies, they're **automatically public and abstract**, just like the abstract members that you used in your abstract classes. They work just like any other abstract methods or properties—because even though you're not using the **abstract** keyword, it's implied. That's why every class that implements an interface **must implement every member**.

The folks who designed C# could have made you mark each of those members public and abstract, but it would have been redundant. So they made the members public and abstract by default to make it all clearer.

Everything in a public interface is automatically public, because you'll use it to define the public methods and properties of any class that implements it.

we shouldn't be using a string for job

The IWorker's Job property is a hack

The Beehive Management System uses the Worker.Job property like this: `if (worker.Job == job)`

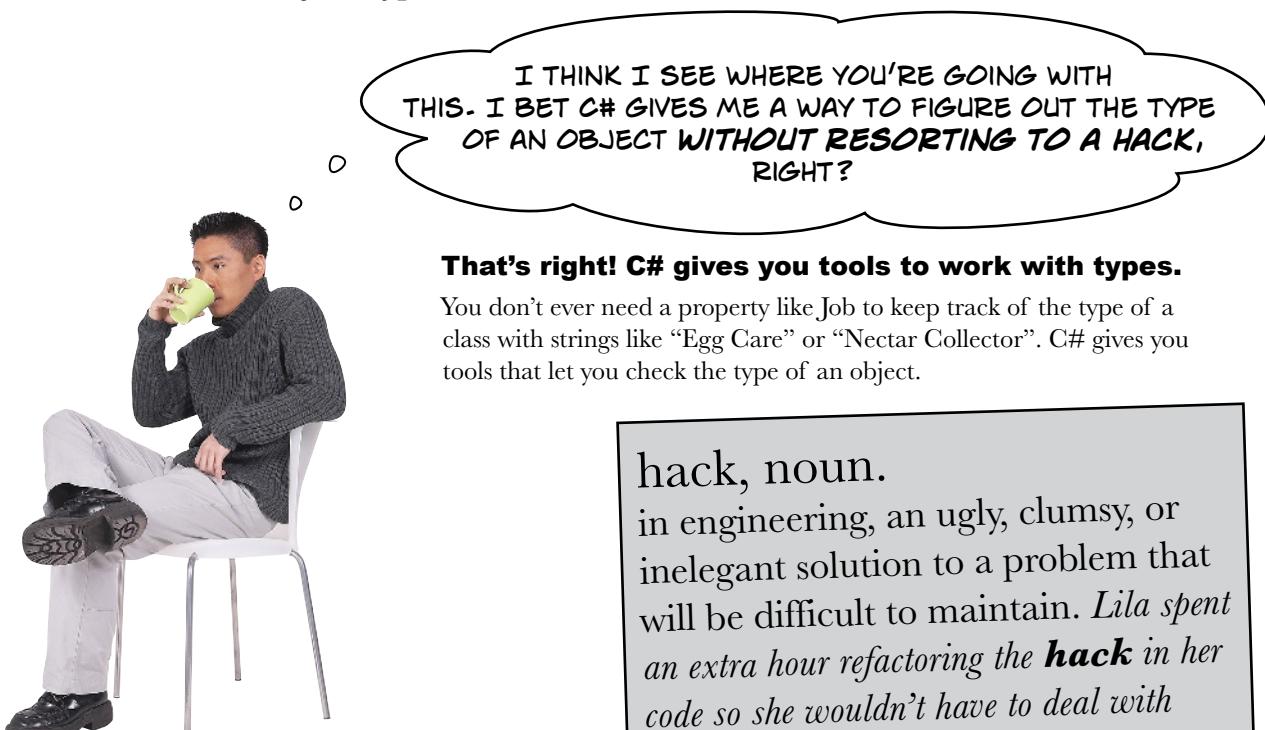
Does something seem a bit odd about that? It does to us. We think it's a **hack**, or a clumsy, inelegant solution. Why do we think the Jobproperty is a hack? Imagine what would happen if you had a typo like this:

```
class EggCare : Bee {  
    public EggCare(Queen queen) : base("Egg Crae") ←  
        // Oops! Now we've got a bug in the EggCare class,  
        // even though the rest of the class is the same.  
}
```

We misspelled "Egg Care"—that's a mistake anyone could make! Can you imagine how hard it would be to track down the bugs that this simple typo would cause?

Now the code has no way to figure out if a Worker reference is pointing to an instance of EggCare. That would be a really nasty bug to try to fix. So we know this code is error-prone...but how is it a hack?

We've talked about **separation of concerns**: all of the code to address a specific problem should be kept together. The Job property *violates the principle of separation of concerns*. If we have a Worker reference, we shouldn't need to check a string to figure out whether it points to an EggCare object or a NectarCollector object. The Job property returns "Egg Care" for an EggCare object and "Nectar Collector" for a NectarCollector object and is *only* used to check the object's type. But we're already keeping track of that information: **it's the object's type**.



Use "is" to check the type of an object

What would it take to get rid of the Job property hack? Right now the Queen has her `workers` array, which means that all she can get is an `IWorker` reference. She uses the `Job` property to figure out which workers are EggCare workers and which ones are NectarCollectors:

```
foreach (IWorker worker in workers) {
    if (worker.Job == "Egg Care") {
        WorkNightShift((EggCare)worker);
    }

    void WorkNightShift(EggCare worker) {
        // Code to work the night shift
    }
}
```

We just saw how that code will fail miserably if we accidentally type “Egg Crae” instead of “Egg Care”. And if you set a HoneyManufacturer’s Job to “Egg Care” accidentally, you’ll get one of those `InvalidCastException` errors. It would be great if the compiler could detect problems like that as soon as we write them, just like we use private or abstract members to get it to detect other kinds of problems.

C# gives us a tool to do exactly that: we can use the **`is` keyword** to check an object’s type. If you have an object reference, you can **use `is` to find out if it’s a specific type**:

`objectReference is ObjectType newVariable`

If the object that `objectReference` is pointing to has `ObjectType` as its type, then it returns true and creates a new reference called `newVariable` with that type.

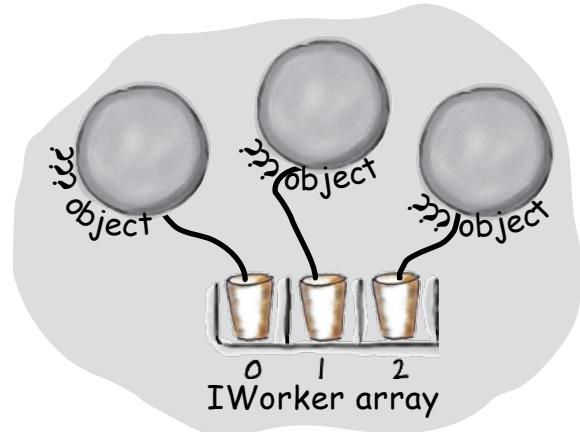
So if the Queen wants to find all of her EggCare workers and have them work a night shift, she can use the `is` keyword:

```
foreach (IWorker worker in workers) {
    if (worker is EggCare eggCareWorker) {
        WorkNightShift(eggCareWorker);
    }
}
```

The `if` statement in this loop uses `is` to check each `IWorker` reference. Look closely at the conditional test:

```
worker is EggCare eggCareWorker
```

If the object referenced by the `worker` variable is an EggCare object, that test returns true, and the `is` statement assigns the reference to a new EggCare variable called `eggCareWorker`. This is just like casting, but the `is` statement is **doing the casting for you safely**.



The **`is` keyword** returns true if an object matches a type, and can declare a variable with a reference to that object.

Use "is" to access methods in a subclass

Do this!

Let's pull together everything we've talked about so far into a new project by creating a simple class model with Animal at the top, Hippo and Canine classes that extend Animal, and a Wolf class that extends Canine.

Create a new console app and **add these Animal, Hippo, Canine, and Wolf classes** to it:

```
abstract class Animal
{
    public abstract void MakeNoise();
```

← The abstract class Animal is at the top of the hierarchy.

```
class Hippo : Animal
{
    public override void MakeNoise()
    {
        Console.WriteLine("Grunt.");
    }

    public void Swim()
    {
        Console.WriteLine("Splash! I'm going for a swim!");
    }
}
```

The Hippo subclass overrides the abstract MakeNoise method, and adds its own Swim method that has nothing to do with the Animal class.

```
abstract class Canine : Animal
{
    public bool BelongsToPack { get; protected set; } = false;
```

← The abstract Canine class extends Animal. It has its own abstract property, BelongsToPack.

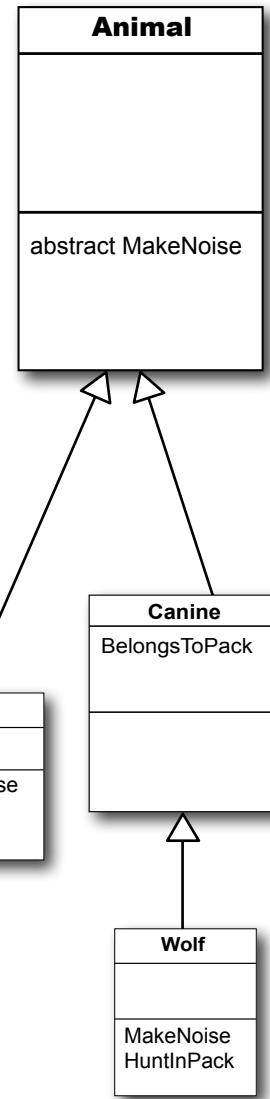
```
class Wolf : Canine
{
    public Wolf(bool belongsToPack)
    {
        BelongsToPack = belongsToPack;
    }

    public override void MakeNoise()
    {
        if (BelongsToPack)
            Console.WriteLine("I'm in a pack.");
        Console.WriteLine("Aroooooo!");
    }

    public void HuntInPack()
    {
        if (BelongsToPack)
            Console.WriteLine("I'm going hunting with my pack!");
        else
            Console.WriteLine("I'm not in a pack.");
    }
}
```

← The Wolf class extends Canine, and adds its own HuntInPack method.

The HuntInPack method is only part of the Wolf class. It's not inherited from a superclass.



Next, fill in the Main method. Here's what it does:

- ★ It creates an array of Hippo and Wolf objects, then uses a foreach loop to go through each of them.
- ★ It uses the Animal reference to call the MakeNoise method.
- ★ If it's a Hippo, the Main method calls its Hippo.Swim method.
- ★ If it's a Wolf, the Main method calls its Wolf.HuntInPack method.

The problem is that if you have an Animal reference pointing to a Hippo object, you can't use it to call Hippo.Swim:

```
Animal animal = new Hippo();
animal.Swim(); // <-- this line will not compile!
```

It doesn't matter that your object is a Hippo. If you're using an Animal variable, you can only access the fields, methods, and properties of Animal.

Luckily, there's a way around this. If you're 100% sure that you have a Hippo object, then you can **cast your Animal reference to a Hippo**—and then you can access its Hippo.Swim method:

```
Hippo hippo = (Hippo)animal;
hippo.Swim(); // It's the same object, but now you can call the Hippo.Swim method.
```

Here's **the Main method that uses the is keyword** to call Hippo.Swim or Wolf.HuntInPack:

```
class Program
{
    static void Main(string[] args)
    {
        Animal[] animals =
        {
            new Wolf(false),
            new Hippo(),
            new Wolf(true),
            new Wolf(false),
            new Hippo()
        };

        foreach (Animal animal in animals)
        {
            animal.MakeNoise();
            if (animal is Hippo hippo)
            {
                hippo.Swim();
            }

            if (animal is Wolf wolf)
            {
                wolf.HuntInPack();
            }

            Console.WriteLine();
        }
    }
}
```

This foreach loop iterates through the “animals” array. It needs to declare a variable of type Animal to match the array type, but that reference won't let us access Hippo.Swim or Wolf.HuntInPack.

This if statement uses the “is” keyword to check if the animal reference is a Hippo or Wolf, and then safely casts it to the hippo or wolf variable so it can call the methods specific to the subclass.

Take a few minutes and use the debugger to really understand what's going on here. Put a breakpoint on the first line of the foreach loop; add watches for animal, hippo, and wolf; and step through it.

What if we want different animals to swim or hunt in packs?

Did you know that lions are pack hunters? Or that tigers can swim? And what about dogs, which hunt in packs AND swim? If we want to add the Swim and HuntInPack methods to all of the animals in our zoo simulator model that need them, the foreach loop is just going to get longer and longer.

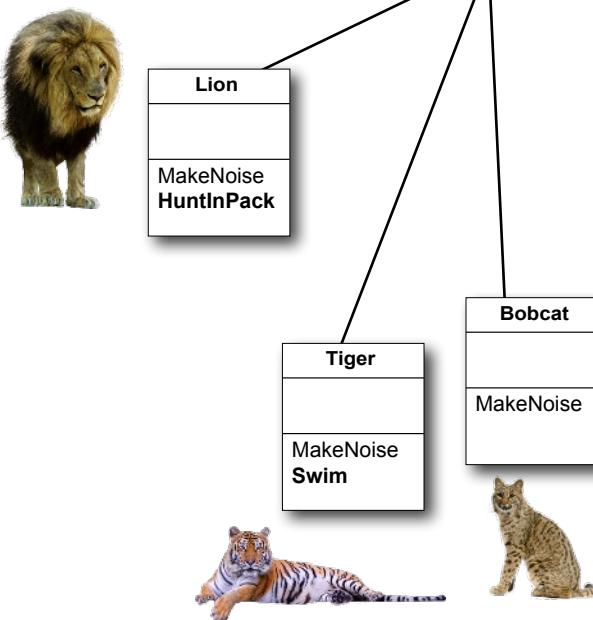
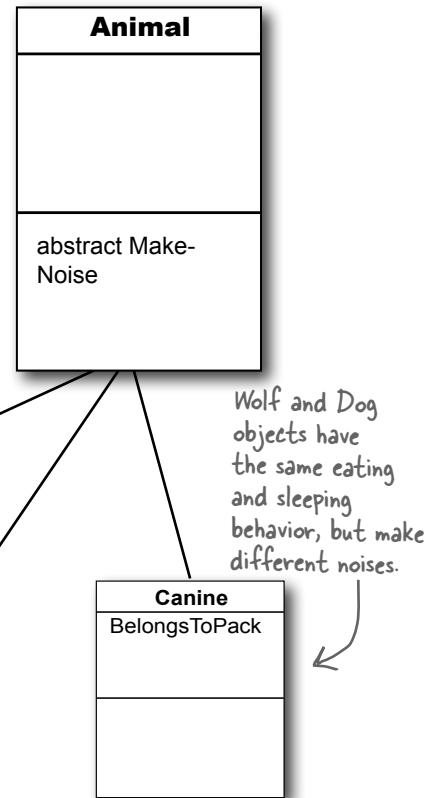
The beauty of defining an abstract method or property in a base class and overriding it in a subclass is that **you don't need to know anything about the subclass** to use it. You can add all of the Animal subclasses you want, and this loop will still work:

```
foreach (Animal animal in animals) {  
    animal.MakeNoise();  
}
```

The MakeNoise method will **always be implemented by the object**.

In fact, you can treat it like a **contract** that the compiler enforces.

So is there a way to treat the HuntInPack and Swim methods like contracts too, so we can use more general variables with them—just like we do with the Animal class?



Use interfaces to work with classes that do the same job

Classes that swim have a `Swim` method, and classes that hunt in packs have a `HuntInPack` method. OK, that's a good start. Now we want to write code that works with objects that swim or hunt in packs—and that's where interfaces shine. Let's use the **interface keyword** to define two interfaces and **add an abstract member** to each interface:

```
interface ISwimmer {
    void Swim();
}

interface IPackHunter {
    void HuntInPack();
}
```

← Add this!

Next, **make the Hippo and Wolf classes implement the interfaces** by adding an interface to the end of each class declaration. Use a **colon** (`:`) to implement an interface, just like you do when you're extending a class. If it's already extending a class, you just add a comma after the superclass and then the interface name. Then you just need to make sure the class **implements all the interface members**, or you'll get a compiler error.

```
class Hippo : Animal, ISwimmer {
    /* The code stays exactly the same - it MUST include the Swim method */
}

class Wolf : Canine, IPackHunter {
    /* The code stays exactly the same - it MUST include the HuntInPack method */
}
```

Use the "is" keyword to check if the Animal is a swimmer or pack hunter

You can use the **is keyword** to check if a specific object implements an interface—and it works no matter what other classes that object implements. If the `animal` variable references an object that implements the `ISwimmer` interface, then `animal is ISwimmer` will return true and you can safely cast it to an `ISwimmer` reference to call its `Swim` method:

```
foreach (Animal animal in animals)
{
    animal.MakeNoise();
    if (animal is ISwimmer swimmer)
    {
        swimmer.Swim();
    }
    if (animal is IPackHunter hunter)
    {
        hunter.HuntInPack();
    }
    Console.WriteLine();
}
```

We're using the "is" keyword just like we did earlier, but this time we're using it with interfaces. It still works the same way.

What would your code look like if you had 20 different Animal subclasses that swim? You'd need 20 different `(animal is...)` statements that cast `animal` to each individual subclass to call the `Swim` method. By using `ISwimmer`, we only have to check it once.

"is" prevents unsafe conversions

Safely navigate your class hierarchy with "is"

When you did the exercise to replace Bee with IWorker in the Beehive Management System, were you able to get it to throw the InvalidCastException? **Here's why it threw the exception.**



You can safely convert a NectarCollector reference to an IWorker reference.

All NectarCollectors are Bees (meaning they extend the Bee base class), so you can always use the = operator to take a reference to a NectarCollector and assign it to a Bee variable.

```
HoneyManufacturer lily = new HoneyManufacturer();  
Bee hiveMember = lily;
```

And since Bee implements the IWorker interface, you can safely convert it to an IWorker reference too.

```
HoneyManufacturer daisy = new HoneyManufacturer();  
IWorker worker = daisy;
```

Those type conversions are safe: they'll never throw an InvalidCastException because they only assign more specific objects to variables with more general types *in the same class hierarchy*.



You can't safely convert a Bee reference to a NectarCollector reference.

You can't safely go in the other direction—converting a Bee to a NectarCollector—because not all Bee objects are instances of NectarCollector. A HoneyManufacturer is *definitely not* a NectarCollector. So this:

```
IWorker pearl = new HoneyManufacturer();  
NectarCollector irene = (NectarCollector)pearl;
```

is an **invalid cast** that tries to cast an object to a variable that doesn't match its type.



The "is" keyword lets you convert types safely.

Luckily, **the is keyword is safer than casting with parentheses**. It lets you check that the type matches, and only casts the reference to a new variable if the types match.

```
if (pearl is NectarCollector irene) {  
    /* Code that uses a NectarCollector object */  
}
```

This code will never throw an InvalidCastException because it only executes the code that uses a NectarCollector object if `pearl` is a NectarCollector.

C# has another tool for safe type conversion: the “as” keyword

C# gives you another tool for safe casting: the **as keyword**. It also does safe type conversion. Here’s how it works. Let’s say you have an IWorker reference called `pearl`, and you want to safely cast it to a NectarCollector variable `irene`. You can convert it safely to a NectarCollector like this:

```
NectarCollector irene = pearl as NectarCollector;
```

If the types are compatible, this statement sets the `irene` variable to reference the same object as the `pearl` variable. If the type of the object doesn’t match the type of the variable, it doesn’t throw an exception. Instead, it just **sets the variable to null**, which you can check with an `if` statement:

```
if (irene != null) {
    /* Code that uses a NectarCollector object */
}
```



Watch it!

The “is” keyword works differently in very old versions of C#.

The `is` keyword has been in C# for a long time, but it wasn’t until C# 7.0 was released in 2017 that `is` let you declare a new variable. So if you’re using Visual Studio 2015, you won’t be able to do this: `if (pearl is NectarCollector irene) { ... }`

Instead, you’ll need to use the `as` keyword to do the conversion, then test the result to see if it’s `null`:

```
NectarCollector irene = pearl as NectarCollector;
if (irene != null) { /* code that uses the irene reference */ }
```



Sharpen your pencil

The array on the left uses types from the Bee class model. Two of these types won’t compile—cross them out. On the right are three statements that use the `is` keyword. Write down which values of `i` would make each of them evaluate to `true`.

```
IWorker[] bees = new IWorker[8];
bees[0] = new HiveDefender();
bees[1] = new NectarCollector();
bees[2] = bees[0] as IWorker;
bees[3] = bees[1] as NectarCollector;
bees[4] = IDefender;
bees[5] = bees[0];
bees[6] = bees[0] as Object;
bees[7] = new IWorker();
```

1. (bees[i] is IDefender)

2. (bees[i] is IWorker)

3. (bees[i] is Bee)

Use upcasting and downcasting to move up and down a class hierarchy

Class diagrams typically have the base class at the top, its subclasses below it, their subclasses below them, etc. The higher a class is in the diagram, the more abstract it is; the lower the class is in the diagram, the more concrete it is. “Abstract higher, concrete lower” isn’t a hard-and-fast rule—it’s a **convention** that makes it easier for us to see at a glance how our class models work.

In Chapter 6 we talked about how you can always use a subclass in place of the base class it inherits from, but you can’t always use a base class in place of a subclass that extends it. You can also think about this another way: in a sense, you’re **moving up or down the class hierarchy**. For example, if you start with this:

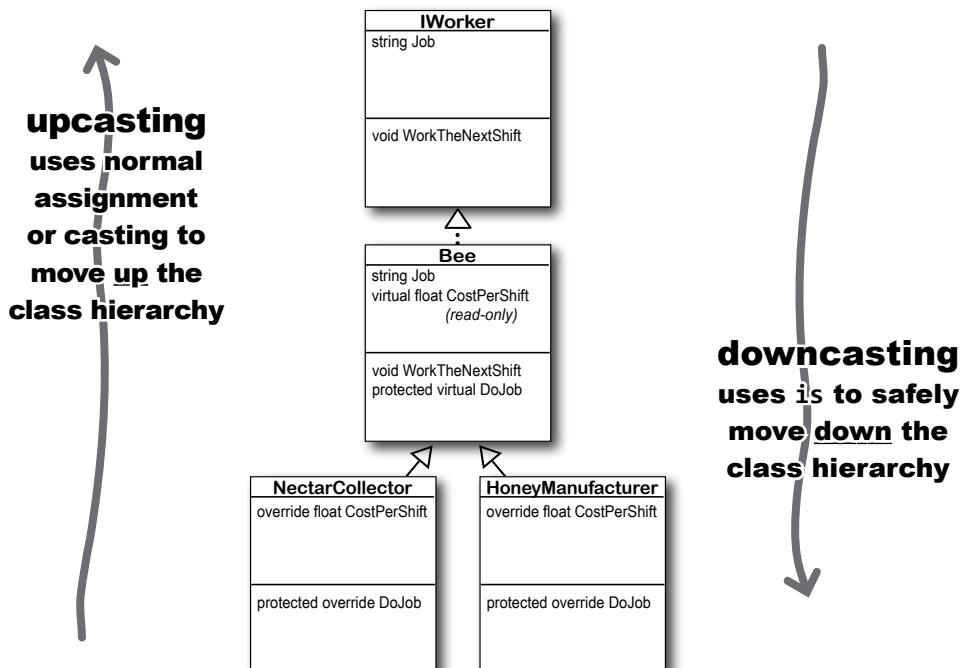
```
NectarCollector ida = new NectarCollector();
```

You can use the = operator to do normal assignment (for superclasses) or casting (for interfaces). That’s like **moving up** the class hierarchy. This is called **upcasting**:

```
// Upcast the NectarCollector to a Bee  
Bee beeReference = ida;  
  
// This upcast is safe because all Bees are IWorkers  
IWorker worker = (IWorker)beeReference;
```

And you can navigate in the other direction by using the is operator to safely **move down** the class hierarchy. This is called **downcasting**:

```
// Downcast the IWorker to NectarCollector  
if (worker is NectarCollector rose) { /* code that uses the rose reference */ }
```



Appliance

ConsumePower

CoffeeMaker

CoffeeLeft

FillWithWater
StartBrewing**Oven**

Capacity

Preheat
CookFood

A quick example of upcasting

If you're trying to figure out how to cut down your energy bill each month, you don't really care what each of your appliances does—you only care that they consume power. So if you were writing a program to monitor your electricity consumption, you'd probably just write an Appliance class. But if you needed to distinguish a coffee maker from an oven, you'd have to build a class hierarchy and add the methods and properties that are specific to a coffee maker or oven to your CoffeeMaker and Oven classes, which would inherit from an Appliance class that has their common methods and properties.

Then you could write a method to monitor the power consumption:

```
void MonitorPower(Appliance appliance) {
    /* code to add data to a household
       power consumption database */
}
```

If you wanted to use that method to monitor the power consumption for a coffee maker, you could create an instance of CoffeeMaker and pass its reference directly to the method:

```
CoffeeMaker mrCoffee = new CoffeeMaker();
MonitorPower(mrCoffee);
```

This is a great example of upcasting. Even though the MonitorPower method takes a reference to an Appliance object, you can pass it the mrCoffee reference because CoffeeMaker is a subclass of Appliance.



Sharpen your pencil Solution

The array on the left uses types from the Bee class model. Two of these types won't compile—cross them out. On the right are three statements that use the `is` keyword. Write down which values of `i` would make each of them evaluate to `true`.

```
IWorker[] bees = new IWorker[8];
bees[0] = new HiveDefender();
bees[1] = new NectarCollector();
bees[2] = bees[0] as IWorker;
bees[3] = bees[1] as NectarCollector;
bees[4] = IDefender;
bees[5] = bees[0];
bees[6] = bees[0] as Object;
bees[7] = new IWorker();
```

Elements 0, 2, 1. (`bees[i]` is `IDefender`)

and b in the array
all point to the → 0, 2, and b
same HiveDefender
object.

2. (`bees[i]` is `IWorker`)

0, 1, 2, 3, 5, 6

All of the objects
extend Bee, and Bee
implements IWorker,
so they're all Bees
and IWorkers.

3. (`bees[i]` is `Bee`)

0, 1, 2, 3, 5, 6

This line casts
the IWorker to a
NectarCollector
but then stores
it as an IWorker
reference again.

Upcasting turns your CoffeeMaker into an Appliance

When you substitute a subclass for a base class—like substituting a CoffeeMaker for an Appliance, or a Hippo for an Animal—it's called **upcasting**. It's a really powerful tool to use when you build class hierarchies. The only drawback to upcasting is that you can only use the properties and methods of the base class. In other words, when you treat a CoffeeMaker like an Appliance, you can't tell it to make coffee or fill it with water. You *can* tell whether or not it's plugged in, since that's something you can do with any Appliance (which is why the PluggedIn property is part of the Appliance class).

1 Let's create some objects.

Let's start by creating instances of the CoffeeMaker and Oven classes as usual:

```
CoffeeMaker misterCoffee = new CoffeeMaker();  
Oven oldToasty = new Oven();
```

2 What if we want to create an array of Appliances?

You can't put a CoffeeMaker in an Oven[] array, and you can't put an Oven in a CoffeeMaker[] array. You *can* put both of them in an Appliance[] array:

```
Appliance[] kitchenWare = new Appliance[2];  
kitchenWare[0] = misterCoffee;  
kitchenWare[1] = oldToasty;
```

↗ You can use upcasting to create an array of Appliances that can hold both CoffeeMakers and Ovens.

3 But you can't treat just any Appliance like an Oven.

When you've got an Appliance reference, you can **only** access the methods and properties that have to do with appliances. You **can't** use the CoffeeMaker methods and properties through the Appliance reference **even if you know it's really a CoffeeMaker**. So these statements will work just fine, because they treat a CoffeeMaker object like an Appliance:

```
Appliance powerConsumer = new CoffeeMaker();  
powerConsumer.ConsumePower();
```

But as soon as you try to use it like a CoffeeMaker:

```
powerConsumer.StartBrewing();
```

This line won't compile because powerConsumer is an Appliance reference, so it can only be used to do Appliance things.

your code won't compile, and the IDE will display an error:

 CS1061 'Appliance' does not contain a definition for 'StartBrewing' and no accessible extension method 'StartBrewing' accepting a first argument of type 'Appliance' could be found (are you missing a using directive or an assembly reference?)

Once you upcast from a subclass to a base class, you can only access the methods and properties that **match the reference** that you're using to access the object.

powerConsumer is an Appliance reference pointing to a CoffeeMaker object.



Downcasting turns your Appliance back into a CoffeeMaker

Upcasting is a great tool, because it lets you use a CoffeeMaker or an Oven anywhere you just need an Appliance. But it's got a big drawback—if you're using an Appliance reference that points to a CoffeeMaker object, you can only use the methods and properties that belong to Appliance. That's where **downcasting** comes in: that's how you take your **previously upcast reference** and change it back. You can figure out if your Appliance is really a CoffeeMaker using the **is** keyword, and if it is you can convert it back to a CoffeeMaker.

1 We'll start with the CoffeeMaker we already upcast.

Here's the code that we used:

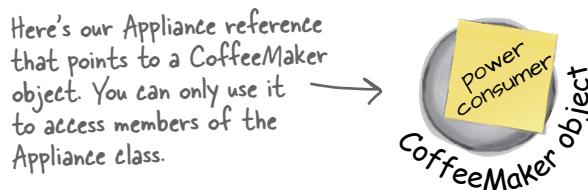
```
Appliance powerConsumer = new CoffeeMaker();
powerConsumer.ConsumePower();
```

2 What if we want to turn the Appliance back into a CoffeeMaker?

Let's say we're building an app that looks in an array of Appliance references so it can make our CoffeeMaker start brewing. We can't just use our Appliance reference to call the CoffeeMaker method:

```
Appliance someAppliance = appliances[5];
someAppliance.StartBrewing()
```

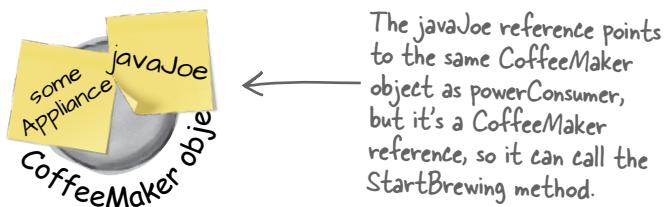
That statement won't compile—you'll get that “Appliance” does not contain a definition for ‘StartBrewing’” compiler error because StartBrewing is a member of CoffeeMaker but you're using an Appliance reference.



3 But since we know it's a CoffeeMaker, let's use it like one.

The **is** keyword is the first step. Once you know that you've got an Appliance reference that's pointing to a CoffeeMaker object, you can use **is** to downcast it. That lets you use the CoffeeMaker class's methods and properties. Since CoffeeMaker inherits from Appliance, it still has its Appliance methods and properties.

```
if (someAppliance is CoffeeMaker javaJoe) {
    javaJoe.StartBrewing();
}
```

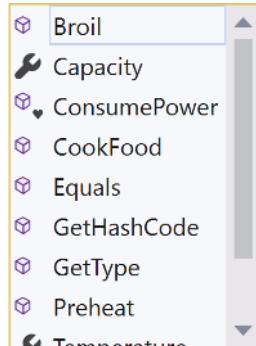


Upcasting and downcasting work with interfaces, too

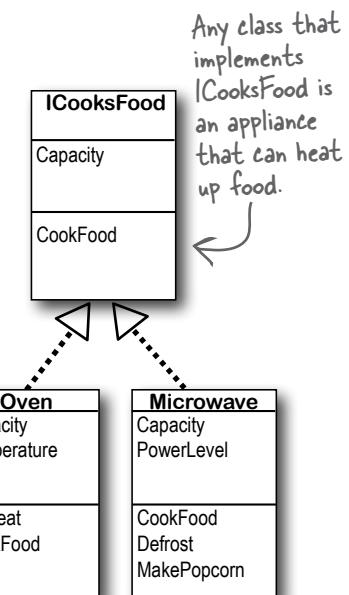
Interfaces work really well with upcasting and downcasting. Let's add an ICooksFood interface for any class that can heat up food. Next, we'll add a Microwave class—both Microwave and Oven implement the ICooksFood interface. Now a reference to an Oven object can be an ICooksFood reference, a Microwave reference, or an Oven reference. That means we have three different types of references that could point to an Oven object—and each of them can access different members, depending on the reference's type. Luckily, the IDE's IntelliSense can help you figure out exactly what you can and can't do with each of them:

```
Oven misterToasty = new Oven();
misterToasty.
```

When you have an
Oven reference,
you can access
all of the Oven
members.



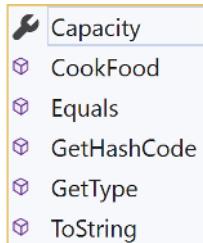
As soon as you type the dot, the IntelliSense window will pop up with a list of all of the members you can use. misterToasty is an Oven reference pointing to an Oven object, so it can access all of the methods and properties. It's the most specific type, so you can only point it at Oven objects.



To access ICooksFood interface members, convert it to an ICooksFood reference:

```
if (misterToasty is ICooksFood cooker) {
    cooker.
```

An ICooksFood
reference can
only access
members that
are part of
the interface.

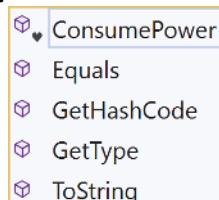


cooker is an ICooksFood
reference pointing to that same
Oven object. It can only access
ICooksFood members, but it can
also point to a Microwave object.

This is the same Oven class that we used earlier, so it also extends the Appliance base class. If you use an Appliance reference to access the object, you'll only see members of the Appliance class:

```
if (misterToasty is Appliance powerConsumer)
    powerConsumer.
```

Appliance only has one
member, ConsumePower,
so that's all you see in
the dropdown.



powerConsumer is an Appliance reference. It only lets you get to the public fields, methods, and properties in Appliance. It's more general than an Oven reference (so you could point it at a CoffeeMaker object if you wanted to).

Three different
references that
point to the same
object can access
different methods
and properties,
depending on the
reference's type.

there are no
Dumb Questions

Q: So back up—I think you told me that I can always upcast but I can’t always downcast. Why?

A: Because an upcast won’t work if you’re trying to set an object equal to a class that it doesn’t inherit from or an interface that it doesn’t implement. The compiler can figure out immediately that you didn’t upcast properly and give you an error. When we say “you can always upcast but can’t always downcast” it’s just like saying “every oven is an appliance but not every appliance is an oven.”

Q: I read online that an interface is like a contract, but I don’t really get why. What does that mean?

A: Yes, many people like to say that an interface is like a contract. (“How is an interface like a contract?” is a really common question during job interviews.) When you make your class implement an interface, you’re telling the compiler that you promise to put certain methods into it. The compiler will hold you to that promise. That’s like a court forcing you to stick to the terms of a contract. If that helps you understand interfaces, then definitely think of them that way.

But we think that it’s easier to remember how interfaces work if you think of an interface as a kind of checklist. The compiler runs through the checklist to make sure that you actually put all of the methods from the interface into your class. If you didn’t, it’ll bomb out and not let you compile.

Q: Why would I want to use an interface? It seems like it’s just adding restrictions, without actually changing my class at all.

A: Because when your class implements an interface, you can use that interface as a type to declare a reference that can point to any instance of a class that implements it. That’s really useful to you—it lets you create one reference type that can work with a whole bunch of different kinds of objects.

Here’s a quick example. A horse, an ox, a mule, and a steer can all pull a cart. In our zoo simulator, Horse, Ox, Mule, and Steer would all be different classes. Let’s say you had a cart-pulling ride in your zoo, and you wanted to create an array of any animal that could pull carts around. Uh-oh—you can’t just create an array that will hold all of those. If they all inherited from the same base class you could create an array of those, but it turns out that they don’t. So what’ll you do?

That’s where interfaces come in handy. You can create an IPuller interface that has methods for pulling carts around. Then you can declare your array like this:

```
IPuller[] pullerArray;
```

Now you can put a reference to any animal you want in that array, as long as it implements the IPuller interface.

An interface is like a checklist that the compiler runs through to make sure your class implemented a certain set of methods.

Interfaces can inherit from other interfaces

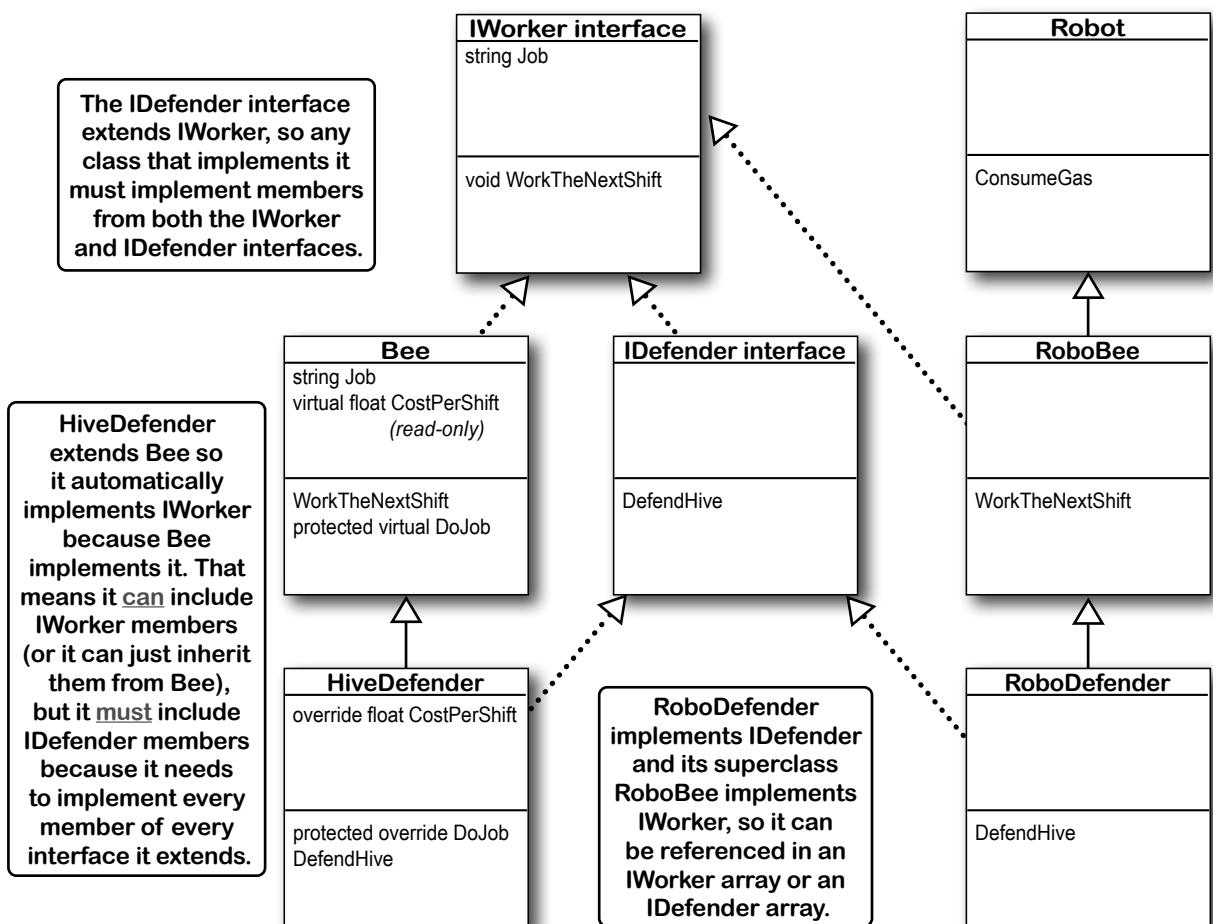
As we've mentioned, when one class inherits from another, it gets all of the methods and properties from the base class. **Interface inheritance** is simpler. Since there's no actual method body in any interface, you don't have to worry about calling base class constructors or methods. The inherited interfaces **accumulate all of the members** of the interfaces that they extend.

So what does this look like in code? Let's add an IDefender interface that inherits from IWorker:

```
interface IDefender : IWorker {
    void DefendHive();
}
```

← Use a colon (:) to make an interface extend another interface.

When a class implements an interface, it must implement every property and method in that interface. If that interface inherits from another one, then all of *those* properties and methods need to be implemented, too. So any class that implements IDefender not only must implement all of the IDefender members, but also all of the IWorker members. Here's a class model that includes IWorker and IDefender, and **two separate hierarchies** that implement them.





Exercise

Create a new console app with classes that implement the `IClown` interface. Can you figure out how to get the code at the bottom to build?

- Start with the `IClown` interface you created earlier:

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}
```

- Extend `IClown` by creating a new interface called `IScaryClown` that extends `IClown`. It should have a string property called `ScaryThingIHave` with a get accessor but no set accessor, and a void method called `ScareLittleChildren`.

- Create these classes that implement the interfaces:

- A class called `FunnyFunny` that implements `IClown`. It uses a `private` string variable called `funnyThingIHave` to store a funny thing. The `FunnyThingIHave` getter uses `funnyThingIHave` as a backing field. Use a constructor that takes a parameter and uses it to set the private field. The `Honk` method prints: “*Hi kids! I have a*” followed by the funny thing and a period.
- A class called `ScaryScary` that implements `IScaryClown`. It uses a private variable to store an integer called `scaryThingCount`. The constructor sets both the `scaryThingCount` field and the `funnyThingIHave` field that `ScaryScary` inherited from `FunnyFunny`. The `ScaryThingIHave` getter returns a string with the number from the constructor followed by “*spiders*”. The `ScareLittleChildren` method writes “*Boo! Gotcha! Look at my...!*” to the console, replacing “...” with the clown’s scary thing.

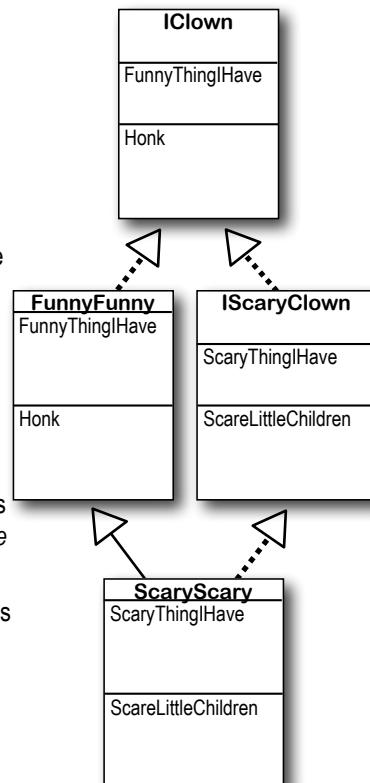
- Here's some new code for the `Main` method—but it's not working. Can you figure out how to fix it so it builds and prints messages to the console?

```
static void Main(string[] args)
{
    IClown fingersTheClown = new ScaryScary("big red nose", 14);
    fingersTheClown.Honk();
    IScaryClown iScaryClownReference = fingersTheClown;
    iScaryClownReference.ScareLittleChildren();
}
```

Before you run the code, write down the output that the `Main` method will print to the console (once you fix it):

.....

Then run the code and see if you got the answer right.



YOU BETTER GET THIS ONE
RIGHT...OR ELSE!

Fingers the →
Clown is scary.





Exercise Solution

Create a new console app with classes that implement the IClown interface. Can you figure out how to get the code at the bottom to build?

The IScaryClown interface extends IClown and adds a property and a method:

```
interface IScaryClown : IClown
{
    string ScaryThingIHave { get; }
    void ScareLittleChildren();
}
```

The IScaryClown interface inherits from the IClown interface. That means any class that implements IScaryClown not only needs to have a ScaryThingIHave property and a ScareLittleChildren method, but also a FunnyThingIHave property and a Honk method.

The FunnyFunny class implements the IClown interface and uses a constructor to set a backing field:

```
class FunnyFunny : IClown
{
    private string funnyThingIHave;
    public string FunnyThingIHave { get { return funnyThingIHave; } }

    public FunnyFunny(string funnyThingIHave)
    {
        this.funnyThingIHave = funnyThingIHave;
    }

    public void Honk()
    {
        Console.WriteLine($"Hi kids! I have a {funnyThingIHave}.");
    }
}
```

These are just like the constructors and backing fields we used in Chapter 5.

The ScaryScary class extends the FunnyFunny class and implements the IScaryClown interface. Its constructor uses the base keyword to call the FunnyFunny constructor to set the private backing field:

```
class ScaryScary : FunnyFunny, IScaryClown
{
    private int scaryThingCount;

    public ScaryScary(string funnyThing, int scaryThingCount) : base(funnyThing)
    {
        this.scaryThingCount = scaryThingCount;
    }

    public string ScaryThingIHave { get { return $"{scaryThingCount} spiders"; } }

    public void ScareLittleChildren()
    {
        Console.WriteLine($"Boo! Gotcha! Look at my {ScaryThingIHave}!");
    }
}
```

FunnyFunny.funnyThingIHave is a private field, so ScaryScary can't access it—it needs to use the base keyword to call the FunnyFunny constructor.

To fix the Main method, replace lines 3 and 4 of the method with these lines that use the **is** operator:

```
if (fingersTheClown is IScaryClown iScaryClownReference)
{
    iScaryClownReference.ScareLittleChildren();
}
```

You can set a FunnyFunny reference equal to a ScaryScary object because ScaryScary inherits from FunnyFunny. You can't set any IScaryClown reference to just any clown, because you don't know if that clown is scary. That's why you need to use the **is** keyword.



I KEEP NOTICING THE IDE ASKING ME IF I WANT TO MAKE FIELDS READONLY. IS THAT SOMETHING I SHOULD DO?

Absolutely! Making fields read-only helps prevent bugs.

Go back to the ScaryScary.scaryThingCount field—the IDE put dots underneath the first two letters of the field name. Hover over the dots to get the IDE to pop up a window:

```
private int ...scaryThingCount;
```

💡 (field) int ScaryScary.scaryThingCount
 Make field readonly
[Show potential fixes \(Alt+Enter or Ctrl+.\)](#)

Press **Ctrl+.** to pop up a list of actions, and choose “**Add readonly modifier**” to add the **readonly keyword** to the declaration:

```
private readonly int scaryThingCount;
```

Now the field can only be set when it’s declared or in the constructor. If you try to change its value anywhere else in the method, you’ll get a compiler error:

✖ CS0191 A readonly field cannot be assigned to (except in a constructor or a variable initializer)

The **readonly** keyword...just another way C# helps you keep your data safe.

The **readonly** keyword

An important reason that we use encapsulation is to prevent one class from accidentally overwriting another class’s data. What’s preventing a class from overwriting its own data? The “**readonly**” keyword can help with that. Any field that you mark **readonly** can only be modified in its declaration or in the constructor.

there are no
Dumb Questions

Q: Why would I want to use an interface instead of just writing all of the methods I need directly into my class?

A: When you use interfaces, you still write methods in your classes. Interfaces let you group those classes by the kind of work they do. They help you be sure that every class that's going to do a certain kind of work does it using the same methods. The class can do the work however it needs to, and because of the interface, you don't need to worry about how it gets the job done.

Here's an example: you can have a Truck class and a Sailboat class that implement ICarryPassenger. Say the ICarryPassenger interface stipulates that any class that implements it has to have a ConsumeEnergy method. Your program could use them both to carry passengers even though the Sailboat class's ConsumeEnergy method uses wind power and the Truck class's method uses diesel fuel.

Imagine if you didn't have the ICarryPassenger interface. Then it would be tough to tell your program which vehicles could carry people and which couldn't. You would have to look through each class that your program might use and figure out whether or not there was a method for carrying people from one place to another. Then you'd have to call each of the vehicles your program was going to use with whatever method was defined for carrying passengers. And since there's no standard interface, they could be named all sorts of things or buried inside other methods. You can see how that gets confusing pretty fast.

Q: Why do I need to use properties in interfaces? Can't I just include fields?

A: Good question. An interface only defines the way a class should do a specific kind of job. It's not an object by itself, so you can't instantiate it and it can't store information. If you added a field that was just a variable declaration, then C# would have to store that data somewhere—and an interface can't store data by itself. A property is a way to make something that looks like a field to other objects, but since it's really a method, it doesn't actually store any data.

Q: What's the difference between a regular object reference and an interface reference?

A: You already know how a regular, everyday object reference works. If you create an instance of Skateboard called `vertBoard`, and then a new reference to it called `halfPipeBoard`, they both point to the same thing. But if Skateboard implements the interface IStreetTricks and you create an interface reference to Skateboard called `streetBoard`, it will only know the methods in the Skateboard class that are also in the IStreetTricks interface.

All three references are actually pointing to the same object. If you call the object using the `halfPipeBoard` or `vertBoard` references, you'll be able to access any method or property in the object. If you call it using the `streetBoard` reference, you'll only have access to the methods and properties in the interface.

Q: Then why would I ever want to use an interface reference, if it limits what I can do with the object?

A: Interface references give you a way of working with a bunch of different kinds of objects that do the same thing. You can create an array using the interface reference type that will let you pass information to and from the methods in ICarryPassenger whether you're working with a Truck object, a Horse object, a Unicycle object, or a Car object. The way each of those objects does the job is probably a little different, but with interface references, you know that they all have the same methods that take the same parameters and have the same return types. So, you can call them and pass information to them in exactly the same way.

Q: Remind me again why would I make a class member protected instead of private or public?

A: Because it helps you encapsulate your classes better. There are a lot of times when a subclass needs access to some internal part of its base class. For example, if you need to override a property, it's pretty common to use the backing field in the base class in the get accessor, so that it returns some sort of variation of it. When you build classes, you should only make something public if you have a reason to do it. Using the `protected` access modifier lets you expose it only to the subclass that needs it, and keep it private from everyone else.

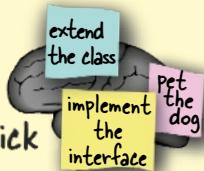
**Interface references only know
about the methods and properties
that are defined in the interface.**

BULLET POINTS

- An **interface** defines methods and properties that a class must implement.
- Interfaces define their **required members** using abstract methods and properties.
- By default, all interface members are **public** and **abstract** (so the public and abstract keywords are typically left off each member).
- When you use a **colon (:)** to make a class implement an interface, the class **must implement all of its members** or the code won’t compile.
- A class can **implement multiple interfaces** (and it doesn’t run into the Deadly Diamond of Death because the interfaces have no implementation).
- Interfaces are really useful because they let **unrelated** classes do the **same job**.
- Whenever you create an interface, you should make its name start with an **uppercase I** (that’s just a convention; it’s not enforced by the compiler).
- We use **dashed arrows** to draw interface implementation relationships in our class diagrams.
- You **can’t use the new keyword** to instantiate an interface because its members are abstract.
- You can use an **interface as a type** to reference an object that implements it.
- Any class can **implement any interface** as long as it keeps the promise of implementing the interface’s methods and properties.
- Everything in a public interface is **automatically public**, because you’ll use it to define the public methods and properties of any class that implements it.
- A **hack** is an ugly, clumsy, or inelegant solution to a problem that will be difficult to maintain.
- The **is keyword** returns true if an object matches a type. You can also use it to declare a variable and set it to reference the object you’re checking.
- **Upcasting** typically means using normal assignment or casting to move up the class hierarchy, or assign a superclass variable to reference a subclass object.
- The **is keyword** lets you **downcast**—safely move down the class hierarchy—to use a subclass variable to reference a superclass object.
- Upcasting and downcasting **work with interfaces**, too—you can upcast an object reference to an interface reference, or downcast from an interface reference.
- The **as keyword** is like a cast, except instead of throwing an exception it returns null if the cast is invalid.
- When you mark a field with the **readonly keyword** it can only be set in the field initializer or constructor.

Look up “implement” in the dictionary—one definition is “to put → a decision, plan, or agreement into effect.”

Make it Stick



To remember how interfaces work: you extend a class, but implement an interface. Extending something means taking what’s already there and stretching it out (in this case, by adding behavior). Implementing means putting an agreement into effect—you’ve agreed to add all the interface members (and the compiler holds you to that agreement).



I THINK THERE'S A **HUGE FLAW** IN INTERFACES.
WHEN I WRITE AN ABSTRACT CLASS I CAN INCLUDE
CODE. DOESN'T THAT MAKE ABSTRACT CLASSES
SUPERIOR TO INTERFACES?

Actually, you can add code to your interfaces by including static members and default implementations.

Interfaces aren't just about making sure classes that implement them include certain members. Sure, that's their main job. But interfaces can also contain code, just like the other tools you use to create your class model.

The easiest way to add code to an interface is to add **static methods, properties, and fields**. These work exactly like static members in classes: they can store data of any type—including references to objects—and you can call them just like any other static method: `Interface.MethodName();`

You can also include code in your interfaces by adding **default implementations** for methods. To add a default implementation, you just add a method body to the method in your interface. This method is not part of the object—this is not the same as inheritance—and you can only access it using an interface reference. It can call methods implemented by the object, as long as they're part of the interface.



Default interface implementations are a recent C# feature.

If you're using an old version of Visual Studio, you may not be able to use default implementations because they were added in C# 8.0, which first shipped in Visual Studio 2019 version 16.3.0, released in September 2019. Support for the current version of C# may not be available in older versions of Visual Studio.

Interfaces can have static members

Everybody loves it when way too many clowns pack themselves into a tiny clown car! So let's update the IClown interface to add static methods that generate a clown car description. Here's what we'll add:

- ★ We'll be using random numbers, so we'll add a static reference to an instance of Random. It only needs to be used in IClown for now, but we'll also use it in IScaryClown soon, so go ahead and mark it **protected**.
- ★ A clown car is only funny if it's packed with clowns, so we'll add a static int property with a private static backing field and a setter that only accepts values over 10.
- ★ A method called ClownCarDescription returns a string that describes the clown car.

Here's the code—it uses a static field, property, and method just like you'd see in a class:

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();

    protected static Random random = new Random();

    private static int carCapacity = 12;

    public static int CarCapacity {
        get { return carCapacity; }
        set {
            if (value > 10) carCapacity = value;
            else Console.Error.WriteLine($"Warning: Car capacity {value} is too small");
        }
    }

    public static string ClownCarDescription()
    {
        return $"A clown car with {random.Next(CarCapacity / 2, CarCapacity)} clowns";
    }
}
```

Now you can update the Main method to access the static IClown members:

```
static void Main(string[] args)
{
    IClown.CarCapacity = 18;
    Console.WriteLine(IClown.ClownCarDescription());

    // the rest of the Main method stays the same
}
```

These static interface members behave exactly like the static class members that you've used in previous chapters. Public members can be used from any class, private members can only be used from inside IClown, and protected members can be used from IClown or any interface that extends it.

IClown
FunnyThingIHave static CarCapacity protected static Random
Honk static ClownCarDescription

Add this!

The static random field is marked with the **protected** access modifier. That means it can only be accessed from within IClown or any interface that extends IClown (such as IScaryClown).

Try adding a **private** field to your interface. You can add one—but only if it's static! If you remove the **static** keyword, the compiler will tell you that interfaces can't contain instance fields.

now your interface method can have a body

Default implementations give bodies to interface methods

All of the methods that you've seen in interfaces so far—except for the static methods—have been abstract: they don't have bodies, so any class that implements the interface must provide an implementation for the method.

But you can also provide a **default implementation** for any of your interface methods. Here's an example:

```
interface IWorker {  
    string Job { get; }  
    void WorkTheNextShift();  
  
    void Buzz() {  
        Console.WriteLine("Buzz!");  
    }  
}
```

You can even add private methods to your interface if you want, but they can only be called from public default implementations.

You can call the default implementation—but you **must use an interface reference** to make the call:

```
IWorker worker = new NectarCollector();  
worker.Buzz();
```

But this code will not compile—it will give you the error “*NectarCollector* does not contain a definition for ‘Buzz’”:

```
NectarCollector pearl = new NectarCollector();  
pearl.Buzz();
```

The reason is that when an interface method has a default implementation, that makes it a virtual method, just like the ones you used in classes. Any class that implements the interface has the option to implement the method. The virtual method is **attached to the interface**. Like any other interface implementation, it's not inherited. That's a good thing—if a class inherited default implementations from every interface that it implemented, then if two of those interfaces had methods with the same name the class would run into the Deadly Diamond of Death.

Use @ to create verbatim string literals

The @ character has special meaning in C# programs. When you put it at the beginning of a string literal, it tells the C# compiler that the literal should be interpreted verbatim. That means slashes are not used for escape sequences—so @“\n” will contain a slash character and an n character, not a newline. It also tells the C# compiler to include any line breaks. So this: @“Line 1

Line 2” is the same as “Line1\nLine2” (including the line break).

You can use verbatim string literals to create multiline strings that include line breaks. They work great with string interpolation—just add a \$ to the beginning.

Add a ScareAdults method with a default implementation

Our IScaryClown interface is state-of-the-art when it comes to simulating scary clowns. But there's a problem: it only has a method to scare little children. What if we want our clowns to terrify the living \$#!* out of adults too?

We **could** add an abstract ScareAdults method to the IScaryClown interface. But what if we already had dozens of classes that implemented IScaryClown? And what if most of them would be perfectly fine with the same implementation of the ScareAdults method? That's where default implementations are really useful. A default implementation lets you add a method to an interface that's already in use **without having to update any of the classes that implement it**. Add a ScareAdults method with a default implementation to IScaryClown:

```
interface IScaryClown : IClown
{
    string ScaryThingIHave { get; }
    void ScareLittleChildren();
    void ScareAdults() { Console.WriteLine($"I am an ancient evil that will haunt your dreams.
Behold my terrifying necklace with {random.Next(4, 10)} of my last victim's fingers.");
    }
    Oh, also, before I forget...");
        ScareLittleChildren();
}
```

We used a verbatim literal here. We could have used a normal string literal instead and added \n's for the line breaks. This way is a lot easier to read.

Add this!

Take a close look at how the ScareAdults method works. That method only has two statements, but there's a lot packed into them. Let's break down exactly what's going on:

- ★ The Console.WriteLine statement uses a verbatim literal with string interpolation. The literal starts with \$@ to tell the C# compiler two things: the \$ tells it to use string interpolation, and the @ tells it to use a verbatim literal. That means the string will include three line breaks.
- ★ The literal uses string interpolation to call random.Next(4, 10), which uses the private static random field that IScaryClown inherited from IClown.
- ★ We've seen throughout the book that when there's a static field, that means there's only one copy of that field. So there's just one instance of Random that both IClown and IScaryClown share.
- ★ The last line of the ScareAdults method calls ScareLittleChildren. That method is abstract in the IScaryClown interface, so it will call the version of ScareLittleChildren in the class that implements IScaryClown.
- ★ That means ScareAdults will call the version of ScareLittleChildren that's defined in whatever class implements IScaryClown.

Call your new default implementation by modifying the block after the **if** statement in your Main method to call ScareAdults instead of ScareLittleChildren:

```
if (fingersTheClown is IScaryClown iScaryClownReference)
{
    iScaryClownReference.ScareAdults();
}
```



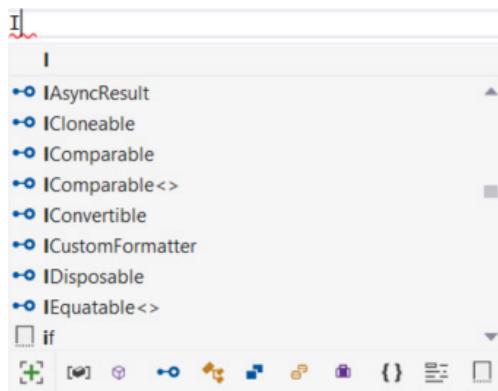
INTERFACES SEEM SO THEORETICAL. I CAN SEE HOW THEY WORK IN THE LITTLE EXAMPLES IN THIS BOOK, BUT DO DEVELOPERS ACTUALLY USE THEM IN REAL PROJECTS?

C# developers use interfaces *all the time*, especially when we use libraries, frameworks, and APIs.

Developers always stand on the shoulders of giants. You’re about halfway through this book, and in the first half you’ve written code that prints text to the console, draws windows with buttons, and renders 3D objects. You didn’t need to write code to specifically output individual bytes to the console, or draw the lines and text to display buttons in a window, or do the math needed to display a sphere—you took advantage of code that other people wrote:

- ★ You’ve used **frameworks** like .NET Core and WPF.
- ★ You’ve used **APIs** like the Unity scripting API.
- ★ The frameworks and APIs contain **class libraries** that you access with **using** directives at the top of your code.

And when you’re using libraries, frameworks, and APIs, you use interfaces a lot. See for yourself: open up a .NET Core or WPF application, click inside of any method, and type **I** to pop up an IntelliSense window. Any potential match that has the symbol next to it is an interface. These are all interfaces that you can use to work with the framework.

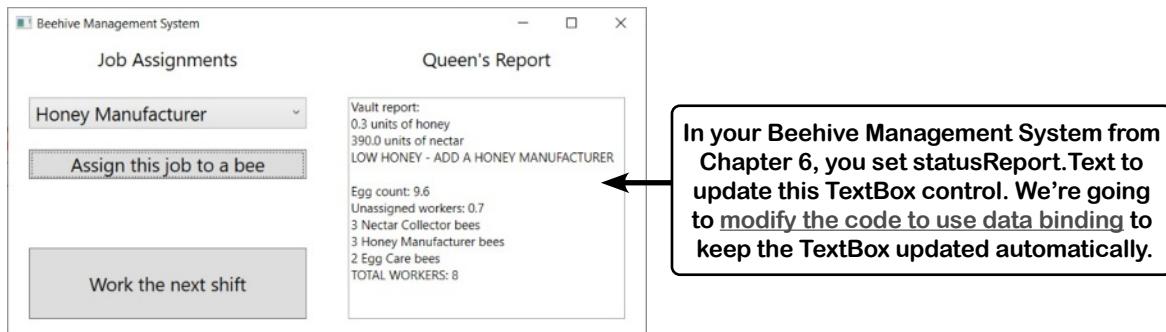


There's no Mac equivalent to the WPF feature discussed next, so the Visual Studio for Mac Learner's Guide skips this section.

interfaces casting and "is"

Data binding updates WPF controls automatically

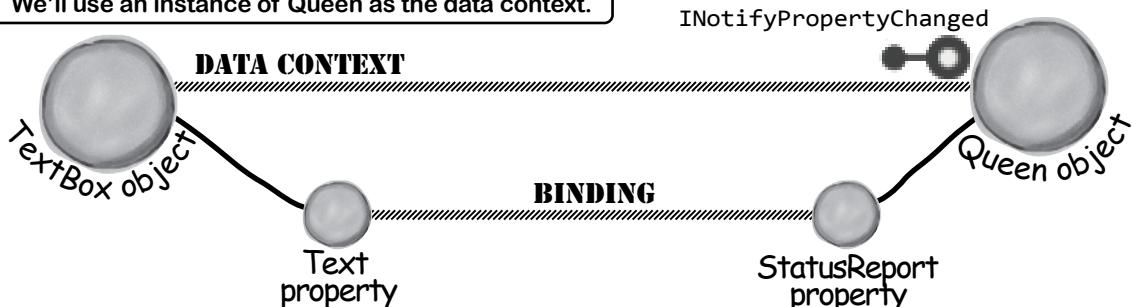
Here's a great example of a real-world use case for an interface: **data binding**. Data binding is a really useful feature in WPF that lets you set up your controls so their properties are automatically set based on a property in an object, and when that property changes your controls' properties are automatically kept up to date.



Here's an overview of the steps to modify your Beehive Management System—we'll dig into them next:

- 1 **Modify the Queen class to implement the `INotifyPropertyChanged` interface.**
This interface lets the Queen announce that the status report has been updated.
- 2 **Modify the XAML to create an instance of Queen.**
We'll bind the `TextBox.Text` property to the Queen's `StatusReport` property.
- 3 **Modify the code-behind so the “queen” field uses the instance of Queen we just created.**
Right now the `queen` field in `MainWindow.xaml.cs` has a field initializer with a `new` statement to create an instance of Queen. We'll modify it to use the instance we created with XAML instead.

Data binding starts with a data context—that's the object that contains the data to display in the `TextBox`. We'll use an instance of Queen as the data context.



The Queen needs to tell the `TextBox` when her `StatusReport` property has been updated. To do that, we'll update the Queen class to implement the `INotifyPropertyChanged` interface.

Modify the Beehive Management System to use data binding

You only need to make a few changes to add data binding to your WPF app.

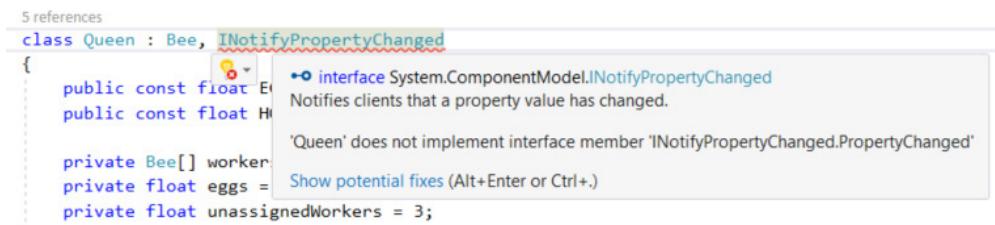
Do this!

1 Modify the Queen class to implement the `INotifyPropertyChanged` interface.

Update the Queen class declaration to make it implement `INotifyPropertyChanged`. That interface is in the `System.ComponentModel` namespace, so you'll need to add a `using` directive to the top of the class:

```
using System.ComponentModel;
```

Now you can add `INotifyPropertyChanged` to the end of the class declaration. The IDE will draw a red squiggly underline beneath it—which is what you'd expect, since you haven't implemented the interface by adding its members yet.



Press Alt+Enter or Ctrl+. to show potential fixes **and choose “Implement interface”** from the context menu. The IDE will add a line of code to your class with the `event keyword`, which you haven't seen yet:

```
public event PropertyChangedEventHandler PropertyChanged;
```

But guess what? You've used events before! The `DispatcherTimer` you used in Chapter 1 has a `Tick` event, and WPF `Button` controls have a `Click` event. **Now your Queen class has a `PropertyChanged` event.** Any class that you use for data binding fires—or **invokes**—its `PropertyChanged` event to let WPF know a property has changed.

Your Queen class needs to fire its event, just like the `DispatcherTimer` fires its `Tick` event on an interval and the `Button` fires its `Click` event when the user clicks on it. So **add this `OnPropertyChanged` method**:

```
protected void OnPropertyChanged(string name)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

Now you just need to **modify the `UpdateStatusReport` method** to call `OnPropertyChanged`:

```
private void UpdateStatusReport()
{
    StatusReport = $"Vault report:\n{n{HoneyVault.StatusReport}}\n" +
    $"\\nEgg count: {eggs:0.0}\\nUnassigned workers: {unassignedWorkers:0.0}\\n" +
    $"\\n{WorkerStatus("Nectar Collector")}\\n{WorkerStatus("Honey Manufacturer")}" +
    $"\\n{WorkerStatus("Egg Care")}\\nTOTAL WORKERS: {workers.Length}";
    OnPropertyChanged("StatusReport");
}
```

You added an `event` to your Queen class, and added a method that uses the `?.` operator to invoke the event. That's all you need to know about events for now—at the end of the book we'll point you to a downloadable chapter that teaches you more about events.

2

Modify the XAML to create an instance of Queen.

You've created objects with the `new` keyword, and you've used Unity's `Instantiate` method. XAML gives you another way to create new instances of your classes. **Add this to your XAML** just above the `<Grid>` tag:

```
<Window.Resources>
    <local:Queen x:Key="queen"/>
</Window.Resources>
```

Next, **modify the `<Grid>` tag** to add a `DataContext` attribute:

```
<Grid DataContext="{StaticResource queen}">
```

This tag creates a new instance of the Queen object and adds it to your window's `resources`, a way for WPF windows to store references to objects used by its controls.

Finally, **add a Text attribute to the `<TextBox>` tag** to bind it to the Queen's `StatusReport` property:

```
<TextBox Text="{Binding StatusReport, Mode=OneWay}">
```

Now the `TextBox` will update automatically any time the `Queen` object invokes its `PropertyChanged` event.

3

Modify the code-behind to use the instance of Queen in the window's resources.

Right now the `queen` field in `MainWindow.xaml.cs` has a field initializer with a `new` statement to create an instance of `Queen`. We'll modify it to use the instance we created with XAML instead.

First, comment out (or delete) the three occurrences of the line that sets `statusReport.Text`. There's one in the `MainWindow` constructor and two in the Click event handlers:

```
// statusReport.Text = queen.StatusReport;
```

Next, modify the `Queen` field declaration to remove the field initializer (`new Queen();`) from the end:

```
private readonly Queen queen;
```

Finally, modify the constructor to set the `queen` field like this:

```
public MainWindow()
{
    InitializeComponent();
    queen = Resources["queen"] as Queen;
    //statusReport.Text = queen.StatusReport;
    timer.Tick += Timer_Tick;
    timer.Interval = TimeSpan.FromSeconds(1.5);
    timer.Start();
}
```

Now that the WPF app uses data binding we don't need to use the `Text` property to update the status report `TextBox`, so go ahead and comment out or delete this line.

This code uses a **dictionary** called `Resources`. (*This is a sneak peek at dictionaries! You'll learn about them in the next chapter.*) Now run your game. It works exactly like before, but now the `TextBox` updates automatically any time the `Queen` updates the status report.

Congratulations! You just used an interface to add data binding to your WPF app.

there are no Dumb Questions

Q: I think I understand everything that we just did. Can you go over it again, just in case I missed something?

A: Absolutely. The Beehive Management System app you built in Chapter 6 updated its TextBox (statusReport) by setting its Text property in code like this:

```
statusReport.Text = queen.StatusReport;
```

You modified that app to use data binding to automatically update the TextBox any time the Queen object updates its StatusReport property. You did this by making three changes. First, you modified the Queen class to implement the INotifyPropertyChanged interface so it could notify the UI of any changes to the property. Then you modified the XAML to create an instance of Queen and bind the TextBox.Text property to the Queen object's StatusReport property. Finally, you modified the code-behind to use the instance created by the XAML and remove the lines that set statusReport.Text.

Q: And what exactly is the interface for?

A: The INotifyPropertyChanged interface gives you a way to tell WPF that a property changed, so it can update any controls that bind to it. When you implement it, you're building a class that can do a specific job: notifying WPF apps of property changes. The interface has a single member, an event called PropertyChanged. When you use your class for data binding, WPF checks to see if it implements INotifyPropertyChanged, and if it does, it attaches an event handler to your class's PropertyChanged event, just like you attached event handlers to your Buttons' click events.

Q: I noticed that when I open the window in the designer, the status report TextBox isn't blank anymore. Is that because of data binding?

A: Good eye! Yes, when you modified the XAML to add the <Window.Resources> section to create a new instance of the Queen object, the Visual Studio XAML designer created an instance of the object. When you modified the Grid to add a data context and added binding to the TextBox's Text property, the designer used that information to display the text. So once you use data binding, your classes aren't just being instantiated when your program runs. Visual Studio will create instances of your objects **while you're editing in the XAML window**. This is a really powerful feature of the IDE, because it lets you change properties in your code and see the results in the designer as soon as you rebuild your code.



**Data binding
works with
properties,
not fields.**

*You can only
use data binding with **public
properties**. If you try to bind
a WPF control attribute to a
public field instead, you won't
see any changes—but you
won't get an exception, either.*

Polymorphism means that one object can take many different forms

Any time you use a RoboBee in place of an IWorker, or a Wolf in place of an Animal, or even an aged Vermont cheddar in a recipe that just calls for cheese, you're using **polymorphism**. That's what you're doing any time you upcast or downcast. It's taking an object and using it in a method or a statement that expects something else.

Keep your eyes open for polymorphism!

You've been using polymorphism throughout—we just didn't use that word to describe it. While you're writing code over the next few chapters, be on the lookout for the many different ways you use it.

Here's a list of four typical ways that you'll use polymorphism. We're providing an example of each of them, though you won't see these particular lines in the exercises. As soon as you write similar code in an exercise in later chapters in the book, come back to this page and **check it off the following list**:

- Taking any reference variable that uses one class and setting it equal to an instance of a different class.

```
NectarStinger bertha = new NectarStinger();
INectarCollector gatherer = bertha;
```

- Upcasting by using a subclass in a statement or method that expects its base class.

```
spot = new Dog();
zooKeeper.FeedAnAnimal(spot);
```

If FeedAnAnimal expects an Animal object, and Dog inherits from Animal, then you can pass Dog to FeedAnAnimal.

- Creating a reference variable whose type is an interface and pointing it to an object that implements that interface.

```
IStingPatrol defender = new StingPatrol();
```

This is upcasting, too!

- Downcasting using the `is` keyword.

```
void MaintainTheHive(IWorker worker) {
    if (worker is HiveMaintainer) {
        HiveMaintainer maintainer = worker as HiveMaintainer;
        ...
    }
}
```

The MaintainTheHive method takes any IWorker as a parameter. It uses "as" to point a HiveMaintainer reference to the worker.



The idea that you could combine your data and your code into classes and objects was a revolutionary one when it was first introduced—but that's how you've been building all your C# programs so far, so you can think of it as just plain programming.

You're an object-oriented programmer.

There's a name for what you've been doing. It's called **object-oriented programming**, or OOP. Before languages like C# came along, people didn't use objects and methods when writing their code. They just used functions (which is what they call methods in a non-OO program) that were all in one place—as if each program were just one big static class that only had static methods. It made it a lot harder to create programs that modeled the problems they were solving. Luckily, you'll never have to write programs without OOP, because it's a core part of C#.

The four core principles of object-oriented programming

When programmers talk about OOP, they're referring to four important principles. They should seem very familiar to you by now because you've been working with every one of them. We just told you about polymorphism, and you'll recognize the first three principles from Chapters 5 and 6: **inheritance**, **abstraction**, and **encapsulation**.

This just means having one class or interface that inherits from another.

* **Inheritance** ←

Encapsulation means creating an object that keeps track of its state internally using private fields, and uses public properties and methods to let other classes work with only the part of the internal data that they need to see.

* **Encapsulation** ←

The word "polymorphism" literally means "many forms." Can you think of a time when an object has taken on many forms in your code?

* **Abstraction**

You're using abstraction when you create a class model that starts with more general—or abstract—classes, and then has more specific classes that inherit from it.

* **Polymorphism**

8 enums and collections

Organizing your data



Data isn't always as neat and tidy as you'd like it to be.

In the real world, you don't receive your data in tidy 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.

Strings don't always work for storing categories of data

We're going to be working with playing cards over the next few chapters, so let's build a Card class that we'll use. First, create a new Card class that has a constructor that lets you pass it a suit and value, which it stores as strings:

```
class Card
{
    public string Value { get; set; }
    public string Suit { get; set; }
    public string Name { get { return $"{Value} of {Suit}"; } }

    public Card(string value, string suit)
    {
        Value = value;
        Suit = suit;
    }
}
```

That looks pretty good. We can create a Card object and use it like this:

```
Card aceOfSpades = new Card("Ace", "Spades");
Console.WriteLine(aceOfSpades); // prints Ace of Spades
```

But there's a problem. Using strings to hold suits and values can have some unexpected results:

```
Card duchessOfRoses = new Card("Duchess", "Roses");
Card fourteenOfBats = new Card("Fourteen", "Bats");
Card dukeOfOxen = new Card("Duke", "Oxen");
```

} This code compiles, but these suits and values
don't make any sense at all. The Card class
really shouldn't allow these types as valid data.

We **could** add code to the constructor to check each string and make sure it's a valid suit or value, and handle bad input by throwing an exception. That's a valid approach—assuming you deal with the exceptions correctly, of course.

But **wouldn't it be great** if the C# compiler could automatically detect those invalid values for us? What if the compiler could ensure that all of the cards are valid before you even run the code? Well, guess what: it **will** do that! All you need to do is **enumerate** the values that are OK to use.

e-nu-me-rate, verb.

to specify one after another.

*Ralph kept losing track of his pigeons, so he decided to **enumerate** them by writing each of their names on a piece of paper.*

Card
Suit Value Name

↑
This Card class uses
string properties for
suits and values.



The rarely played Duke of Oxen card.

Enums let you work with a set of valid values

An **enum** or **enumeration type** is a data type that only allows certain values for that piece of data. So we could define an enum called `Suits`, and define the allowed suits:

```
enum Suits {
    Diamonds,
    Clubs,
    Hearts,
    Spades,
}
```

Every enum starts with the `enum` keyword followed by its name. This enum is called `Suits`.

The rest of the enum is a list of members inside a set of curly braces, separated by commas. There's one member for each unique value—in this case, a member for each suit.

The last enum member doesn't have a comma after it, but using one makes it easier to rearrange them using cut and paste.

An enum defines a new type

When you use the `enum` keyword you're **defining a new type**. Here are a few useful things to know about enums:

- ✓ You can use an enum as the type in a variable definition, just like you'd use `string`, `int`, or any other type:

```
Suits mySuit = Suits.Diamonds;
```

- ✓ Since an enum is a type, you can use it to create an array:

```
Suits[] myVals= new Suits[3] { Suits.Spades, Suits.Clubs, mySuit };
```

- ✓ Use `==` to compare enum values. Here's a method that takes a `Suit` enum as a parameter, and uses `==` to check if it's equal to `Suits.Hearts`:

```
void IsItAHeart(Suits suit) {
    if (suit == Suits.Hearts) {
        Console.WriteLine("You pulled a heart!");
    } else {
        Console.WriteLine($"You didn't pull a heart: {suit}");
    }
}
```

An enum's `ToString` method returns the equivalent string, so `Suits.Spades.ToString` returns "Spades".

- ✓ But you can't just make up a new value for the enum. If you do, the program won't compile—which means you can avoid some annoying bugs:

```
IsItAHeart(Suits.Oxen);
```

The compiler gives you an error if you use a value that's not part of the enum:

✖ CS0117 'Suits' does not contain a definition for 'Oxen'

An enum lets you define a new type that only allows a specific set of values. Any value that's not part of the enum will break the code, which can prevent bugs later.

Enums let you represent numbers with names

Sometimes it's easier to work with numbers if you have names for them. You can assign numbers to the values in an enum and use the names to refer to them. That way, you don't have a bunch of unexplained numbers floating around in your code. Here's an enum to keep track of the scores for tricks at a dog competition:

```
enum TrickScore {  
    Sit = 7,  
    Beg = 25,  
    RollOver = 50,  
    Fetch = 10,  
    ComeHere = 5,  
    Speak = 30,  
}
```

Members don't have to be in any particular order, and you can give multiple names to the same number.

Supply a name, then “=”, then the number that name stands in for.

You can cast an int to an enum, and you can cast an (int-based) enum back to an int.

Some enums use a different type, like byte or long (like the one below). You can cast those to their type instead of int.

Here's an excerpt from a method that uses the `TrickScore` enum by casting it to and from an int value:

```
int score = (int)TrickScore.Fetch * 3;  
// The next line prints: The score is 30  
Console.WriteLine($"The score is {score}");
```

The (int) cast tells the compiler to turn this into the number it represents. So since `TrickScore.Fetch` has a value of 10, (int) `TrickScore.Fetch` turns it into the int value 10.

You can cast the enum as a number and do calculations with it. You can even convert it to a string—an enum's `ToString` method returns a string with the member name:

```
TrickScore whichTrick = (TrickScore)7;  
// The next line prints: Sit  
Console.WriteLine(whichTrick.ToString());
```

You can cast an int back to a `TrickScore`, and `TrickScore.Sit` has the value 7.

Console.WriteLine calls the enum's `ToString` method, which returns a string with the member name.

If you don't assign any number to a name, the items in the list will be given values by default. The first item will be assigned a 0 value, the second a 1, etc. But what happens if you want to use really big numbers for one of the enumerators? The default type for the numbers in an enum is int, so you'll need to specify the type you need using the colon (:) operator, like this:

```
enum LongTrickScore : long {  
    Sit = 7,  
    Beg = 250000000025  
}
```

This number is too big to fit into an int.

This tells the compiler to treat values in the `TrickScore` enum as longs, not ints.

If you tried to use this enum without specifying long as the type, you'd get an error:

 CS0266 Cannot implicitly convert type 'long' to 'int'.



Exercise

Use what you've learned about enums to build a class that holds a playing card. Start by creating a new .NET Core Console App project and adding a class called Card.

Add two public properties to Card: Suit (which will be Spades, Clubs, Diamonds, or Hearts) and Value (Ace, Two, Three...Ten, Jack, Queen, King). You'll also need one more property: a public read-only property called Name that returns a string like "Ace of Spades" or "Five of Diamonds".

Card
Value
Suit
Name



To add an enum in Visual Studio for Mac, add a file and choose "Empty Enumeration" as the file type.

Add two enums to define the suits and values in their own *.cs files

Add each enum. In Windows, use the familiar *Add >> Class* feature, then **replace class with enum** in each file. In macOS, use *Add >> New File...* and choose Empty Enumeration. **Use the Suits enum that we just showed you**, then create an enum for values. Make the values equal to their face values: (int)Values.Ace should equal 1, Two should be 2, Three should be 3, etc. Jack should equal 11, Queen should be 12, and King should be 13.

Add a constructor and Name property that returns a string with the name of the card

Add a constructor that takes two parameters, a Suit and a Value:

```
Card myCard = new Card(Values.Ace, Suits.Spades);
```

Name should be a read-only property. The get accessor should return a string that describes the card. So this code:

```
Console.WriteLine(myCard.Name);
```

should print the following:

Ace of Spades

Make the Main method print the name of a random card

You can get your program to create a card with a random suit and value by casting a random number between 0 and 3 as a Suits enum and another random number between 1 and 13 as a Values enum. To do this, you can take advantage of a feature of the built-in Random class that gives it three different ways to call its Next method:

When you've got more than one way to call a method, it's called overloading.

```
Random random = new Random();
int numberBetween0and3 = random.Next(4);
int numberBetween1and13 = random.Next(1, 14);
int anyRandomInteger = random.Next();
```

You did this back in Chapter 3. It tells Random to return a value of at least 1 but under 14.

there are no Dumb Questions

Q: I remember calling Random.Next with two arguments earlier in the book. I noticed when I called the method, an IntelliSense window popped up that said "3 of 3" in the corner. Does that have to do with overloading?

A: Yes! When a class has an **overloaded** method—or a method that you can call more than one way—the IDE lets you know all of the options that you have. In this case, the Random class has three possible Next methods. As soon as you type `random.Next()` into the code window, the IDE pops up its IntelliSense box that shows

the parameters for the different overloaded methods. The up and down arrows next to the "3 of 3" let you scroll between them. That's really useful when you're dealing with a method that has dozens of overloaded definitions. So when you're calling Random.Next, make sure you choose the right overloaded method. But don't worry too much now—we'll talk a lot about overloading later on in the chapter.

▲ 3 of 3 ▼ `int Random.Next()`
 ★ IntelliCode suggestion based on this context
 Returns a non-negative random integer.



Exercise Solution

A deck of cards is a great example of a program where limiting values is important. Nobody wants to turn over their cards and be faced with a 28 of Hearts or Ace of Hammers. Here's our Card class—you'll reuse it a few times over the next few chapters.

The Suits enum is in a file called *Suits.cs*. You already have the code for it—it's identical to the Suits enum that we showed you earlier in the chapter. The Values enum is in a file called *Values.cs*. Here's its code:

```
enum Values {
    Ace = 1,
    Two = 2,
    Three = 3,
    Four = 4,
    Five = 5,
    Six = 6,
    Seven = 7,
    Eight = 8,
    Nine = 9,
    Ten = 10,
    Jack = 11,
    Queen = 12,
    King = 13,
}
```

Here's where we set the value of Values.Ace to 1.

And the value of Values.King is 13.

We chose the names **Suits** and **Values** for the enums, while the properties in the Card class that use those enums for types are called **Suit** and **Value**. What do you think about these names? Look at the names of other enums that you'll see throughout the book. Would **Suit** and **Value** make better names for these enums?

There's no right or wrong answer—in fact, Microsoft's C# language reference page for enums has both singular (e.g., Season) and plural (e.g., Days) names: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum>.

The Card class has a constructor that sets its Suit and Value properties, and a Name property that generates a string description of the card:

```
class Card {
    public Values Value { get; private set; }
    public Suits Suit { get; private set; }

    public Card(Values value, Suits suit) {
        this.Suit = suit;
        this.Value = value;
    }

    public string Name {
        get { return $"{Value} of {Suit}"; }
    }
}
```

Here's an example of encapsulation.
We made the setters for the Value and Suit properties private because they only ever need to be called from the constructor. That way we'll never accidentally change them.

The Name property's get accessor takes advantage of the way an enum's ToString method returns its name converted to a string.

The Program class uses a static Random reference to cast **Suits** and **Values** to instantiate a random Card:

```
class Program
{
    private static readonly Random random = new Random();

    static void Main(string[] args)
    {
        Card card = new Card((Values)random.Next(1, 14), (Suits)random.Next(4));
        Console.WriteLine(card.Name);
    }
}
```

The overloaded Random.Next method is used here to generate a random number from 1 to 13. It gets cast to a Values value.

We could use an array to create a deck of cards...

What if you wanted to create a class to represent a deck of cards? It would need a way to keep track of every card in the deck, and it'd need to know what order they were in. A `Cards` array would do the trick—the top card in the deck would be at value 0, the next card at value 1, etc. Here's a starting point—a `Deck` that starts out with a full deck of 52 cards:

```
class Deck
{
    private readonly Card[] cards = new Card[52];

    public Deck()
    {
        int index = 0;
        for (int suit = 0; suit <= 3; suit++)
        {
            for (int value = 1; value <= 13; value++)
            {
                cards[index++] = new Card((Values)value, (Suits)suit);
            }
        }
    }

    public void PrintCards()
    {
        for (int i = 0; i < cards.Length; i++)
            Console.WriteLine(cards[i].Name);
    }
}
```

We used two “for” loops to iterate through all of the possible suit and value combinations.

...but what if you wanted to do more?

Think of everything you might need to do with a deck of cards, though. If you're playing a card game, you routinely need to change the order of the cards, and add and remove cards from the deck. You just can't do that with an array very easily. For example, take another look at the `AddWorker` method from the Beehive Management System exercise in Chapter 6:

```
private void AddWorker(Bee worker)
{
    if (unassignedWorkers >= 1) {
        unassignedWorkers--;
        Array.Resize(ref workers, workers.Length + 1);
        workers[workers.Length - 1] = worker;
    }
}
```

You used this code to add an element to an array in Chapter 6. What would you have to do if you wanted to add the Bee reference to the middle of the array instead of the end?

You had to use `Array.Resize` to make the array longer, then add the worker to the end. That's a lot of work.

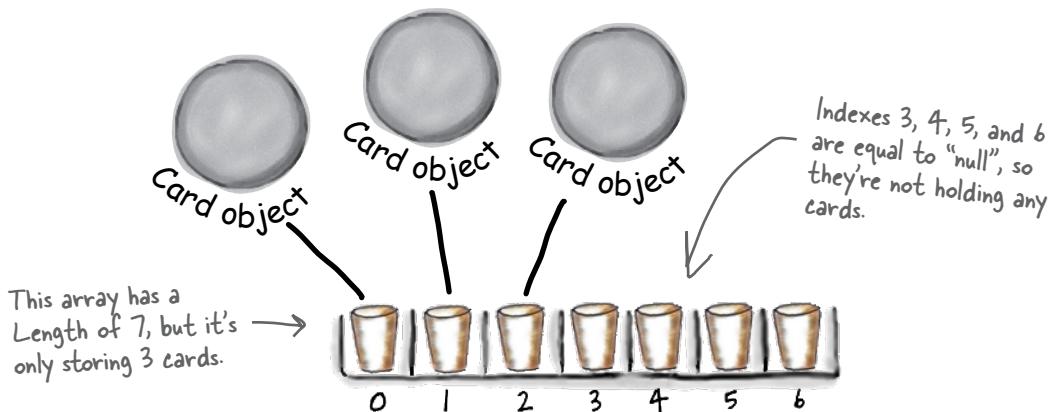


How would you add a `Shuffle` method to the `Deck` class that rearranges the cards in random order? What about a method to deal the first card off the top of the deck that returns it and then removes it from the deck? How would you add a card to the deck?

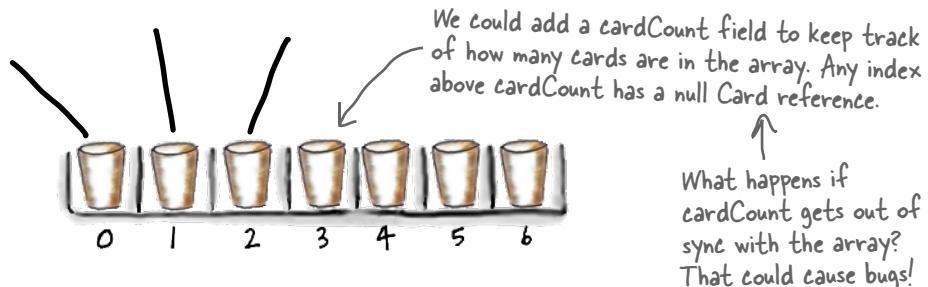
Arrays can be annoying to work with

An array is fine for storing a fixed list of values or references. Once you need to move array elements around, or add more elements than the array can hold, things start to get a little sticky. Here are a few ways that working with arrays can be troublesome.

Every array has a length. That length doesn't change unless you resize the array, so you need to know the length to work with it. Let's say you want to use an array to store Card references. If the number of references you want to store is less than the length of the array, you could use null references to keep some array elements empty.



You'd need to keep track of how many cards are being held in the array. You could add an int field—maybe you'd call it `cardCount`—that would hold the index of the last card in the array. So your three-card array would have a Length of 7, but you'd set `cardCount` equal to 3.



Now things get complicated. It's easy enough to add a `Peek` method that just returns a reference to the top card, so you can peek at the top of the deck. What if you want to add a card? If `cardCount` is less than the array's Length, you can just put your card in the array at that index and add 1 to `cardCount`. But if the array is full, you'll need to create a new, bigger array and copy the existing cards to it. Removing a card is easy enough—but after you subtract 1 from `cardCount`, you'll need to make sure to set the removed card's array index back to `null`. What if you need to remove a card **from the middle of the list**? If you remove card 4, you'll need to move card 5 back to replace it, and then move 6 back, then 7...wow, what a mess!

The AddWorker method from Chapter 6 used the Array.Resize method to do this.

Lists make it easy to store collections of...anything

C# and .NET have **collection** classes that handle all of those nasty issues that come up when you add and remove array elements. The most common sort of collection is a `List<T>`. Once you create a `List<T>` object, it's easy to add an item, remove an item from any location in the list, peek at an item, and even move an item from one place in the list to another. Here's how a list works.

We'll sometimes leave the `<T>` off when referring to `List` in the book. When you see `List`, think `List<T>`.

- 1 First you create a new instance of `List<T>`. Recall that every array has a type—you don't just have an array, you have an `int` array, a `Card` array, etc. Lists are the same. You need to specify the type of object or value that the list will hold by putting it in angle brackets (`<>`) when you use the `new` keyword to create it:

```
List<Card> cards = new List<Card>();
```



You specified `<Card>` when you created the list, so now this list only holds references to `Card` objects.



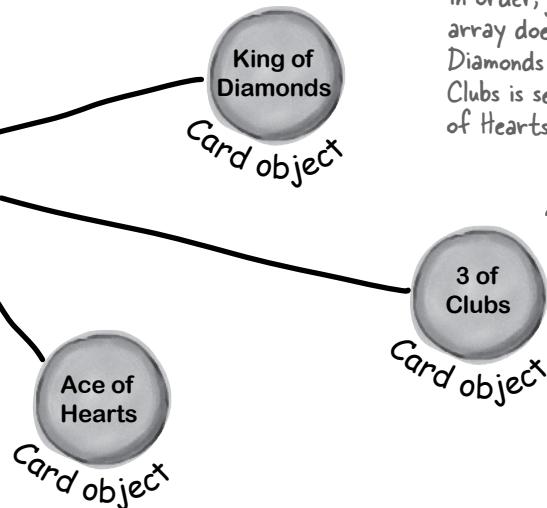
The `<T>` at the end of `List<T>` means it's generic.

The `T` gets replaced with a type—so `List<int>` just means a List of `ints`. You'll get plenty of practice with generics over the next few pages.

- 2 Now you can add to your `List<T>`. Once you've got a `List<T>` object, you can add as many items to it as you want as long as they're **polymorphic** with whatever type you specified when you created your new `List<T>`—which means they're assignable to the type (and that includes interfaces, abstract classes, and base classes).

```
cards.Add(new Card(Values.King, Suits.Diamonds));
cards.Add(new Card(Values.Three, Suits.Clubs));
cards.Add(new Card(Values.Ace, Suits.Hearts));
```

You can add as many Cards as you want to the List—just call its `Add` method. It'll make sure it's got enough "slots" for the items. If it starts to run out, it'll automatically resize itself.



The values or object references contained in a list are typically referred to as its elements.

A list keeps its elements in order, just like an array does. King of Diamonds is first, 3 of Clubs is second, and Ace of Hearts is third.

Lists are more flexible than arrays

The List class is built into the .NET Framework, and it lets you do a lot of things with objects that you can't do with a plain old array. Check out some of the things you can do with a List<T>.

- 1 Use the new keyword to instantiate a List (like you'd expect!).

```
List<Egg> myCarton = new List<Egg>();
```

Here's a reference to an Egg object.

- 2 Add something to a List.

```
Egg x = new Egg();  
myCarton.Add(x);
```

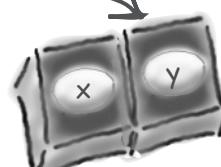


new List<egg>(); creates a List of Egg objects. It starts out empty. You can add or remove objects, but since it's a List of Eggs, you can only add references to Egg objects, or to any object that can be cast to an Egg.

- 3 Add something else to the List.

```
Egg y = new Egg();  
myCarton.Add(y);
```

Here's another Egg.



Now the List expands to hold the Egg object...

...and expands again to hold the second Egg object.

- 4 Find out how many things are in a List.

```
int theSize = myCarton.Count;
```

- 5 Find out if your List has a particular thing in it.

```
bool isIn = myCarton.Contains(x);
```

Now you can search for a specific Egg inside the List. This would definitely come back true because you just added that egg to your List.

- 6 Figure out where in the List that thing is.

```
int index = myCarton.IndexOf(x);
```

The index for x would be 0 and the index for y would be 1.

- 7 Take that thing out of the List.

```
myCarton.Remove(x);
```



When we removed x, we left only y in the List, so it shrank! If we remove y, eventually it will get garbage-collected.

Sharpen your pencil



Here are a few lines from the middle of a program. Assume these statements are all executed in order, one after another, and that the variables were previously declared.

Fill in the rest of the table below by looking at the List code on the left and entering what you think the code might be if it were using a regular array instead. We don't expect you to get all of them exactly right, so just make your best guess.

List

We filled in a couple for you....

Regular array

List<String> myList = new List <String>();	String [] myList = new String[2];
String a = "Yay!"; myList.Add(a);	String a = "Yay!";
String b = "Bummer"; myList.Add(b);	String b = "Bummer";
int theSize = myList.Count;	
Guy o = guys[1];	
bool foundIt = myList.Contains(b);	
Hint: you'll need more than one line of code here. →	



Sharpen your pencil Solution

Your job was to fill in the rest of the table by looking at the List code on the left and entering what you think the code might be if it were using a regular array instead.

List	Regular array
<code>List<String> myList = new List <String>();</code>	<code>String[] myList = new String[2];</code>
<code>String a = "Yay!"</code>	<code>String a = "Yay!";</code>
<code>myList.Add(a);</code>	<code>myList[0] = a;</code>
<code>String b = "Bummer";</code>	<code>String b = "Bummer";</code>
<code>myList.Add(b);</code>	<code>myList[1] = b;</code>
<code>int theSize = myList.Count;</code>	<code>int theSize = myList.Length;</code>
<code>Guy o = guys[1];</code>	<code>Guy o = guys[1];</code>
<code>bool foundIt = myList.Contains(b);</code>	<pre>bool foundIt = false; for (int i = 0; i < myList.Length; i++) { if (b == myList[i]) { isIn = true; } }</pre>



Lists are objects that have methods, just like every other class you've used so far. You can see the list of methods available from within the IDE just by typing a . next to the List name, and you pass parameters to them just the same as you would for a class you created yourself.

The elements in a list are ordered, and the element's position in a list is called its **index**. Just like with an array, list indexes start at 0. You can access the element at a specific index of a list using its **indexer**:

Guy o = guys[1];

"Elements" is another name
for the items in a list.

With arrays, you're a lot more limited. You need to set the size of the array when you create it, and you'll have to write any logic that needs to be performed on it on your own.

The `Array` class does have some static methods which make some of these things a little easier to do—for example, you already saw the `Array.Resize` method, which you used in your `AddWorker` method. But we're concentrating on `List` objects because they're a lot easier to use.

Let's build an app to store shoes

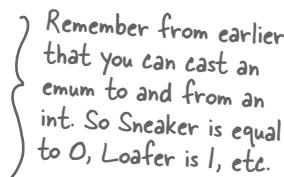
It's time to see a List in action. Let's build a .NET Core console app that prompts the user to add or remove shoes. Here's an example of what it looks like to run the app, adding two shoes and then removing them:

We'll start with a Shoe class that stores the style and color for a shoe. Then we'll create a class called ShoeCloset that stores the shoes in a List<Shoe>, with AddShoe and RemoveShoe methods that prompt the user to add or remove shoes.

 **Do this!**

- 1** **Add an enum for shoe style.** Some shoes are sneakers, others are sandals, so an enum makes sense:

```
enum Style
{
    Sneaker,
    Loafer,
    Sandal,
    Flipflop,
    Wingtip,
    Clog,
}
```



- 2** **Add the Shoe class.** It uses the Style enum for the shoe style and a string for shoe color, and works just like the Card class we created earlier in the chapter:

```
class Shoe
{
    public Style Style {
        get; private set;
    }
    public string Color {
        get; private set;
    }
    public Shoe(Style style, string color)
    {
        Style = style;
        Color = color;
    }
    public string Description
    {
        get { return $"A {Color} {Style}"; }
    }
}
```



The shoe closet is empty.

Press 'a' to add or 'r' to remove a shoe: a
Add a shoe

Press 0 to add a Sneaker

Press 1 to add a Loafer

Press 2 to add a Sandal

Press 3 to add a Flipflop

Press 4 to add a Wingtip

Press 5 to add a Clog

Enter a style: 1

Enter the color: black



The shoe closet contains:

Shoe #1: A black Loafer

Press 'a' to add or 'r' to remove a shoe: a
Add a shoe

Press 0 to add a Sneaker

Press 1 to add a Loafer

Press 2 to add a Sandal

Press 3 to add a Flipflop

Press 4 to add a Wingtip

Press 5 to add a Clog

Enter a style: 0

Enter the color: blue and white

Press 'a' to add a shoe, then choose the type of shoe and type in the color.

The shoe closet contains:

Shoe #1: A black Loafer

Shoe #2: A blue and white Sneaker

Press 'a' to add or 'r' to remove a shoe: r

Enter the number of the shoe to remove: 2

Removing A blue and white Sneaker

Press 'r' to remove a shoe, then enter the number of the shoe to remove.

Press 'a' to add or 'r' to remove a shoe: r

Enter the number of the shoe to remove: 1

Removing A black Loafer

The shoe closet is empty.

Press 'a' to add or 'r' to remove a shoe:



using a list of shoes in an app

3

The ShoeCloset class uses a List<Shoe> to manage its shoes. The ShoeCloset class has three methods—the PrintShoes method prints a list of shoes to the console, the AddShoe method prompts the user to add a shoe to the closet, and the RemoveShoe method prompts the user to remove a shoe:

```
using System.Collections.Generic; ← Make sure you have this using line at the top of your code, otherwise you won't be able to use the List class.
```

```
class ShoeCloset
{
    private readonly List<Shoe> shoes = new List<Shoe>();
```

This foreach loop iterates through the "shoes" list and writes a line to the console for each shoe.

```
    public void PrintShoes()
    {
        if (shoes.Count == 0)
        {
            Console.WriteLine("\nThe shoe closet is empty.");
        }
        else
        {
            Console.WriteLine("\nThe shoe closet contains:");
            int i = 1;
            foreach (Shoe shoe in shoes)
            {
                Console.WriteLine($"Shoe #{i++}: {shoe.Description}");
            }
        }
    }
```

Here's where we create a new Shoe instance and add it to the list.

```
    public void AddShoe()
    {
        Console.WriteLine("\nAdd a shoe");
        for (int i = 0; i < 6; i++)
        {
            Console.WriteLine($"Press {i} to add a {(Style)i}");
        }
        Console.Write("Enter a style: ");
        if (int.TryParse(Console.ReadKey().KeyChar.ToString(), out int style))
        {
            Console.Write("\nEnter the color: ");
            string color = Console.ReadLine();
            Shoe shoe = new Shoe((Style)style, color);
            shoes.Add(shoe);
        }
    }
```

Here's where we remove a Shoe instance from the list.

```
    public void RemoveShoe()
    {
        Console.Write("\nEnter the number of the shoe to remove: ");
        if (int.TryParse(Console.ReadKey().KeyChar.ToString(), out int shoeNumber) && (shoeNumber >= 1) && (shoeNumber <= shoes.Count))
        {
            Console.WriteLine($"\\nRemoving {shoes[shoeNumber - 1].Description}");
            shoes.RemoveAt(shoeNumber - 1);
        }
    }
```

Here's the List that contains the references to the Shoe objects.

ShoeCloset
private List<Shoe> shoes
PrintShoes
AddShoe
RemoveShoe

The for loop sets "i" to an integer from 0 to 5. The interpolated string uses `{(Style)i}` to cast it to a Style enum and then calls its `ToString` method to print the member name.

This is just like code you've seen before: it calls `Console.ReadKey`, then uses `KeyChar` to get the character that was pressed. `int.TryParse` needs a string, not a char, so we call `ToString` to convert the char to a string.

4 Add the Program class with the entry point.

Notice how it doesn't do very much? That's because all of the interesting behavior is encapsulated in the ShoeCloset class:

```
class Program
{
    static ShoeCloset shoeCloset = new ShoeCloset();

    static void Main(string[] args)
    {
        while (true)
        {
            shoeCloset.PrintShoes();
            Console.Write("\nPress 'a' to add or 'r' to remove a shoe: ");
            char key = Console.ReadKey().KeyChar;

            switch (key)
            {
                case 'a':
                case 'A':
                    shoeCloset.AddShoe();
                    break;
                case 'r':
                case 'R':
                    shoeCloset.RemoveShoe();
                    break;
                default:
                    return;
            }
        }
    }
}
```

There's no break statement after the 'a' case, so it falls through to the 'A' case—so they're both handled by shoeCloset.AddShoe.



We used a **switch** statement to handle the user input. We wanted uppercase 'A' to work the same as lowercase 'a', so we included two **case** statements next to each other without a **break** between them:

```
case 'a':
case 'A':
```

When a **switch** encounters a new **case** statement without a **break** before it, it falls through to the next case. You can even have statements between the two **case** statements. Be really careful with this—it's easy to accidentally leave out a **break** statement.

5

Run your app and reproduce the sample output. Try debugging the app, and start to get familiar with how you work with lists. No need to memorize anything right now—you'll get plenty of practice with them!

List Class Members Up Close



The List collection class has an **Add** method that adds an item to the end of the list. The **AddShoe** method creates a **Shoe** instance, then calls the **shoes.Add** method with the reference to that instance:

```
shoes.Add(shoe);
```

The List class also has a **RemoveAt** method that removes an item from a specific index in the list. Lists, like arrays, are **zero-indexed**, which means the first item has index 0, the second item has index 1, etc.:

```
shoes.RemoveAt(shoeNumber - 1);
```

And finally, the **PrintShoes** method uses the **List.Count** property to check if the list is empty:

```
if (shoes.Count == 0)
```

Generic collections can store any type

You've already seen that a list can store strings or Shoes. You could also make lists of integers or any other object you can create. That makes a list a **generic collection**. When you create a new list object, you tie it to a specific type: you can have a list of ints, or strings, or Shoe objects. That makes working with lists easy—once you've created your list, you always know the type of data that's inside it.

But what does “generic” really mean? Let's use Visual Studio to explore generic collections. Open *ShoeCloset.cs* and hover your mouse cursor over *List*:

```
private readonly List<Shoe> shoes = new List<Shoe>();
```

class System.Collections.Generic.List<T>

Represents a strongly typed list of objects that can be accessed by index.
Provides methods to search, sort, and manipulate lists.

T is Shoe

There are a few things to notice:

- ★ The List class is in the namespace System.Collections.Generic—this namespace has several classes for generic collections (which is why you needed the `using` line).
- ★ The description says that List provides “methods to search, sort, and manipulate lists.” You used some of these methods in your ShoeCloset class.
- ★ The top line says `List<T>` and the bottom says `T is Shoe`. This is how generics are defined—it's saying that List can handle any type, but for this specific list that type is the Shoe class.

A **generic collection** can hold any type of object, and gives you a consistent set of methods to work with the objects in the collection no matter what type of object it's holding.

Generic lists are declared using <angle brackets>

When you declare a list—no matter what type it holds—you always declare it the same way, using <angle brackets> to specify the type of object being stored in the list.

You'll often see generic classes (not just List) written like this: `List<T>`. That's how you know the class can take any type.

This doesn't actually mean that you add the letter T. It's a notation that you'll see whenever a class or interface works with all types. The `<T>` part means you can put a type in there, like `List<Shoe>`, which limits its members to that type.

```
List<T> name = new List<T>();
```

Lists can be either very flexible (allowing any type) or very restrictive. So they do what arrays do, and then quite a few things more.

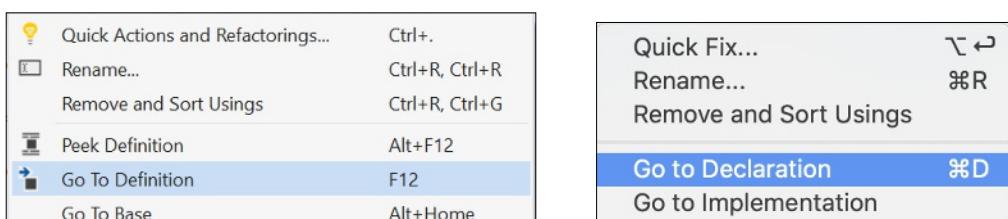
ge-ne-ric, adjective.
characteristic of or relating to a class or group of things; not specific. *Developer* is a **generic** term for anyone who writes code, no matter what kind of code they write.

IDE Tip: Go To Definition (Windows) / Go to Declaration (macOS)

The `List` class is part of .NET Core, which has a whole bunch of really useful classes, interfaces, types, and more. Visual Studio has a really powerful tool that you can use to explore these classes, and any other code you've written. Open `Program.cs` and find this line: `static ShoeCloset shoeCloset = new ShoeCloset();`

Right-click on `ShoeCloset` and choose **Go To Definition** on Windows, or **Go to Declaration** on macOS.

In Windows you can also go to a class, member, or variable definition by Control-clicking on it.



The IDE will jump straight to the definition of the `ShoeCloset` class. Now go back to `Program.cs` and go to the definition of `PrintShoes` in this line: `shoeCloset.PrintShoes();`. The IDE will jump straight to that method definition in the `ShoeCloset` class. You can use **Go To Definition/Declaration** to quickly jump around your code.

Use Go To Definition/Declaration to explore generic collections

Now comes the really interesting part. Open up `ShoeCloset.cs` and go to the definition of `List`. The IDE will open a separate tab with the definition of the `List` class. Don't worry if this new tab has a lot of complex stuff on it! You don't need to understand it all—just find this line of code, which shows you how `List<T>` implements a *bunch* of interfaces:

```
public class List<[NullableAttribute(2)] T> : ICollection<T>, IEnumerable<T>, IEnumerable,
    IList<T>, IReadOnlyCollection<T>, IReadOnlyList<T>, ICollection, IList
```

Notice how the first interface is `ICollection<T>`? That's the interface used by every generic collection. You probably guessed what you'll do next—go to the definition/declaration for `ICollection<T>`. Here's what you'll see in Visual Studio for Windows (the XML comments are collapsed and replaced with `...` buttons; they may be expanded on Mac):

```
namespace System.Collections.Generic
{
    public interface ICollection<[NullableAttribute(2)] T> : IEnumerable<T>, IEnumerable
    {
        int Count { get; }
        bool IsReadOnly { get; }

        void Add(T item);
        void Clear();
        bool Contains(T item);
        void CopyTo(T[] array, int arrayIndex);
        bool Remove(T item);
    }
}
```

A generic collection lets you find out how many items it has, add new items, clear it, check if it contains an item, and remove an item. It may do other things too—like a `List`, which lets you remove an item at a specific index—but any generic collection meets this minimum standard.

In the last chapter we talked about how interfaces are all about making classes do jobs. A generic collection is a specific job. Any class can do it, as long as it implements the `ICollection<T>` interface. The `List<T>` class does, and you'll see a few more collection classes later in the chapter that do. They all work a little differently, but because they all do the job of being a generic collection, you can depend on them all doing the basic job of storing values or references.

BULLET POINTS

- **List** is a .NET class that lets you store, manage, and easily work with a set of values or object references. The values or references stored in a list are often referred to as its **elements**.
- A List **resizes dynamically** to whatever size is needed. As you add data to the List, it grows to accommodate it.
- To put something into a List, use **Add**. To remove something from a List, use **Remove**.
- You can remove objects from a List using their index number with **RemoveAt**.
- You declare the type of the List using a **type argument**, which is a type name in angle brackets. For example: `List<Frog>` means the List will be able to hold only objects of type Frog.
- You can use the **Contains** method to find out if a particular object is in a List. The **IndexOf** method returns the index of a specific element in the List.
- The **Count** property returns the number of elements in the list.
- Use an **indexer** (like `guys[3]`) to access the item in a collection at a specific index.
- You can use **foreach loops** to iterate through lists, just like you do with arrays.
- A List is a **generic collection**, which means it can store any type.
- All generic collections implement the generic **ICollection<T> interface**.
- The `<T>` in a generic class or interface definition is **replaced with a type** when you instantiate it.
- Use the **Go To Definition** (Windows) or **Go to Declaration** (macOS) feature in Visual Studio to explore your code and other classes that you use.



Watch it!

Don't modify a collection while you're using foreach to iterate through it!
*If you do, it will throw an `InvalidOperationException`. You can see this for yourself. Create a new .NET Core console app, then add code to create a new `List<string>`, add a value to it, use `foreach` to iterate through it, and add another value to the collection **inside** the `foreach` loop. When you run your code, your `foreach` loop will throw an exception. And remember, you always specify a type when using generic classes—so `List<string>` refers to a list of strings.*

```
static void Main(string[] args)
{
    List<string> values = new List<string>();
    values.Add("a value");
    foreach (string s in values) ✘
    {
        values.Add("another value");
    }
}
```





Code Magnets

Can you rearrange the code snippets to make a working console app that will write the specified output to the console?

```
static void Main(string[] args)
{
```

```
string zilch = "zero";
string first = "one";
string second = "two";
string third = "three";
string fourth = "4.2";
string twopointtwo = "2.2";
```

```
a.Add(zilch);
a.Add(first);
a.Add(second);
a.Add(third);
```

```
static void PppPppL (List<string> a){
```

```
foreach (string element in a)
{
    Console.WriteLine(element);
}
```

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

```
if (a.IndexOf("four") != 4)
{
    a.Add(fourth);
}
```

```
a.RemoveAt(2);
```

```
if (a.Contains("three"))
{
    a.Add("four");
}
```

```
}
```

Output

```
zero
one
three
four
4.2
```

```
PppPppL(a);
```

```
if (a.Contains("two"))
{
    a.Add(twopointtwo);
}
```



Code Magnets Solution

If you want to run this code, make sure you have a "using System.Collections.Generic" line at the top.

Remember how we talked about using intuitive names back in Chapter 3? Well, that may make for good code, but it makes these puzzles a bit too easy. Just don't use cryptic names like PppPppL in real life!

Output

zero
one
three
four
4.2

```
static void Main(string[] args)
{
```

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

```
string zilch = "zero";
string first = "one";
string second = "two";
string third = "three";
string fourth = "4.2";
string twopointtwo = "2.2";
```

```
a.Add(zilch);
a.Add(first);
a.Add(second);
a.Add(third);
```

```
if (a.Contains("three"))
{
    a.Add("four");
}
```

```
a.RemoveAt(2);
```

```
if (a.IndexOf("four") != 4)
{
    a.Add(fourth);
}
```

```
if (a.Contains("two")) {
    a.Add(twopointtwo);
}
```

```
PppPppL(a);
```

```
}
```

```
static void PppPppL(List<string> a){
```

```
foreach (string element in a)
{
    Console.WriteLine(element);
}
```

```
}
```

```
}
```

RemoveAt removes the element at index #2—which is the third element in the list.

The foreach loop goes through all of the elements in the list and prints them.

Can you figure out why "2.2" never gets added to the list, even though it's declared here and passed to a.Add below? Use the debugger to sleuth it out!

The PppPppL method uses a foreach loop to go through a list of strings, add each of them to one big string, and then show it in a message box.

there are no
Dumb Questions

Q: So why would I ever use an enum instead of a collection? Don't they kind of solve somewhat similar problems?

A: Enums do very different things from collections. First and foremost, enums are **types**, while collections are **objects**.

You can think of enums as a handy way to store **lists of constants** so you can refer to them by name. They're great for keeping your code readable and making sure that you are always using the right variable names to access values that you use really frequently.

A collection can store just about anything because it stores **object references**, which can access the objects' members as usual. Enums, on the other hand, have to be assigned one of the **value types** in C# (like the ones introduced in Chapter 4). You can cast them to values, but not references.

Enums can't dynamically change their size either. They can't implement interfaces or have methods, and you'll have to cast them to another type to store a value from an enum in another variable. Add all of that up and you've got some pretty big differences between the two ways of storing data. Both are really useful in their own right.

Q: It sounds like the List class is pretty powerful. So why would I ever want to use an array?

A: If you need to store a collection of objects, you'll generally use a list and not

Arrays actually take up less memory and CPU time for your programs, but that only accounts for a tiny performance boost. If you have to do the same thing, say, millions of times a second, you might want to use an array and not a list. But if your program is running slowly, it's pretty unlikely that switching from lists to arrays will fix the problem.

an array. One place where you'll use arrays (which you'll see later in the book) is where you're reading sequences of bytes—for example, from a file. In that case, you'll often call a method on a .NET class that returns a `byte[]`. Luckily, it's easy to convert a list to an array (by calling its `ToArray` method), or an array to a List (using an overloaded list constructor).

Q: I don't get the name "generic." Why is it called a generic collection?

A: A generic collection is a collection object (or a built-in object that lets you store and manage a bunch of other objects) that's been set up to store only one type (or more than one type, as you'll see in a minute).

Q: OK, that explains the "collection" part. But what makes it "generic"?

A: Supermarkets used to carry generic items that were packaged in white wrappers with black type that just said the name of what was inside ("Potato Chips," "Cola," "Soap," etc.). The generic brand was all about what was inside the bag, and not about how it was displayed.

The same thing happens with generic data types. Your `List<T>` will work exactly the same with whatever happens to be inside it. A list of Shoe objects, Card objects, ints, longs, or even other lists will still act at the container level. So you can always add, remove, insert, etc., no matter what's inside the list itself.



The term "generic" refers to the fact that even though a specific instance of `List` can only store one specific type, the `List` class in general works with any type. That's what the `<T>` is for—it says that the list contains a bunch of references of type `T`.

Q: Can I have a list that doesn't have a type?

A: No. Every list—in fact, every generic collection (and you'll learn about the other generic collections in just a minute)—must have a type connected to it. C# does have nongeneric lists called `ArrayLists` that can store any kind of object. If you want to use an `ArrayList`, you need to include a `using System.Collections;` line in your code. You should rarely need to do this, because a `List<Object>` will usually work just fine in a scenario where you might want to use an untyped `ArrayList`.

When you create a new List object, you always supply a type—that tells C# what type of data it'll store.

A List can store a value type (like int, bool, or decimal) or a class.

Collection initializers are similar to object initializers

C# gives you a nice bit of shorthand to cut down on typing when you need to create a list and immediately add a bunch of items to it. When you create a new List object, you can use a **collection initializer** to give it a starting list of items. It'll add them as soon as the list is created.

This code creates a new List<Shoe> and fills it with new Shoe objects by calling the Add method over and over again.

```
List<Shoe> shoeCloset = new List<Shoe>();  
shoeCloset.Add(new Shoe() { Style = Style.Sneakers, Color = "Black" });  
shoeCloset.Add(new Shoe() { Style = Style.Clogs, Color = "Brown" });  
shoeCloset.Add(new Shoe() { Style = Style.Wingtips, Color = "Black" });  
shoeCloset.Add(new Shoe() { Style = Style.Loafers, Color = "White" });  
shoeCloset.Add(new Shoe() { Style = Style.Loafers, Color = "Red" });  
shoeCloset.Add(new Shoe() { Style = Style.Sneakers, Color = "Green" });
```

Notice how each Shoe object is initialized with its own object initializer? You can nest them inside a collection initializer, just like this.

The same code rewritten using a collection initializer

The statement to create the list is followed by curly brackets that contain separate "new" statements, separated by commas.

You're not limited to using "new" statements in the initializer—you can include variables, too.

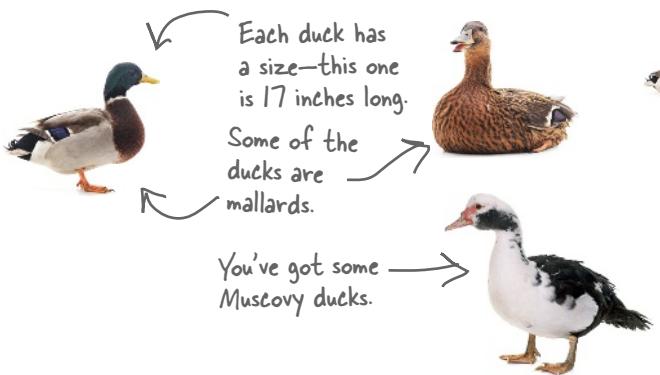
```
List<Shoe> shoeCloset = new List<Shoe>() {  
    new Shoe() { Style = Style.Sneakers, Color = "Black" },  
    new Shoe() { Style = Style.Clogs, Color = "Brown" },  
    new Shoe() { Style = Style.Wingtips, Color = "Black" },  
    new Shoe() { Style = Style.Loafers, Color = "White" },  
    new Shoe() { Style = Style.Loafers, Color = "Red" },  
    new Shoe() { Style = Style.Sneakers, Color = "Green" },  
};
```

You can create a collection initializer by taking each item that was being added using Add and adding it to the statement that creates the list.

A collection initializer makes your code more compact by letting you combine creating a list with adding an initial set of items.

Let's create a List of Ducks

Here's a Duck class that keeps track of your many neighborhood ducks. (You *do* collect ducks, don't you?) **Create a new Console App project** and add a new Duck class and KindOfDuck enum.



Here's the initializer for your List of Ducks

You've got six ducks, so you'll create a `List<Duck>` that has a collection initializer with six statements. Each statement in the initializer creates a new Duck, using an object initializer to set each Duck object's Size and Kind fields. Make sure this **using directive** is at the top of `Program.cs`:

```
using System.Collections.Generic;
```

Then **add this PrintDucks method** to your Program class:

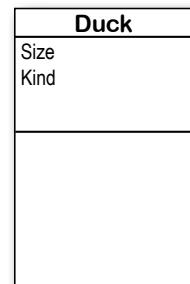
```
public static void PrintDucks(List<Duck> ducks)
{
    foreach (Duck duck in ducks) {
        Console.WriteLine($"{duck.Size} inch {duck.Kind}");
    }
}
```

Finally, **add this code** to your Main method in `Program.cs` to create a List of Ducks and then print them:

```
List<Duck> ducks = new List<Duck>() {
    new Duck() { Kind = KindOfDuck.Mallard, Size = 17 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 18 },
    new Duck() { Kind = KindOfDuck.Loon, Size = 14 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 11 },
    new Duck() { Kind = KindOfDuck.Mallard, Size = 14 },
    new Duck() { Kind = KindOfDuck.Loon, Size = 13 },
};
```

```
PrintDucks(ducks);
```

Do this!



```
class Duck {
    public int Size {
        get; set;
    }
    public KindOfDuck Kind {
        get; set;
    }
}

enum KindOfDuck {
    Mallard,
    Muscovy,
    Loon,
}
```

Add Duck and KindOfDuck to your project. You'll use the KindOfDuck enum to keep track of what sorts of ducks are in your collection. Notice that we're not assigning values—that's very typical. We won't need numerical values for ducks, so the default enum values (0, 1, 2,...) will be just fine.

Run your code—it will print a bunch of Ducks to the console.

Lists are easy, but SORTING can be tricky

It's not hard to think about ways to sort numbers or letters. But how do you sort two individual objects, especially if they have multiple fields? In some cases you might want to order objects by the value in the Name field, while in other cases it might make sense to order objects based on height or date of birth. There are lots of ways you can order things, and lists support all of them.

You could sort a list of ducks by size...



Sorted smallest to biggest....

...or by kind.

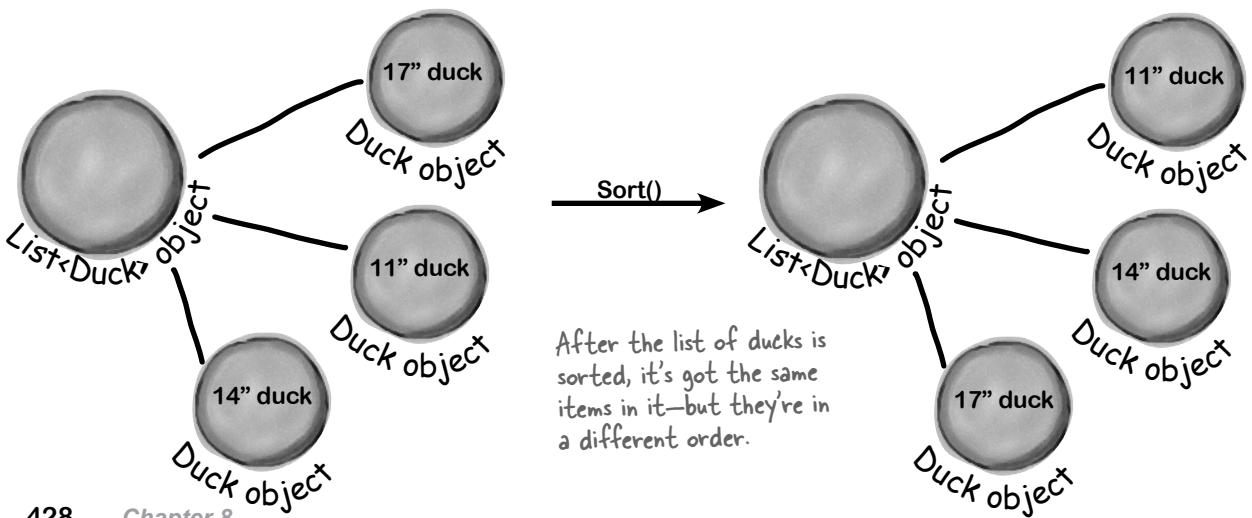


Sorted by kind of duck...

Lists know how to sort themselves

Every List comes with a **Sort method** that rearranges all of the items in the list to put them in order. Lists already know how to sort most built-in types and classes, and it's easy to teach them how to sort your own classes.

Technically, it's not the `List<T>` that knows how to sort itself. This is the job of an `IComparer<T>` object, which you'll learn about in a minute.



IComparable<Duck> helps your List sort its Ducks

If you have a List of numbers and call its Sort method, it will sort the list with the smallest numbers first and largest last. How does the List know which way to sort the Duck objects? We tell List.Sort that the Duck class can be sorted—and we do that the way we typically indicate that a class can do a certain job, *with an interface*.

The List.Sort method knows how to sort any type or class that **implements the IComparable<T> interface**. That interface has just one member—a method called CompareTo. Sort uses an object's CompareTo method to compare it with other objects, and uses its return value (an int) to determine which comes first.

An object's CompareTo method compares it to another object

One way to give our List object the ability to sort ducks is to **modify the Duck class to implement IComparable<Duck>** and add its only member, a CompareTo method that takes a Duck reference as a parameter.

Update your project's Duck class by implementing IComparable<Duck> so that it sorts itself based on duck size:

Most CompareTo methods look a lot like this. This method first compares the Size field against the other Duck's Size field. If this Duck is bigger, it returns 1. If it's smaller, it returns -1. And if they're the same size, it returns 0.

```
class Duck : IComparable<Duck> {
    public int Size { get; set; }
    public KindOfDuck Kind { get; set; }

    public int CompareTo(Duck duckToCompare) {
        if (this.Size > duckToCompare.Size)
            return 1;
        else if (this.Size < duckToCompare.Size)
            return -1;
        else
            return 0;
    }
}
```

You specify the type being compared when you have the class implement the IComparable<T> interface.

If you want to sort your list from smallest to biggest, make CompareTo return a positive number if it's comparing to a smaller duck, and a negative number if it's comparing to a bigger one.

If the duck to compare should come after the current duck in the sorted list, CompareTo needs to return a positive number. If it should come before, CompareTo returns a negative number. If they're the same, it returns zero.

Add this line of code to the end of your Main method just before the call to PrintDucks. This tells your list of ducks to sort itself. Now it sorts the ducks by size before printing them to the console:

```
ducks.Sort();
PrintDucks(ducks);
```



Use IComparer to tell your List how to sort

Your Duck class implements IComparable, so List.Sort knows how to sort a List of Duck objects. But what if you want to sort them in a different way than usual? Or what if you want to sort a type of object that doesn't implement IComparable? Then you can pass a **comparer object** as an argument to List.Sort, to give it a different way to sort its objects. Notice how List.Sort is overloaded:

▲ 3 of 4 ▼ void List<Duck>.Sort(IComparer<Duck>? comparer)

Sorts the elements in the entire List<T> using the specified comparer.

comparer: The IComparer<in T> implementation to use when comparing elements, or null to use the default comparer Comparer<T>.Default.

There's an overloaded version of List.Sort that **takes an IComparer<T> reference**, where T will be replaced by the generic type for your list (so for a List<Duck> it takes an IComparer<Duck> argument, for a List<string> it's an IComparer<string>, etc.). You'll pass it a reference to an object that implements an interface, and we know what that means: that it *does a specific job*. In this case, that job is comparing pairs of items in the list to tell List.Sort what order to sort them in.

The IComparer<T> interface has one member, a **method called Compare**. It's just like the CompareTo method in IComparable<T>; it takes two object parameters, x and y, and returns an a positive value if x comes before y, a negative value if x comes after y, or zero if they're the same.

Add an IComparer to your project

Add the DuckComparerBySize class to your project. It's a comparer object that you can pass as a parameter to List.Sort to make it sort your ducks by size.

The IComparer interface is in the System.Collections.Generic namespace, so if you're adding this class to a new file make sure it has the right **using** directive:

```
using System.Collections.Generic;
```

Here's the code for the comparer class:

```
class DuckComparerBySize : IComparer<Duck>
{
    public int Compare(Duck x, Duck y)
    {
        if (x.Size < y.Size)
            return -1;
        if (x.Size > y.Size) ←
            return 1;
        return 0; ←
    }
}
```

If Compare returns a negative number, that means object x should go before object y. x is "less than" y.

Any positive value means object x should go after object y. x is "greater than" y.

Zero means they're "equal."

A comparer object is an instance of a class that implements IComparer<T> that you can pass as a reference to List.Sort. Its Compare method works just like the ComapreTo method in the IComparable<T> interface. When List.Sort compares its elements to sort them, it passes pairs of objects to the Compare method in your comparer object, so your List will sort differently depending on how you implement the comparer.

Can you figure out how to modify DuckComparerBySize so it sorts the ducks largest to smallest instead?

Create an instance of your comparer object

When you want to sort using `IComparer<T>`, you need to create a new instance of the class that implements it—in this case, `Duck`. That's the comparer object that will help `List.Sort` figure out how to sort its elements. Like any other (nonstatic) class, you need to instantiate it before you use it:

```
IComparer<Duck> sizeComparer = new DuckComparerBySize();
ducks.Sort(sizeComparer);
PrintDucks(ducks);
```

You'll pass `Sort` a reference to the new `DuckComparerBySize` object as its parameter.

Replace `ducks.Sort` in your `Main` method with these two lines of code. It still sorts the ducks, but now it uses the comparer object.



Multiple `IComparer` implementations, multiple ways to sort your objects

You can create multiple `IComparer<Duck>` classes with different sorting logic to sort the ducks in different ways. Then you can use the comparer you want when you need to sort in that particular way. Here's another duck comparer implementation to add to your project:

```
class DuckComparerByKind : IComparer<Duck> {
    public int Compare(Duck x, Duck y) {
        if (x.Kind < y.Kind)
            return -1;
        if (x.Kind > y.Kind)
            return 1;
        else
            return 0;
    }
}
```

We compared the ducks' Kind properties, so the ducks are sorted based on the index value of the Kind property, a `KindOfDuck` enum.

Notice how "greater than" and "less than" have a different meaning here. We used `<` and `>` to compare enum index values, which lets us put the ducks in order.

This comparer sorts by duck type. Remember, when you compare the enum `Kind`, you're comparing their enum index values. We didn't assign values when we declared the `KindOfDuck` enum, so they're given values 0, 1, 2, etc. in the order they appear in the enum declaration (so Mallard is 0, Muscovy is 1, and Loon is 2).

Here's an example of how enums and lists work together. Enums stand in for numbers, and are used in sorting of lists.

Go back and modify your program to use this new comparer. Now it sorts the ducks by kind before it prints them.

```
IComparer<Duck> kindComparer = new DuckComparerByKind();
ducks.Sort(kindComparer);
PrintDucks(ducks);
```



Comparers can do complex comparisons

One advantage to creating a separate class for sorting your ducks is that you can build more complex logic into that class—and you can add members that help determine how the list gets sorted.

```
enum SortCriteria {
    SizeThenKind,
    KindThenSize,
}
This enum tells the comparer
object which way to sort the ducks.
```

```
class DuckComparer : IComparer<Duck> {
    public SortCriteria SortBy = SortCriteria.SizeThenKind;
```

```
public int Compare(Duck x, Duck y) {
    if (SortBy == SortCriteria.SizeThenKind)
        if (x.Size > y.Size)
            return 1;
        else if (x.Size < y.Size)
            return -1;
        else
            if (x.Kind > y.Kind)
                return 1;
            else if (x.Kind < y.Kind)
                return -1;
            else
                return 0;
    else
        if (x.Kind > y.Kind)
            return 1;
        else if (x.Kind < y.Kind)
            return -1;
        else
            if (x.Size > y.Size)
                return 1;
            else if (x.Size < y.Size)
                return -1;
            else
                return 0;
}
```

```
DuckComparer comparer = new DuckComparer();
Console.WriteLine("\nSorting by kind then size\n");
comparer.SortBy = SortCriteria.KindThenSize;
ducks.Sort(comparer);
PrintDucks(ducks);
Console.WriteLine("\nSorting by size then kind\n");
comparer.SortBy = SortCriteria.SizeThenKind;
ducks.Sort(comparer);
PrintDucks(ducks);
```

Here's a more complex class to compare ducks. Its `Compare` method takes the same parameters, but it looks at the public `SortBy` field to determine how to sort the ducks.

This "if" statement checks the `SortBy` field. If it's set to `SizeThenKind`, then it first sorts the ducks by size, and then within each size it'll sort the ducks by their kind.

Instead of just returning 0 if the two ducks are the same size, the comparer checks their kind, and only returns 0 if the two ducks are both the same size and the same kind.

If `SortBy` isn't set to `SizeThenKind`, then the comparer first sorts by the kind of duck. If the two ducks are the same kind, then it compares their size.

Add this code to the end of your `Main` method. It uses the `comparer` object, setting its `SortBy` field before calling `ducks.Sort`. Now you can change the way the list sorts its ducks by changing a property in the comparer.



**Exercise**

Build a console app that creates a list of cards in random order, prints them to the console, uses a comparer object to sort the cards, and then prints the sorted list.

**1 Write a method to make a jumbled set of cards.**

Create a new console app. Add the `Suits` enum, `Values` enum, and `Card` class from earlier in the chapter. Then add two static methods to `Program.cs`: a `RandomCard` method that returns a reference to a card with a random suit and value, and a `PrintCards` method that prints a `List<Card>`.

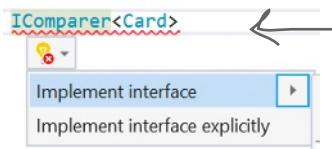
2 Create a class that implements `IComparer<Card>` to sort the cards.

Here's a good opportunity to use the IDE's **Quick Actions menu** to implement an interface. Add a class called `CardComparerByValue`, then make it implement the `IComparer<Card>` interface:

```
class CardComparerByValue : IComparer<Card>
```

Click on `IComparer<Card>` and hover over the `I`. You'll see a light bulb icon (💡 or 💡) appear. When you click on the icon, the IDE pops up its Quick Actions menu:

The IDE has a useful shortcut
that makes it easy to access the
Quick Actions menu: press `Ctrl+T`.
(Windows) or `Option+Enter` (Mac).



Your `IComparer` object
needs to sort the cards
by value, so the cards
with the lowest values are
first in the list.

Choose “Implement interface” —that tells the IDE to automatically fill in all of the methods and properties in the interface that you need to implement. In this case, it creates an empty `Compare` method to compare two cards, `x` and `y`. Make it return `1` if `x` is bigger than `y`, `-1` if it's smaller, and `0` if they're the same. First, order by suit: Diamonds come first, then Clubs, then Hearts, then Spades. Make sure that any King comes after any Jack, which comes after any 4, which comes after any Ace. You can compare enum values without casting: `if (x.Suit < y.Suit)`

Enter number of cards: 9
Eight of Spades

Nine of Hearts

Four of Hearts

Nine of Hearts

King of Diamonds

King of Spades

Six of Spades

Seven of Clubs

Seven of Clubs

... sorting the cards ...

King of Diamonds

Seven of Clubs

Seven of Clubs

Four of Hearts

Nine of Hearts

Nine of Hearts

Six of Spades

Eight of Spades

King of Spades

3 Make sure the output looks right.

Write the `Main` method so the output looks like this. →

- ★ It prompts for a number of cards.
- ★ If the user enters a valid number and presses Enter, it generates a list of random cards and then prints them.
- ★ It sorts the list of cards using the comparer.
- ★ It prints the sorted list of cards.



Exercise Solution

```

class CardComparerByValue : IComparer<Card>
{
    public int Compare(Card x, Card y)
    {
        if (x.Suit < y.Suit)           } We want all Diamonds to come
                                         before all Clubs, so we need to
        return -1;                     compare the suits first. We can
                                         take advantage of the enum values.
        if (x.Suit > y.Suit)           }
                                         { These statements only get
                                         { executed if x and y have the
                                         same value—that means the
                                         first two return statements
                                         weren't executed.
                                         { If none of the other four return
                                         { statements were hit, the cards
                                         must be the same—so return zero.
                                         }

class Program
{
    private static readonly Random random = new Random();

    static Card RandomCard()
    {
        return new Card((Values)random.Next(1, 14), (Suits)random.Next(4));
    }

    static void PrintCards(List<Card> cards)
    {
        foreach (Card card in cards)
        {
            Console.WriteLine(card.Name);
        }
    }

    static void Main(string[] args)
    {
        List<Card> cards = new List<Card>();
        Console.WriteLine("Enter number of cards: ");
        if (int.TryParse(Console.ReadLine(), out int numberOfCards))
            for (int i = 0; i < numberOfCards; i++)
                cards.Add(RandomCard());

        PrintCards(cards);

        cards.Sort(new CardComparerByValue());
        Console.WriteLine("\n... sorting the cards ...");

        PrintCards(cards);
    }
}

```

Here's the "guts" of the card sorting, which uses the built-in `List.Sort` method. Sort takes an `IComparer` object, which has one method: `Compare`. This implementation takes two cards and first compares their values, then their suits.

Here's where we create a generic `List` of `Card` objects to store the cards. Once they're in the list, it's easy to sort them using an `IComparer`.

We left out the curly brackets here. Do you think it makes the code easier or harder to read?

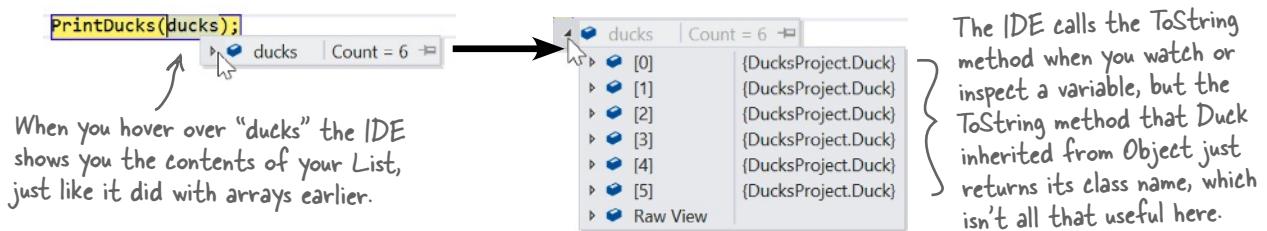
Overriding a `ToString` method lets an object describe itself

Every object has a **method called `ToString` that converts it to a string**. You've already used it—any time you use `{curly braces}` in string interpolation that calls the `ToString` method of whatever's inside them—and the IDE also takes advantage of it. When you create a class, it inherits the `ToString` method from `Object`, the top-level base class that all other classes extend.

The `Object.ToString` method prints the **fully qualified class name**, or the namespace followed by a period followed by the class name. Since we used the namespace `DucksProject` when we were writing this chapter, the fully qualified class name for our `Duck` class is `DucksProject.Duck`:

```
Console.WriteLine(new Duck().ToString()); → "DucksProject.Duck"
```

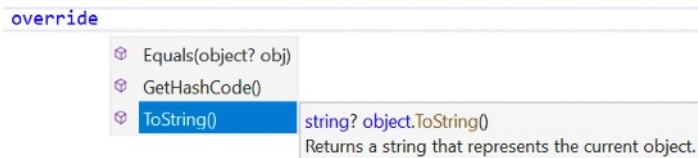
The IDE also calls the `ToString` method—for example, when you watch or inspect a variable:



Hmm, that's not as useful as we'd hoped. You can see that there are six `Duck` objects in the list. If you expand a `Duck`, you can see its `Kind` and `Size` values. Wouldn't it be easier if you could see all of them at once?

Override the `ToString` method to see your Ducks in the IDE

Luckily, `ToString` is a virtual method of `Object`, the base class of every object. So all you need to do is **override the `ToString` method**—and when you do, you'll see the results immediately in the IDE's Watch window! Open up your `Duck` class and start adding a new method by typing `override`. As soon as you add a space, the IDE will show you the methods you can override:

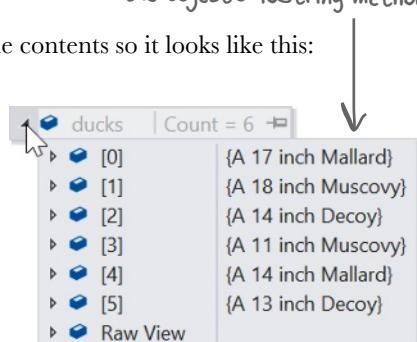


When the IDE's debugger shows you an object, it calls the object's `ToString` method.

Click on `ToString()` to tell the IDE to add a new `ToString` method. Replace the contents so it looks like this:

```
public override string ToString()
{
    return $"A {Size} inch {Kind}";
}
```

Run your program and look at the list again. Now the IDE shows you the contents of your `Duck` objects.



Update your foreach loops to let your Ducks and Cards write themselves to the console

You've seen two different examples of programs looping through a list of objects and calling `Console.WriteLine` to print a line to the console for each object—like this `foreach` loop that prints every `Card` in a `List<Card>`:

```
foreach (Card card in cards)
{
    Console.WriteLine(card.Name);
}
```

The `PrintDucks` method did something similar for Duck objects in a `List`:

```
foreach (Duck duck in ducks) {
    Console.WriteLine($"{duck.Size} inch {duck.Kind}");
}
```

This is a pretty common thing to do with objects. Now that your Duck has a `ToString` method, your `PrintDucks` method should take advantage of it. Use the IDE's IntelliSense to look through the overloads for the `Console.WriteLine` method—specifically this one:

▲ 10 of 18 ▼ **void Console.WriteLine(object value)**



If you pass `Console.WriteLine` a reference to an object, it will call that object's `ToString` method automatically.

You can pass any object to `Console.WriteLine`, and it will call its `ToString` method. So you can replace the `PrintDucks` method with one that calls this overload:

```
public static void PrintDucks(List<Duck> ducks) {
    foreach (Duck duck in ducks) {
        Console.WriteLine(duck);
    }
}
```

Replace the `PrintDucks` method with this one and run your code again. It prints the same output. If you want to add, say, a `Color` or `Weight` property to your Duck object, you just have to update the `ToString` method, and everything that uses it (including the `PrintDucks` method) will reflect that change.

Add a `ToString` method to your `Card` object, too

Your `Card` object already has a `Name` property that returns the name of the card:

```
public string Name { get { return $"{Value} of {Suit}"; } }
```

That's exactly what its `ToString` method should do. So, add a `ToString` method to the `Card` class:

```
public override string ToString()
{
    return Name;
}
```

Now your programs that use `Card` objects will be easier to debug.

We decided to make the `ToString` method call the `Name` property. Do you think we made the right choice? Would it have been better to delete the `Name` property and move its code to the `ToString` method? When you're going back to modify your code, you have to make choices like this—and it's not always obvious which choice is best.

Sharpen your pencil



Read through this code and write the output below the code.

```
enum Breeds
{
    Collie = 3,
    Corgi = -9,
    Dachshund = 7,
    Pug = 0,
}
class Dog : IComparable<Dog>
{
    public Breeds Breed { get; set; }
    public string Name { get; set; }

    public int CompareTo(Dog other)
    {
        if (Breed > other.Breed) return -1;
        if (Breed < other.Breed) return 1;
        return -Name.CompareTo(other.Name);
    }

    public override string ToString()
    {
        return $"A {Breed} named {Name}";
    }
}

class Program
{
    static void Main(string[] args)
    {
        List<Dog> dogs = new List<Dog>()
        {
            new Dog() { Breed = Breeds.Dachshund, Name = "Franz" },
            new Dog() { Breed = Breeds.Collie, Name = "Petunia" },
            new Dog() { Breed = Breeds.Pug, Name = "Porkchop" },
            new Dog() { Breed = Breeds.Dachshund, Name = "Brunhilda" },
            new Dog() { Breed = Breeds.Collie, Name = "Zippy" },
            new Dog() { Breed = Breeds.Corgi, Name = "Carrie" },
        };
        dogs.Sort();
        foreach (Dog dog in dogs)
            Console.WriteLine(dog);
    }
}
```

Hint—pay
attention to
the minus signs!

This app writes six lines to the console. Can you figure out what they are and write them down here? See if you can figure it out just from reading the code, and without running the app.

Here's the output of the app. Did you get it right? It's OK if you didn't! Go back and take another look at the enum.

- Did you notice that the enumerators had different values?
- The Name property is a string, and strings also implement `IComparable`, so we can just call their `CompareTo` method to compare them.
- Also, take a closer look at the `CompareTo` method—did you notice that it returned `-1` if the other breed was greater and `1` if the other breed was less, or the minus sign before `-Name.CompareTo(other.Name)`? So first it's sorting by Breed and then by Name, but it's sorting both Breed and Name in reverse order.

Here's the output:

A Dachshund named Franz

A Dachshund named Brunhilda

A Collie named Zippy

A Collie named Petunia

A Pug named Porkechop

A Corgi named Carrie

Sharpen your pencil Solution



When `CompareTo` uses `>` and `<` to compare Breed values, it uses the int values in the Breed enum declaration, so Collie is `3`, Corgi is `-9`, Dachshund is `7`, and Pug is `0`.

BULLET POINTS

- Collection initializers** let you specify the contents of a `List<T>` or other collection when you create it, using angle brackets with a comma-separated list of objects.
- A collection initializer makes your code more **compact** by letting you combine list creation with adding an initial set of items (but your code won't run more quickly).
- The **List.Sort method** sorts the contents of the collection, changing the order of the items it contains.
- The **`IComparable<T>` interface** contains a single method, `CompareTo`, which `List.Sort` uses to determine the order of objects to sort.
- An **overloaded method** is a method that you can call in more than one way, with different arrangements of parameters. The IDE's IntelliSense pop-up lets you scroll through the different overloads for a method.
- The Sort method has an overload that takes an **`IComparer<T>` object**, which it will then use for sorting.
- `IComparable.CompareTo` and `IComparer.Compare` both **compare pairs of objects**, returning `-1` if the first object is less than the second, `1` if the first is greater than the second, or `0` if they're equal.
- The **String class implements `IComparable`**. An `IComparer` or `IComparable` for a class that includes a string member can call its `Compare` or `CompareTo` method to help determine the sort order.
- Every object has a **`ToString` method** that converts it to a string. The `ToString` method is called any time you use string interpolation or concatenation.
- The default `ToString` method is inherited from `Object`. It returns the **fully qualified class name**, or the namespace followed by a period followed by the class name.
- Override the `ToString` method** to get interpolation, concatenation, and many other operations to use a custom string.

Foreach Loops Up Close



Let's take a closer look at foreach loops. Go to the IDE, find a List<Duck> variable, and use IntelliSense to take a look at its GetEnumerator method. Start typing “.GetEnumerator” and see what comes up:

```
ducks.GetEnumerator();
```

List<Duck>.Enumerator List<Duck>.GetEnumerator()
Returns an enumerator that iterates through the List<T>.

Create an Array[Duck] and do the same thing—the array also has a GetEnumerator method. That’s because lists, arrays, and other collections implement an interface called **IEnumerable<T>**.

You already know that interfaces are all about making different objects do the same job. When an object implements the **IEnumerable<T>** interface, the specific job it does is that **it supports iteration over a nongeneric collection**—or in other words, it lets you write code that loops through it. Specifically, that means you can use it with a foreach loop.

So what does that look like under the hood? Use Go To Definition/Declaration on **List<Duck>** to see the interfaces that it implements, just like you did earlier. Then do it again to see the members of **IEnumerable<T>**. What do you see?

The **IEnumerable<T>** interface contains a single member: a method called **GetEnumerator**, which returns an **Enumerator object**. The Enumerator object provides the machinery that lets you loop through a list in order. So when you write this foreach loop:

```
foreach (Duck duck in ducks) {
    Console.WriteLine(duck);
}
```

Here’s what that loop is actually doing behind the scenes:

```
IEnumerator<Duck> enumerator = ducks.GetEnumerator();
while (enumerator.MoveNext()) {
    Duck duck = enumerator.Current;
    Console.WriteLine(duck);
}
if (enumerator is IDisposable disposable) disposable.Dispose();
```

When a collection implements **IEnumerable<T>**, it’s giving you a way to write a loop that goes through its contents in order.

Technically, there’s a little more code than this, but this is enough to give you the basic idea of what’s going on.

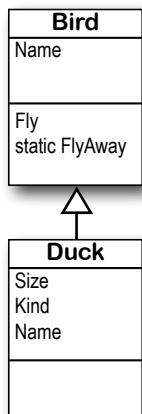
Both loops write the same Ducks to the console. See this for yourself by running both of them—they’ll both have the same output. (And don’t worry about the last line for now; you’ll learn about **IDisposable** in Chapter 10.)

When you’re looping through a list or array (or any other collection), the **MoveNext** method returns true if there’s another element in the collection, or false if the enumerator has reached the end. The **Current** property always returns a reference to the current element. Add it all together, and you get a foreach loop.

You can upcast an entire list using `IEnumerable<T>`

Remember how you can upcast any object to its superclass? Well, when you've got a List of objects, you can upcast the entire list at once. It's called **covariance**, and all you need for it is an `IEnumerable<T>` interface reference.

Let's see how this works. We'll start with the Duck class that you've been working with throughout the chapter. Then we'll add a Bird class that it will extend. The Bird class will include a static method that iterates over a collection of Bird objects. Can we get it to work with a List of Ducks?



Here's the Bird class that you'll make your Duck class extend. You'll change its declaration to extend Bird, but leave the rest of the class the same. Then you'll add them both to a console app so you can experiment with covariance.

Do this!

Since all Ducks are Birds, covariance lets us convert a collection of Ducks to a collection of Birds. That can be really useful if you have to pass a `List<Duck>` to a method that only accepts a `List<Bird>`.

- ① Create a new Console App project. Add a base class, Bird (for Duck to extend), and a Penguin class. We'll use the `ToString` method to make it easy to see which class is which.

```

class Bird
{
    public string Name { get; set; }
    public virtual void Fly(string destination)
    {
        Console.WriteLine($"{this} is flying to {destination}");
    }

    public override string ToString()
    {
        return $"A bird named {Name}";
    }

    public static void FlyAway(List<Bird> flock, string destination)
    {
        foreach (Bird bird in flock)
        {
            bird.Fly(destination);
        }
    }
}
  
```

Covariance is C#'s way of letting you implicitly convert a subclass reference to its superclass. That word "implicitly" just means C# can figure out how to do the conversion without you needing to explicitly use casting.

The static `FlyAway` method works with a collection of Birds. But what if we want to pass a List of Ducks to it?

- ② Add your Duck class to the application. Modify its declaration to **make it extend Bird**. You'll also need to **add the KindOfDuck enum** from earlier in the chapter:

```
class Duck : Bird {
    public int Size { get; set; }
    public KindOfDuck Kind { get; set; }

    public override string ToString()
    {
        return $"A {Size} inch {Kind}";
    }
}
```

We added Bird to the declaration to make the Duck class extend Bird. The rest of the Duck class is exactly the same as in your earlier project.

```
enum KindOfDuck {
    Mallard,
    Muscovy,
    Loon,
}
```

The Duck.Kind property uses KindOfDuck, so you'll need to add it too.

- ③ Create the List<Duck> collection. Go ahead and **add this code to your Main method**—it's the code from earlier in the chapter, plus one line to upcast it to a List<Bird>:

```
List<Duck> ducks = new List<Duck>() {
    new Duck() { Kind = KindOfDuck.Mallard, Size = 17 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 18 },
    new Duck() { Kind = KindOfDuck.Loon, Size = 14 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 11 },
    new Duck() { Kind = KindOfDuck.Mallard, Size = 14 },
    new Duck() { Kind = KindOfDuck.Loon, Size = 13 },
};
```

```
Bird.FlyAway(ducks, "Minnesota");
```

Copy the same collection initializer you've been using to initialize your List of Ducks.

 CS1503 Argument 1: cannot convert from 'System.Collections.Generic.List<BirdCovariance.Duck>' to 'System.Collections.Generic.List<BirdCovariance.Bird>'

Uh-oh—that code won't compile. The error message is telling you that you can't convert your Duck collection to a Bird collection. Let's try assigning **ducks** to a List<Bird>:

```
List<Bird> upcastDucks = ducks;
```

Well, that didn't work. We got a different error, but it still says we can't convert the type:

 CS0029 Cannot implicitly convert type 'System.Collections.Generic.List<BirdCovariance.Duck>' to 'System.Collections.Generic.List<BirdCovariance.Bird>'

Which makes sense—it's exactly like safely upcasting versus downcasting, which you learned about in Chapter 6: we can use assignment to downcast, but we need to use the **is** keyword to safely upcast. So how do we safely upcast our List<Duck> to a List<Bird>?

- ④ Use covariance to make your ducks fly away. That's where **covariance** comes in: you can **use assignment to upcast your List<Duck> to an IEnumerable<Bird>**. Once you've got your I Enumerable<Bird>, you can call its ToList method to convert it to a List<Bird>. You'll need to add using System.Collections.Generic; and using System.Linq; to the top of the file:

```
IEnumerable<Bird> upcastDucks = ducks;
Bird.FlyAway(upcastDucks.ToList(), "Minnesota");
```

Now your collection of Duck references has been converted to a collection of Bird references. Fly away, little ducks!

Use a Dictionary to store keys and values

A list is like a big long page full of names. What if you also want, for each name, an address? Or for every car in the **garage** list, you want details about that car? You need another kind of .NET collection: a **dictionary**. A dictionary lets you take a special value—the **key**—and associate that key with a bunch of data—the **value**. One more thing: a specific key can **only appear once** in any dictionary.

In a real-world dictionary, the word being defined is the **key**. You use it to look up the **value**, or the definition of the word.

dic·tion·ar·y, noun.

a book that lists the words of a language in alphabetical order and gives their meaning

The definition is the **value**. It's the data associated with a particular key (in this case, the word being defined).

Here's how you declare a .NET Dictionary in C#:

Dictionary< TKey, TValue > dict = new Dictionary< TKey, TValue >();

These are the generic types for the Dictionary. **TKey** is the type used for the key that looks up the values, and **TValue** is the type of the values. So if you're storing words and their definitions, you'd use a **Dictionary<string, string>**. If you wanted to keep track of the number of times each word appears in a book, you could use **Dictionary<string, int>**.

These represent types. The first type in the angle brackets is always the key, and the second is always the data.

Let's see a dictionary in action. Here's a small console app that uses a **Dictionary<string, string>** to keep track of the favorite foods of a few friends:

```
using System.Collections.Generic;           ← You need this "using" directive to work
                                         with Dictionary, like you do with List.

class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, string> favoriteFoods = new Dictionary<string, string>();
        favoriteFoods["Alex"] = "hot dogs";
        favoriteFoods["A'ja"] = "pizza";
        favoriteFoods["Jules"] = "falafel";
        favoriteFoods["Naima"] = "spaghetti";           } We're adding four key/value pairs to our dictionary.
                                                       In this case, the key is a person's name, and the
                                                       value is that person's favorite food.

        string name;
        while ((name = Console.ReadLine()) != "")
        {
            if (favoriteFoods.ContainsKey(name))
                Console.WriteLine($"{name}'s favorite food is {favoriteFoods[name]}");
            else
                Console.WriteLine($"I don't know {name}'s favorite food");
        }
    }
}
```

A dictionary's **ContainsKey** method returns true if it contains a value for a specific key.

Here's how you get the value for a key.

The Dictionary functionality rundown

Dictionaries are a lot like lists. Both types are flexible in letting you work with lots of data types, and also come with lots of built-in functionality. Here are the basic things you can do with a dictionary.

★ Add an item.

You can add an item to a dictionary using its **indexer** with square brackets:

```
Dictionary<string, string> myDictionary = new Dictionary<string, string>();
myDictionary["some key"] = "some value";
```

You can also add an item to a dictionary using its **Add method**:

```
Dictionary<string, string> myDictionary = new Dictionary<string, string>();
myDictionary.Add("some key", "some value");
```

★ Look up a value using its key.

The most important thing you'll do with a dictionary is **look up values with the indexer**—which makes sense, because you stored those values in a dictionary so you could look them up using their unique keys. This example shows a `Dictionary<string, string>`, so we'll look up values using a string key, and the dictionary returns a string value:

```
string lookupValue = myDictionary["some key"];
```

★ Remove an item.

Just like with a list, you can remove an item from a dictionary using the **Remove method**.

All you need to pass to the Remove method is the key value to have both the key and the value removed:

```
myDictionary.Remove("some key");
```

Keys are unique in a dictionary; any key appears exactly once. Values can appear any number of times—two keys can have the same value. That way, when you look up or remove a key, the dictionary knows what to remove.

★ Get a list of keys.

You can get a list of all of the keys in a dictionary using its **Keys property** and loop through it using a foreach loop. Here's what that would look like:

```
foreach (string key in myDictionary.Keys) { ... };
```

← Keys is a property of your Dictionary object. This particular dictionary has string keys, so Keys is a collection of strings.

★ Count the pairs in the dictionary.

The **Count property** returns the number of key/value pairs that are in the dictionary:

```
int howMany = myDictionary.Count;
```

It's common to see a dictionary that maps integers to objects when you're assigning unique ID numbers to objects.

Your key and value can be different types

Dictionaries are versatile! They can hold just about anything, not just value types but **any kind of object**. Here's an example of a dictionary that's storing an integer as a key and a Duck object reference as a value:

```
Dictionary<int, Duck> duckIds = new Dictionary<int, Duck>();
duckIds.Add(376, new Duck() { Kind = KindOfDuck.Mallard, Size = 15 });
```

Build a program that uses a dictionary



Here's a quick app that New York Yankees baseball fans will like. When an important player retires, the team retires the player's jersey number. **Create a new console app** that looks up some Yankees who wore famous numbers and when those numbers were retired. Here's a class to keep track of a retired baseball player:

```
class RetiredPlayer
{
    public string Name { get; private set; }
    public int YearRetired { get; private set; }

    public RetiredPlayer(string player, int yearRetired)
    {
        Name = player;
        YearRetired = yearRetired;
    }
}
```

And here's the Program class with a Main method that adds retired players to a dictionary. We can use the jersey number as the dictionary key because it's **unique**—once a jersey number is retired, the team **never uses it again**. That's an important thing to consider when designing an app that uses a dictionary: you never want to discover your key is not as unique as you thought!

Yogi Berra was #8 for the New York Yankees, while Cal Ripken Jr. was #8 for the Baltimore Orioles. But in a Dictionary, you can have duplicate values, but every key must be unique. Can you think of a way to store retired numbers for multiple teams?

```
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        Dictionary<int, RetiredPlayer> retiredYankees = new Dictionary<int, RetiredPlayer>() {
            {3, new RetiredPlayer("Babe Ruth", 1948)},
            {4, new RetiredPlayer("Lou Gehrig", 1939)},
            {5, new RetiredPlayer("Joe DiMaggio", 1952)},
            {7, new RetiredPlayer("Mickey Mantle", 1969)},
            {8, new RetiredPlayer("Yogi Berra", 1972)},
            {10, new RetiredPlayer("Phil Rizzuto", 1985)},
            {23, new RetiredPlayer("Don Mattingly", 1997)},
            {42, new RetiredPlayer("Jackie Robinson", 1993)},
            {44, new RetiredPlayer("Reggie Jackson", 1993)},
        };

        foreach (int jerseyNumber in retiredYankees.Keys)
        {
            RetiredPlayer player = retiredYankees[jerseyNumber];
            Console.WriteLine($"{player.Name} #{jerseyNumber} retired in {player.YearRetired}");
        }
    }
}
```

Use a collection initializer to populate your Dictionary with JerseyNumber objects.



Use a foreach loop to iterate through the keys and write a line for each retired player in the collection.

And yet MORE collection types...

List and Dictionary are two of the most frequently used collection types that are part of the .NET. Lists and dictionaries are very flexible—you can access any of the data in them in any order. But sometimes you’re using a collection to represent a bunch of things in the real world that need to be accessed in a specific order. You can restrict how your code accesses the data in a collection by using **a Queue or a Stack**. Those are generic collections like `List<T>`, but they’re especially good at making sure that your data is processed in a certain order.

There are other types of collections, too—but these are the ones that you’re most likely to come in contact with.

Use a Queue when the first object you store will be the first one you'll use, like with:

- ★ Cars moving down a one-way street
- ★ People standing in line
- ★ Customers on hold for a customer service support line
- ★ Anything else that’s handled on a first-come, first-served basis

A queue is first in, first out, which means that the first object that you put into the queue is the first one you pull out of it to use.

Use a Stack when you always want to use the object you stored most recently, like with:

- ★ Furniture loaded into the back of a moving truck
- ★ A stack of books where you want to read the most recently added one first
- ★ People boarding or leaving a plane
- ★ A pyramid of cheerleaders, where the ones on top have to dismount first... imagine the mess if the one on the bottom walked away first!

A stack is last in, first out: the first object that goes into the stack is the last one that comes out of it.

Generic .NET collections implement `IEnumerable`

Almost every large project that you’ll work on will include some sort of generic collection, because your programs need to store data. When you’re dealing with groups of similar things in the real world, they almost always naturally fall into a category that corresponds pretty well to one of these kinds of collections. No matter which of these collection types you use—List, Dictionary, Stack, or Queue—you’ll always be able to use a foreach loop with them because all of them implement `IEnumerable<T>`.

You can, however, use foreach to enumerate through a stack or queue, because they implement `IEnumerable`!

A queue is like a list that lets you add objects to the end and use the ones at the beginning. A stack only lets you access the last object you put into it.

A queue is FIFO—first in, first out

A **queue** is a lot like a list, except that you can't just add or remove items at any index. To add an object to a queue, you **enqueue** it. That adds the object to the end of the queue. You can **dequeue** the first object from the front of the queue. When you do that, the object is removed from the queue, and the rest of the objects in the queue move up a position.

After the first Dequeue call, the first item in the queue is removed and returned, and the second item shifts into the first place.

```
// Create a Queue and add four strings to it
Queue<string> myQueue = new Queue<string>();
myQueue.Enqueue("first in line");
myQueue.Enqueue("second in line");
myQueue.Enqueue("third in line");
myQueue.Enqueue("last in line");
```

Here's where we call Enqueue to add four items to the queue. When we pull them out of the queue, they'll come out in the same order they went in.

```
// Peek "looks" at the first item in the queue without removing it
Console.WriteLine($"Peek() returned:\n{myQueue.Peek()}"); ①
```

```
// Dequeue pulls the next item from the FRONT of the queue
```

```
Console.WriteLine(
    $"The first Dequeue() returned:\n{myQueue.Dequeue()}"); ②
Console.WriteLine(
    $"The second Dequeue() returned:\n{myQueue.Dequeue()}"); ③
```

```
// Clear removes all of the items from the queue
```

```
Console.WriteLine($"Count before Clear():\n{myQueue.Count}"); ④
myQueue.Clear();
Console.WriteLine($"Count after Clear():\n{myQueue.Count}"); ⑤
```



Objects in a queue need to wait their turn. The first one in the queue is the first one to come out of it.

Output

- | |
|----------------------------------------------------|
| Peek() returned:
① first in line |
| The first Dequeue() returned:
② first in line |
| The second Dequeue() returned:
③ second in line |
| Count before Clear():
④ 2 |
| Count after Clear():
⑤ 0 |

A stack is LIFO—last in, first out

A **stack** is really similar to a queue—with one big difference. You **push** each item onto a stack, and when you want to take an item from the stack, you **pop** one off of it. When you pop an item off of a stack, you end up with the most recent item that you pushed onto it. It's just like a stack of plates, magazines, or anything else—you can drop something onto the top of the stack, but you need to take it off before you can get to whatever's underneath it.

```
// Create a Stack and add four strings to it
```

```
Stack<string> myStack = new Stack<string>();
```

```
myStack.Push("first in line");
myStack.Push("second in line");
myStack.Push("third in line");
myStack.Push("last in line");
```

Creating a stack is just like creating any other generic collection.

When you push an item onto a stack, it pushes the other items back one slot and sits on top.

```
// Peek with a stack works just like it does with a queue
```

```
Console.WriteLine($"Peek() returned:\n{myStack.Peek()}"); ①
```

```
// Pop pulls the next item from the BOTTOM of the stack
```

```
Console.WriteLine(
```

```
    $"The first Pop() returned:\n{myStack.Pop()}"); ②
```

```
Console.WriteLine(
```

```
    $"The second Pop() returned:\n{myStack.Pop()}"); ③
```

When you pop an item off the stack, you get the most recent item that was added.

```
Console.WriteLine($"Count before Clear():\n{myStack.Count}"); ④
```

```
myStack.Clear();
```

```
Console.WriteLine($"Count after Clear():\n{myStack.Count}"); □
```

Output

```
Peek() returned:  
① last in line  
The first Pop() returned:  
② last in line  
The second Pop() returned:  
③ third in line  
Count before Clear():  
④ 2  
Count after Clear():  
⑤ 0
```

The last object you put on a stack is the first object that you pull off of it.





WAIT A MINUTE, SOMETHING'S BUGGING ME. YOU HAVEN'T SHOWN ME ANYTHING I CAN DO WITH A STACK OR A QUEUE THAT I CAN'T DO WITH A LIST. THEY JUST SAVE ME A COUPLE OF LINES OF CODE, BUT I CAN'T GET AT THE ITEMS IN THE MIDDLE OF A STACK OR A QUEUE. I CAN DO THAT WITH A LIST PRETTY EASILY! SO WHY WOULD I GIVE THAT UP JUST FOR A LITTLE CONVENIENCE?

You don't give up anything when you use a queue or a stack.

It's really easy to copy a Queue object to a List object. It's just as easy to copy a List to a Queue, a Queue to a Stack...in fact, you can create a List, Queue, or Stack from any other object that implements the `IEnumerable<T>` interface. All you have to do is use the overloaded constructor that lets you pass the collection you want to copy from as a parameter. That means you have the flexibility and convenience of representing your data with the collection that best matches the way you need it to be used. (But remember, you're making a copy, which means you're creating a whole new object and adding it to the heap.)

Let's set up a stack
with four items—in this
case, a stack of strings.

```
Stack<string> myStack = new Stack<string>();
myStack.Push("first in line");
myStack.Push("second in line");
myStack.Push("third in line");
myStack.Push("last in line");
```

It's easy to convert that stack
to a queue, then copy the queue
to a list, and then copy the list
to another stack.

```
Queue<string> myQueue = new Queue<string>(myStack);
List<string> myList = new List<string>(myQueue);
Stack<string> anotherStack = new Stack<string>(myList);
```

```
Console.WriteLine($"\"myQueue has {myQueue.Count} items
myList has {myList.Count} items
anotherStack has {anotherStack.Count} items\"");
```

All four items were
copied into the new
collections.

Output

```
myQueue has 4 items
myList has 4 items
anotherStack has 4 items
```

...and you can always use
a foreach loop to access
all of the members in a
stack or a queue!



Exercise

Write a program to help a cafeteria full of lumberjacks eat some flapjacks. You'll use a Queue of Lumberjack objects, and each Lumberjack has a Stack of Flapjack enums. We've given you some details as a starting point. Can you create a console app that matches the output?

Start with a Lumberjack class and a Flapjack enum

The Lumberjack class has a public Name property that's set with a constructor, and a private Stack<Flapjack> field called flapjackStack that's initialized with an empty stack.

The TakeFlapjack method takes a single argument, a Flapjack, and pushes it onto the stack. EatFlapjacks pops the flapjacks off the stack and writes the lines to the console for the lumberjack.

Lumberjack
Name
private flapjackStack
TakeFlapjack
EatFlapjacks

```
enum Flapjack {
    Crispy,
    Soggy,
    Browned,
    Banana,
}
```

Then add the Main method

The Main method prompts the user for the first lumberjack's name, then asks for the number of flapjacks to give it. If the user gives a valid number, the program calls TakeFlapjack that number of times, passing it a random Flapjack each time, and adds the Lumberjack to a Queue. It keeps asking for more Lumberjacks until the user enters a blank line, then it uses a while loop to dequeue each Lumberjack and calls its EatFlapjacks method to write lines to the output.

The Main method writes these lines and takes input, creating each Lumberjack object, setting its name, having it take a number of random flapjacks, and adding it to a queue.

When the user is done entering lumberjacks, the Main method uses a while loop to dequeue each Lumberjack and call its EatFlapjacks method. The rest of the output lines are written by each Lumberjack object.

Lumberjacks write this line when they start eating flapjacks.

This lumberjack took 4 flapjacks. When her EatFlapjacks method was called, she popped 4 Flapjack enums off of her stack.

First lumberjack's name: Erik
 Number of flapjacks: 4
 Next lumberjack's name (blank to end): Hildur
 Number of flapjacks: 6
 Next lumberjack's name (blank to end): Jan
 Number of flapjacks: 3
 Next lumberjack's name (blank to end): Betty
 Number of flapjacks: 4
 Next lumberjack's name (blank to end):
 Erik is eating flapjacks
 Erik ate a soggy flapjack
 Erik ate a browned flapjack
 Erik ate a browned flapjack
 Erik ate a soggy flapjack
 Hildur is eating flapjacks
 Hildur ate a browned flapjack
 Hildur ate a browned flapjack
 Hildur ate a crispy flapjack
 Hildur ate a crispy flapjack
 Hildur ate a soggy flapjack
 Hildur ate a browned flapjack
 Jan is eating flapjacks
 Jan ate a banana flapjack
 Jan ate a crispy flapjack
 Jan ate a soggy flapjack
 Betty is eating flapjacks
 Betty ate a soggy flapjack
 Betty ate a browned flapjack
 Betty ate a browned flapjack
 Betty ate a crispy flapjack





Exercise Solution

Here's the code for the Lumberjack class and the Main method. Don't forget that each file needs a `using System.Collections.Generic;` line at the top.

```
class Lumberjack
{
    private Stack<Flapjack> flapjackStack = new Stack<Flapjack>();
    public string Name { get; private set; }

    public Lumberjack(string name)
    {
        Name = name;
    }

    public void TakeFlapjack(Flapjack flapjack)
    {
        flapjackStack.Push(flapjack);
    }

    public void EatFlapjacks()
    {
        Console.WriteLine($"{Name} is eating flapjacks");
        while (flapjackStack.Count > 0)
        {
            Console.WriteLine(
                $"{Name} ate a {flapjackStack.Pop().ToString().ToLower()} flapjack");
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Random random = new Random();
        Queue<Lumberjack> lumberjacks = new Queue<Lumberjack>();

        string name;
        Console.Write("First lumberjack's name: ");
        while ((name = Console.ReadLine()) != "")
        {
            Console.Write("Number of flapjacks: ");
            if (int.TryParse(Console.ReadLine(), out int number))
            {
                Lumberjack lumberjack = new Lumberjack(name);
                for (int i = 0; i < number; i++)
                {
                    lumberjack.TakeFlapjack((Flapjack)random.Next(0, 4));
                }
                lumberjacks.Enqueue(lumberjack);
            }
            Console.Write("Next lumberjack's name (blank to end): ");
        }

        while (lumberjacks.Count > 0)
        {
            Lumberjack next = lumberjacks.Dequeue();
            next.EatFlapjacks();
        }
    }
}
```

Here's the stack of Flapjack enums. It gets filled up when the Main method calls TakeFlapjack with random flapjacks, and drained when it calls the EatFlapjacks method.

The TakeFlapjack method just pushes a flapjack onto the stack.



The Main method keeps its Lumberjack references in a queue.

It creates each Lumberjack object, calls its TakeFlapjack method with a random flapjacks, and then enqueues the reference.

When the user is done adding flapjacks, the Main method uses a while loop to dequeue each Lumberjack reference and call its EatFlapjacks method.

there are no
Dumb Questions

Q: What happens if I try to get an object from a dictionary using a key that doesn't exist?

A: If you pass a dictionary a key that doesn't exist, it will throw an exception. For example, if you add this code to a console app:

```
Dictionary<string, string> dict =
    new Dictionary<string, string>();
string s = dict["This key doesn't exist"];
```

you'll get the exception "System.Collections.Generic.

KeyNotFoundException: 'The given key 'This key doesn't exist' was not present in the dictionary.'" Conveniently, the exception includes the key—or, more specifically, the string that the key's ToString method returns. That's really useful if you're trying to debug a problem in a program that accesses a dictionary thousands of times.

Q: Is there a way to avoid that exception—like if I don't know if a key exists?

A: Yes, there are two ways to avoid a KeyNotFoundException. One way is to use the Dictionary.ContainsKey method. You pass it the key you want to use with the dictionary, and it returns true only if the key exists. The other way is to use Dictionary.TryGetValue, which lets you do this:

```
if (dict.TryGetValue("Key", out string value))
{
    // do something
}
```

That code does exactly the same thing as this:

```
if (dict.ContainsKey("Key"))
{
    string value = dict["Key"];
    // do something
}
```

BULLET POINTS

- Lists, arrays, and other collections implement the **IEnumerable<T> interface**, which supports iteration over a non-generic collection.
- A **foreach loop** works with any class that implements **IEnumerable<T>**, which includes a method to return an Enumerator object that lets the loop iterate through its contents in order.
- **Covariance** is C#'s way of letting you implicitly convert a subclass reference to its superclass.
- The word "**implicitly**" refers to C#'s ability to figure out how to do the conversion without you needing to explicitly use casting.
- Covariance is useful when you need to pass a collection of objects to a method that only works with the class they inherit from. For example, covariance allows you to use **normal assignment to upcast** a List<Subclass> to an **IEnumerable<Superclass>**.
- A **Dictionary< TKey, TValue >** is a collection that stores a set of key/value pairs, and lets you use keys to look up their associated values.
- Dictionary keys and values can be **different types**. Every **key must be unique** in the dictionary, but values can be duplicated.
- The Dictionary class has a **Keys** property that returns an iterable sequence of keys.
- A **Queue<T>** is a first-in, first-out collection with methods to enqueue an item at the end of the queue, and dequeue the item at the front of the queue.
- A **Stack<T>** is a last-in, first-out collection with methods to push an item onto the top of the stack and pop an item off of the top of the stack.
- The Stack<T> and Queue<T> classes **implement IEnumerable<T>**, and can easily be converted to Lists or other collection types.

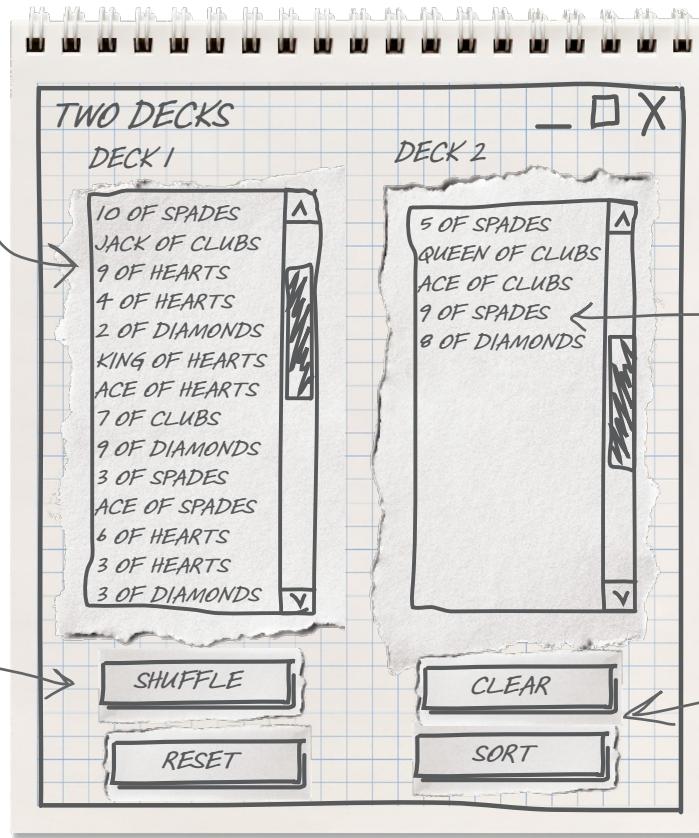


Downloadable exercise: Two Decks

In the next exercise you'll build an app that lets you move cards between two decks. The left deck has buttons that let you shuffle it and reset it to 52 cards, and the right deck has buttons that let you clear and sort it.

When you start the app, the left box contains a complete deck of cards. The right box is empty.

The Shuffle button shuffles the cards in Deck 1, and the Reset button resets it to a sorted 52-card deck.



Double-clicking on a card in one deck transfers it to the other. So clicking on 9 of Spades will remove it from Deck 2 and add it to Deck 1.

The Clear button removes all cards from Deck 2, and the Sort button sorts the cards in it so they're in order.

One of the most important ideas we've emphasized throughout this book is that writing C# code is a skill, and the best way to improve that skill is to **get lots of practice**. We want to give you as many opportunities to get practice as possible!

That's why we created **additional Windows WPF and macOS ASP.NET Core Blazor projects** to go with some of the chapters in the rest of this book. We've included these projects at the end of the next few chapters too. Go ahead and download the PDF for this project—we think you should take the time and do it before moving on to the next chapter, because it will help reinforce some important concepts that will help you with the material in the rest of the book.

Go to the GitHub repo for the book and download the project PDF now:

<https://github.com/head-first-csharp/fourth-edition>

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 your game's 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.

Your program so far is an interesting visual simulation. The goal of this Unity Lab is to **finish building the game**. It starts off with a score of zero. Billiard balls will start to appear and fly around the screen. When the player clicks on a ball, the score goes up by 1 and the ball disappears. More and more balls appear; once 15 balls are flying around the screen, the game ends. For your game to work, your players need a way to start it and to play again once the game is over, and they'll want to see their score as they click on the balls. So you'll add a **user interface** that displays the score in the corner of the screen, and shows a button to start a new game,

Add a score that goes up when the player clicks a ball

You've got a really interesting simulation. Now it's time to turn it into a game. **Add a new field** to the GameController class to keep track of the score—you can add it just below the OneBallPrefab field:

```
public int Score = 0;
```

Next, **add a method called ClickedOnBall to the GameController class**. This method will get called every time the player clicks on a ball:

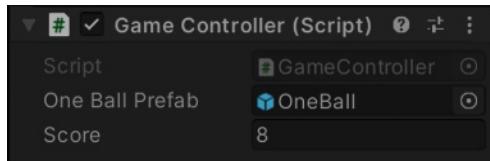
```
public void ClickedOnBall()
{
    Score++;
}
```

Unity makes it really easy for your GameObjects to respond to mouse clicks and other input. If you add a method called OnMouseDown to a script, Unity will call that method any time the GameObject it's attached to is clicked. **Add this method to the OneBallBehaviour class**:

```
void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    controller.ClickedOnBall();
    Destroy(gameObject);
}
```

The first line of the OnMouseDown method gets the instance of the GameController class, and the second line calls its ClickedOnBall method, which increments its Score field.

Now run your game. Click on Main Camera in the hierarchy and watch its Game Controller (Script) component in the Inspector. Click on some of the rotating balls—they'll disappear and the Score will go up.



there are no
Dumb Questions

Q: Why do we use Instantiate instead of the new keyword?

A: Instantiate and Destroy are **special methods that are unique to Unity**—you won't see them in your other C# projects. The Instantiate method isn't quite the same thing as the C# new keyword, because it's creating a new instance of a prefab, not a class. Unity does create new instances of objects, but it needs to do a lot of other things, like making sure that it's included in the update loop. When a GameObject's script calls Destroy(gameObject) it's telling Unity to destroy itself. The Destroy method tells Unity to destroy a GameObject—but not until after the update loop is complete.

Q: I'm not clear on how the first line of the OnMouseDown method works. What's going on there?

A: Let's break down that statement piece by piece. The first part should be pretty familiar: it declares a variable called controller of type GameController, the class that you defined in the script that you attached to the Main Camera. In the second half, we want to call a method on the GameController attached to the Main Camera. So we use Camera.main to get the Main Camera, and GetComponent<GameController>() to get the instance of GameController that we attached to it.

Unity Lab #4

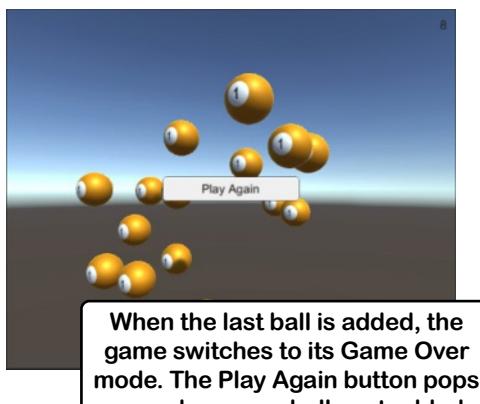
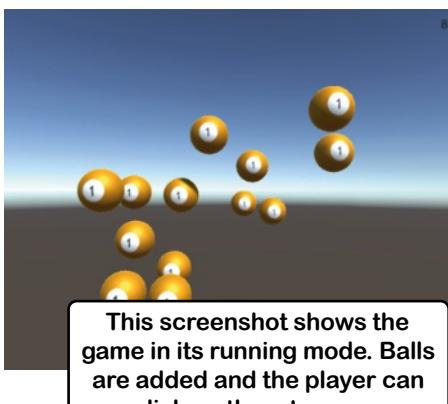
User Interfaces

Add two different modes to your game

Start up your favorite game. Do you immediately get dropped into the action? Probably not—you’re probably looking at a start menu. Some games let you pause the action to look at a map. Many games let you switch between moving the player and working with an inventory, or show an animation while the player is dying that can’t be interrupted. These are all examples of **game modes**.

Let’s add two different modes to your billiard ball game:

- ★ **Mode #1: The game is running.** Balls are being added to the scene, and clicking on them makes them disappear and the score go up.
- ★ **Mode #2: The game is over.** Balls are no longer getting added to the scene, clicking on them doesn’t do anything, and a “Game over” banner is displayed.



You’ll add two modes to your game. You already have the “running” mode, so now you just need to add a “game over” mode.

Here’s how you’ll add the two game modes to your game:

① Make `GameController.AddABall` pay attention to the game mode.

Your new and improved `AddABall` method will check if the game is over, and will only instantiate a new `OneBall` prefab if the game is not over.

② Make `OneBallBehaviour.OnMouseDown` only work when the game is running.

When the game is over, we want the game to stop responding to mouse clicks. The player should just see the balls that were already added continue to circle until the game restarts.

③ Make `GameController.AddABall` end the game when there are too many balls.

`AddABall` also increments its `NumberOfBalls` counter, so it goes up by 1 every time a ball is added. If the value reaches `MaximumBalls`, it sets `GameOver` to true to end the game.

In this lab, you’re building this game in parts, and making changes along the way. You can download the code for each part from the book’s GitHub repository: <https://github.com/head-first-csharp/fourth-edition>.

Add game mode to your game

Modify your GameController and OneBallBehaviour classes to **add modes to your game** by using a Boolean field to keep track of whether or not the game is over.

1 Make GameController.AddABall pay attention to the game mode.

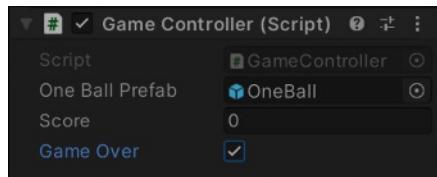
We want the GameController to know what mode the game is in. When we need to keep track of what an object knows, we use fields. Since there are two modes—running and game over—we can use a Boolean field to keep track of the mode. **Add the GameOver field** to your GameController class:

```
public bool GameOver = false;
```

The game should only add new balls to the scene if the game is running. Modify the AddABall method to add an **if** statement that only calls Instantiate if GameOver is not true:

```
public void AddABall()
{
    if (!GameOver)
    {
        Instantiate(OneBallPrefab);
    }
}
```

Now you can test it out. Start your game, then **click on Main Camera** in the Hierarchy window.



Check the Game Over box while the game is running to toggle the GameController's GameOver field. If you check it while the game is running, Unity will reset it when you stop the game.

Set the GameOver field by unchecking the box in the Script component. The game should stop adding balls until you check the box again.

2 Make OneBallBehaviour.OnMouseDown only work when the game is running.

Your OnMouseDown method already calls the GameController's ClickedOnBall method. Now **modify OnMouseDown in OneBallBehaviour** to use the GameController's GameOver field as well:

```
void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    if (!controller.GameOver)
    {
        controller.ClickedOnBall();
        Destroy(gameObject);
    }
}
```

Run your game again and test that balls disappear and the score goes up only when the game is not over.

Unity Lab #4

User Interfaces

3 Make `GameController.AddABall` end the game when there are too many balls.

The game needs to keep track of the number of balls in the scene. We'll do this by **adding two fields** to the `GameController` class to keep track of the current number of balls and the maximum number of balls:

```
public int NumberOfBalls = 0;  
public int MaximumBalls = 15;
```

Every time the player clicks on a ball, the ball's `OneBallBehaviour` script calls `GameController.ClickedOnBall` to increment (add 1 to) the score. Let's also decrement (subtract 1 from) `NumberOfBalls`:

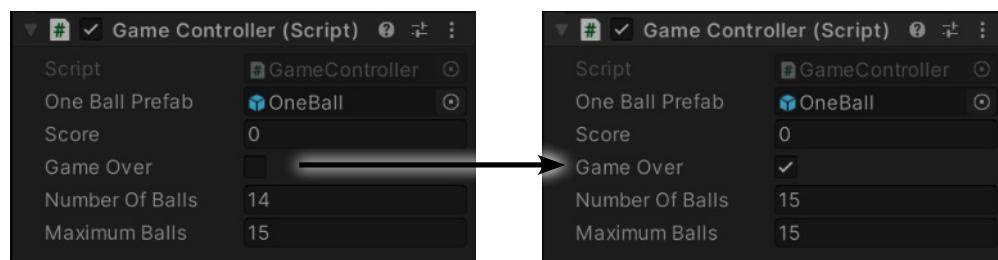
```
public void ClickedOnBall()  
{  
    Score++;  
    NumberOfBalls--;  
}
```

Now **modify the `AddABall` method** so that it only adds balls if the game is running, and ends the game if there are too many balls in the scene:

```
public void AddABall()  
{  
    if (!GameOver)  
    {  
        Instantiate(OneBallPrefab);  
        NumberOfBalls++;  
        if (NumberOfBalls >= MaximumBalls)  
        {  
            GameOver = true;  
        }  
    }  
}
```

The `GameOver` field is true if the game is over and false if the game is running. The `NumberOfBalls` field keeps track of the number of balls currently in the scene. Once it hits the `MaximumBalls` value, the `GameController` will set `GameOver` to true.

Now test your game one more time by running it and then clicking on Main Camera in the Hierarchy window. The game should run normally, but as soon as the `NumberOfBalls` field is equal to the `MaximumBalls` field, the `AddABall` method sets its `GameOver` field to true and ends the game.



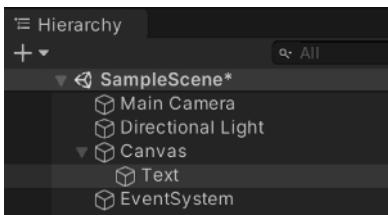
Once that happens, clicking on the balls doesn't do anything because `OneBallBehaviour.OnMouseDown` checks the `GameOver` field and only increments the score and destroys the ball if `GameOver` is false.

Your game needs to keep track of its game mode. Fields are a great way to do that.

Add a UI to your game

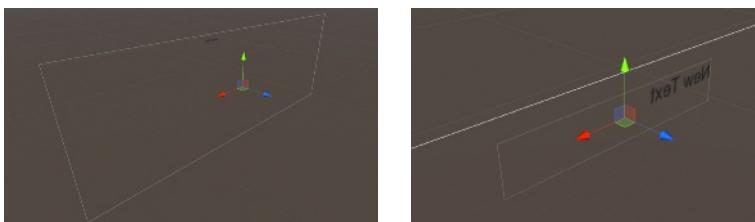
Almost any game you can think of—from Pac Man to Super Mario Brothers to Grand Theft Auto 5 to Minecraft—features a **user interface (or UI)**. Some games, like Pac Man, have a very simple UI that just shows the score, high score, lives left, and current level. Many games feature an intricate UI incorporated into the game’s mechanics (like a weapon wheel that lets the player quickly switch between weapons). Let’s add a UI to your game.

Choose `UI >> Text` from the `GameObject` menu to add a 2D Text GameObject to your game’s UI. This adds a Canvas to the Hierarchy, and a Text under that Canvas:



When you added the Text to your scene, Unity automatically added Canvas and Text GameObjects. Click the triangle (▲) next to Canvas to expand or collapse it—the Text GameObject will appear and disappear because it’s nested under Canvas.

Double-click on Canvas in the Hierarchy window to focus on it. It’s a 2-D rectangle. Click on its Move Gizmo and drag it around the scene. It won’t move! The Canvas that was just added will always be displayed, scaled to the size of the screen and in front of everything else in the game.

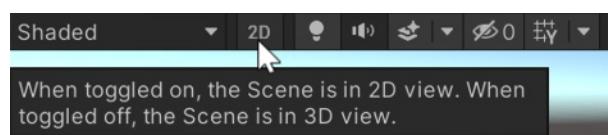


Did you notice an EventSystem in the Hierarchy? Unity automatically added it when you created the UI. It manages mouse, keyboard, and other inputs and sends them back to GameObjects—and it does all of that automatically, so you won’t need to work directly with it.

Then double-click on Text to focus on it—the editor will zoom in, but the default text (“New Text”) will be backward because the Main Camera is pointing at the back of the Canvas.

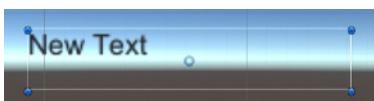
Use the 2D view to work with the Canvas

The **2D button** at the top of the Scene window toggles 2D view on and off:



A Canvas is a two-dimensional GameObject that lets you lay out your game’s UI. Your game’s Canvas will have two GameObjects nested under it: the Text GameObject that you just added will be in the upper-right corner to display the score, and there’s a Button GameObject to let the player start a new game.

Click the 2D view—the editor flips around its view to shows the canvas head-on. **Double-click on Text** in the Hierarchy window to zoom in on it.



Use the mouse wheel to zoom in and out in 2D view.

You can **click the 2D button to switch between 2D and 3D**. Click it again to return to the 3D view.

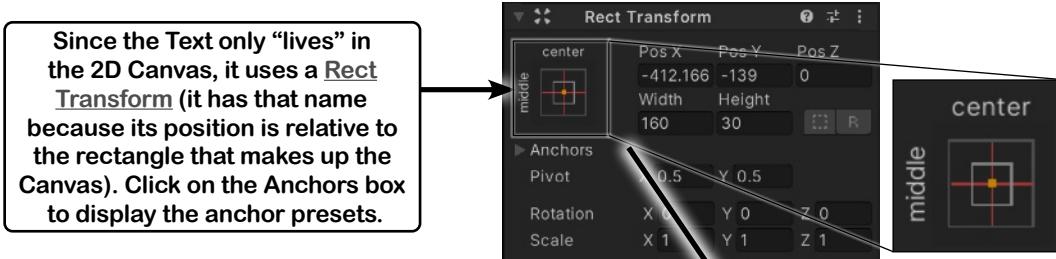
Unity Lab #4

User Interfaces

Set up the Text that will display the score in the UI

Your game's UI will feature one Text GameObject and one Button. Each of those GameObjects will be **anchored** to a different part of the UI. For example, the Text GameObject that displays the score will show up in the upper-right corner of the screen (no matter how big or small the screen is).

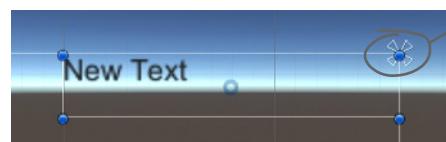
Click on Text in the Hierarchy window to select it, then look at the Rect Transform component. We want the Text in the upper-right corner, so **click the Anchors box** in the Rect Transform panel.



The Text is anchored to a specific point in the 2D Canvas.

Make sure you hold down both Shift and Alt (Option on a Mac) so you set both the pivot and position.

The Anchor Presets window lets you anchor your UI GameObjects to various parts of the Canvas. **Hold down Alt and Shift** (or Option+Shift on a Mac) and **choose the top right anchor preset**. Click the same button you used to bring up the Anchor Presets window. The Text is now in the upper-right corner of the Canvas—double-click on it again to zoom into it.



You set the anchor pivot to the top right. The Text's position is the anchor's position relative to the Canvas.



Let's add a little space above and to the right of the Text. Go back to the Rect Transform panel and **set both Pos X and Pos Y to -10** to position the text 10 units to the left and 10 down from the top-right corner. Then **set the Alignment on the Text component to right**, and use the box at the top of the Inspector to **change the GameObject's name to Score**.



Your new Text should now show up in the Hierarchy window with the name Score. It should now be right-aligned, with a small gap between the edge of the Text and the edge of the Canvas.

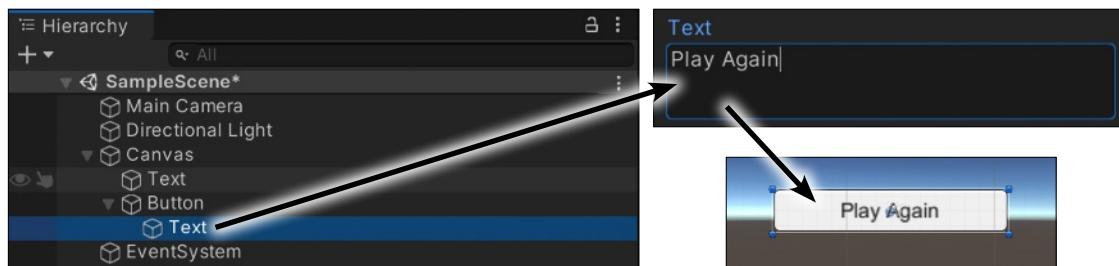


Add a button that calls a method to start the game

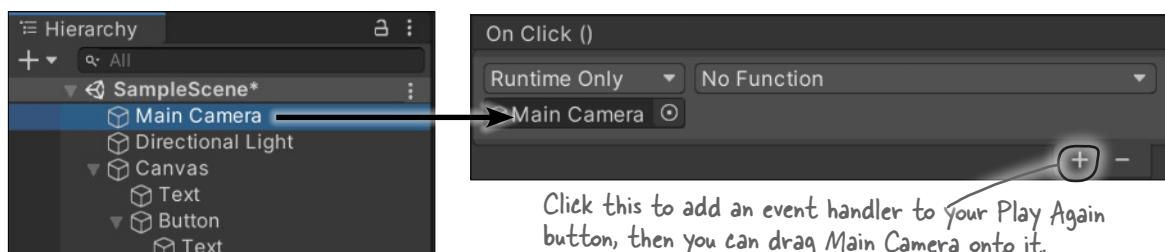
When the game is in its “game over” mode, it will display a button labeled Play Again that calls a method to restart the game. **Add an empty StartGame method** to your GameController class (we’ll add its code later):

```
public void StartGame()
{
    // We'll add the code for this method later
}
```

Click on Canvas in the Hierarchy window to focus on it. Then **choose UI > Button** from the GameObject menu to add a Button. Since you’re already focused on the Canvas, the Unity editor will add the new Button and anchor it to the center of the Canvas. Did you notice that Button has a triangle next to it in the Hierarchy? Expand it—there’s a TextGameObject nested under it. Click on it and set its text to **Play Again**.



Now that the Button is set up, we just need to make it call the StartGame method on the GameController object attached to the Main Camera. A UI button is **just a GameObject with a Button component**, and you can use its On Click () box in the Inspector to hook it up to an event handler method. Click the **+** button at the bottom of the On Click () box to add an event handler, then **drag Main Camera onto the None (Object) box**.



Now the Button knows which GameObject to use for the event handler. Click the **No Function** dropdown and choose **GameController >> StartGame**. Now when the player presses the button, it will call the StartGame method on the GameController object hooked up to the Main Camera.



Make the Play Again button and Score Text work

Your game's UI will work like this:

- ★ The game starts in the game over mode.
- ★ Clicking the Play Again button starts the game.
- ★ Text in the upper-right corner of the screen displays the current score.

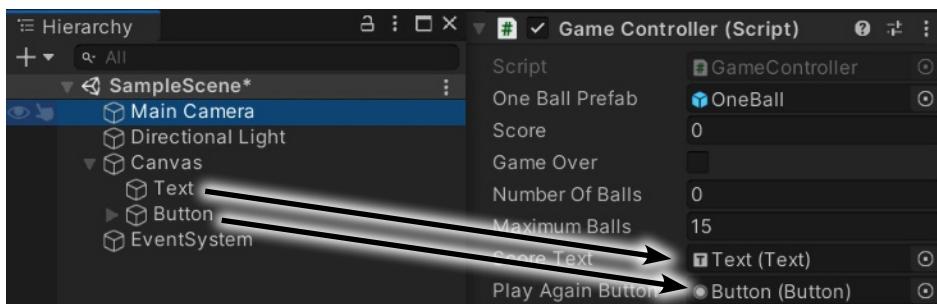
You'll be using the Text and Button classes in your code. They're in the UnityEngine.UI namespace, so **add this using statement** to the top of your GameController class:

```
using UnityEngine.UI;
```

Now you can add Text and Button fields to your GameController (just above the OneBallPrefab field):

```
public Text ScoreText;  
public Button PlayAgainButton;
```

Click on Main Camera in the Hierarchy window. **Drag the Text GameObject** out of the Hierarchy and **onto** the Score Text field in the Script component, **then drag the Button GameObject onto** the Play Again Button field.



Go back to your GameController code and **set the GameController field's default value to true**:

```
public bool GameOver = true; ← Change this from false to true.
```

Now go back to Unity and check the Script component in the Inspector.

Hold on, something's wrong!



The Unity editor still shows the Game Over checkbox as unchecked—it didn't change the field value. Make sure to check the box so your game starts in the game over mode:



Now the game will start in the game over mode, and the player can click the Play Again button to start playing.



Watch it!

Unity remembers your scripts' field values.

When you wanted to change the `GameController.GameOver` field from false to true, it wasn't enough to change the code. When you add a Script component to Unity, it keeps track of the field values, and it won't reload the default values unless you reset it from the context menu (R).

Finish the code for the game

The GameController object attached to the Main Camera keeps track of the score in its Score field. **Add an Update method to the GameController class** to update the Score Text in the UI:

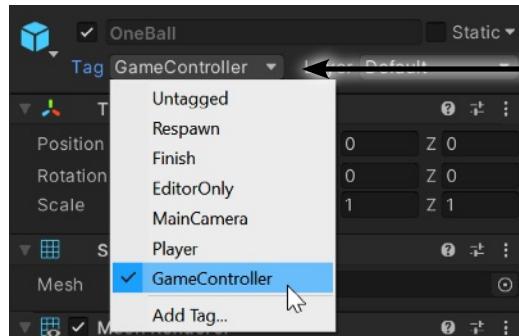
```
void Update()
{
    ScoreText.text = Score.ToString();
}
```

Next, **modify your GameController.AddABall method** to enable the Play Again button when it ends the game:

```
if (NumberOfBalls >= MaximumBalls)
{
    GameOver = true;
    PlayAgainButton.gameObject.SetActive(true);
}
```

Every GameObject has a property called `gameObject` that lets you manipulate it. You'll use its `SetActive` method to make the Play Again button visible or invisible.

There's just one more thing to do: get your StartGame method working so that it starts the game. It needs to do a few things: destroy any balls that are currently flying around the scene, disable the Play Again button, reset the score and number of balls, and set the mode to "running." You already know how to do most of those things! You just need to be able to find the balls in order to destroy them. **Click on the OneBall prefab in the Project window and set its tag:**



A tag is a keyword that you can attach to any of your GameObjects that you can use in your code when you need to identify them or find them. When you click on a prefab in the Project window and use this dropdown to assign a tag, that tag will be assigned to every instance of that prefab that you instantiate.

Now you have everything in place to fill in your StartGame method. It uses a `foreach` loop to find and destroy any balls left over from the previous game, hides the button, resets the score and number of balls, and changes the game mode:

```
public void StartGame()
{
    foreach (GameObject ball in GameObject.FindGameObjectsWithTag("GameController"))
    {
        Destroy(ball);
    }
    PlayAgainButton.gameObject.SetActive(false);
    Score = 0;
    NumberOfBalls = 0;
    GameOver = false;
}
```

Now run your game. It starts in "game over" mode. Press the button to start the game. The score goes up each time you click on a ball. As soon as the 15th ball is instantiated, the game ends and the Play Again button appears again.

Unity Lab #4

User Interfaces



Exercise

Here's a Unity **coding challenge** for you! Each of your GameObjects has a `transform.Translate` method that moves it a distance from its current position. The goal of this exercise is to modify your game so that instead of using `transform.RotateAround` to circle balls around the Y axis, your `OneBallBehaviour` script uses `transform.Translate` to make the balls fly randomly around the scene.

- Remove the XRotation, YRotation, and ZRotation fields from `OneBallBehaviour`. Replace them with fields to hold the X, Y, and Z speed called `XSpeed`, `YSpeed`, and `ZSpeed`. They're float fields—no need to set their values.

- Replace all of the code in the **Update** method with this line of code that calls the `transform.Translate` method:

```
transform.Translate(Time.deltaTime * XSpeed,  
                  Time.deltaTime * YSpeed, Time.deltaTime * ZSpeed);
```

The parameters represent the speed that the ball is traveling along the X, Y, or Z axis. So if `XSpeed` is 1.75, multiplying it by `Time.deltaTime` causes ball move along the X axis at a rate of 1.75 units per second.

- Replace the **DegreesPerSecond** field with a field called `Multiplier` with a value of 0.75F—the F is important! Use it to update the `XSpeed` field in the `Update` method, and add two similar lines for the `YSpeed` and `ZSpeed` fields:

```
XSpeed += Multiplier - Random.value * Multiplier * 2;
```

Part of this exercise is to understand exactly how this line of code works. `Random.value` is a static method that returns a random floating-point number between 0 and 1. What is this line of code doing to the `XSpeed` field?

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

- Then add a method called `ResetBall` and call it from the `Start` method. Add this line of code to `ResetBall`:

```
XSpeed = Multiplier - Random.value * Multiplier * 2;
```

What does that line of code do?

Before you start working on the game,
figure out what these lines of code do.
↑
↓

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

Add two more lines just like it to `ResetBall` that update `YSpeed` and `ZSpeed`. Then move the line of code that updates `transform.position` out of the `Start` method and into the `ResetBall` method.

- Modify the `OneBallBehaviour` class to add a field called `TooFar` and set it to 5. Then modify the `Update` method to check whether the ball went too far. You can check if a ball went too far along the X axis like this:

```
Mathf.Abs(transform.position.x) > TooFar
```

That checks the *absolute value* of the X position, which means that it will check if `transform.position.x` is greater than 5F or less than -5F. Here's an `if` statement that checks if the ball went too far along the X, Y, or Z axis:

```
if ((Mathf.Abs(transform.position.x) > TooFar)  
    || (Mathf.Abs(transform.position.y) > TooFar)  
    || (Mathf.Abs(transform.position.z) > TooFar)) {
```

Modify your `OneBallBehaviour.Update` method to use that `if` statement to call `ResetBall` if the ball went too far.



Exercise Solution

Here's what the entire OneBallBehaviour class looks like after updating it following the instructions in the exercise. The key to how this game works is that each ball's speed along the X, Y, and Z axes is determined by its current XSpeed, YSpeed, and ZSpeed values. By making small changes to those values, you've made your ball move randomly throughout the scene.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class OneBallBehaviour : MonoBehaviour
{
    public float XSpeed;
    public float YSpeed;
    public float ZSpeed;
    public float Multiplier = 0.75F;
    public float TooFar = 5;

    static int BallCount = 0;
    public int BallNumber;

    // Start is called before the first frame update
    void Start()
    {
        BallCount++;
        BallNumber = BallCount;

        ResetBall();
    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate(Time.deltaTime * XSpeed,
                           Time.deltaTime * YSpeed, Time.deltaTime * ZSpeed);

        XSpeed += Multiplier - Random.value * Multiplier * 2;
        YSpeed += Multiplier - Random.value * Multiplier * 2;
        ZSpeed += Multiplier - Random.value * Multiplier * 2;

        if ((Mathf.Abs(transform.position.x) > TooFar)
            || (Mathf.Abs(transform.position.y) > TooFar)
            || (Mathf.Abs(transform.position.z) > TooFar))
        {
            ResetBall();
        }
    }
}

```

You added these fields to your OneBallBehaviour class. Don't forget to add the F to 0.75F, otherwise your code won't build.

When the ball is first instantiated, its Start method calls ResetBall to give it a random position and speed.

The Update method first moves the ball, then updates the speed, and finally checks if it went out of bounds. It's OK if you did these things in a different order.

Unity Lab #4

User Interfaces



Exercise Solution

```
void ResetBall()
{
    XSpeed = Random.value * Multiplier;
    YSpeed = Random.value * Multiplier;
    ZSpeed = Random.value * Multiplier;

    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);
}

void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    if (!controller.GameOver)
    {
        controller.ClickedOnBall();
        Destroy(gameObject);
    }
}
```

We reset the ball when it's first instantiated or if it flies out of bounds by giving it a random speed and position. It's OK if you set the position first.

Here are our answers to the questions—did you come up with similar answers?

XSpeed += Multiplier - Random.value * Multiplier * 2;

What is this line of code doing to the XSpeed field?

Random.value * Multiplier * 2 finds a random number between 0 and 1.5. Subtracting that from Multiplier gives us a random number between -0.75 and 0.75. Adding that value to XSpeed causes it to either speed up or slow down a small amount for each frame.

By increasing or decreasing the ball's speed along all three axes, we're giving each ball a wobbly random path.

XSpeed = Multiplier - Random.value * Multiplier * 2;

What does that line of code do?

It sets the XSpeed field to a random value between -0.75 and 0.75. This causes some balls to start going forward along the X axis and others to go backward, all at different speeds.

Did you notice that you didn't have to make any changes to the GameController class? That's because you didn't make changes to the things that GameController does, like managing the UI or the game mode. If you can make a change by modifying one class but not touching others, that can be a sign that you designed your classes well.

Get creative!

Can you find ways to improve your game and get practice writing code? Here are some ideas:

- ★ Is the game too easy? Too hard? Try changing the parameters that you pass to InvokeRepeating in your GameController.Start method. Try making them fields. Play around with the MaximumBalls value, too. Small changes in these values can make a big difference in gameplay.
- ★ We gave you texture maps for all of the billiard balls. Try adding different balls that have different behaviors. Use the scale to make some balls bigger or smaller, and change their parameters to make them go faster or slower, or move differently.
- ★ Can you figure out how to make a “shooting star” ball that flies off really quickly in one direction and is worth a lot if the player clicks on it? How about making a “sudden death” 8 ball that immediately ends the game?
- ★ Modify your GameController.ClickedOnBall method to take a score parameter instead of incrementing the Score field and add the value that you pass. Try giving different values to different balls.

If you change fields in the OneBallBehaviour script, don't forget to reset the Script component of the OneBall prefab! Otherwise, it will remember the old values.

The more practice you get writing C# code, the easier it will get. Getting creative with your game is a great opportunity to get some practice!

BULLET POINTS

- Unity games display a **user interface (UI)** with controls and graphics on a flat, two-dimensional plane in front of the game's 3D scene.
- Unity provides a set of **2D UI GameObjects** specifically made for building user interfaces.
- A **Canvas** is a 2D GameObject that lets you lay out your game's UI. UI components like Text and Button are nested under a Canvas GameObject.
- The **2D button** at the top of the Scene window toggles 2D view on and off, which makes it easier to lay out a UI.
- When you add a **Script component** to Unity, it keeps track of the field values, and it won't reload the default values unless 2D reset it from the context menu.
- A **Button** can call any method in a script that's attached to a GameObject.
- You can use the Inspector to **modify field values** in your GameObjects' scripts. If you modify them while the game is running, they'll reset to saved values when it stops.
- The **transform.Translate** method moves a GameObject a distance from its current position.
- A **tag** is a keyword that you can attach to any of your GameObjects that you can use in your code when you need to identify them or find them.
- The **GameObject.FindGameObjectsWithTag** method returns a collection of GameObjects that match a given tag.

9 LINQ and lambdas

Get control of your data

SO IF YOU TAKE THE FIRST FIVE WORDS FROM THIS ARTICLE, AND THE LAST FIVE IN THAT ONE, AND ADD THEM TO THE REVERSE OF THE HEADLINE OVER HERE...YOU GET SECRET MESSAGES FROM THE ALIEN MOTHERSHIP!



It's a data-driven world...we all need to know how to live in it.

Gone are the days when you could program for days, even weeks, without dealing with **loads of data**. Today, **everything is about data**, and that's where **LINQ** comes in. LINQ is a feature of C# and .NET that not only lets you **query data** in your .NET collections in an intuitive way, but lets you **group data** and **merge data from different data sources**. You'll add **unit tests** to make sure your code is working the way you want. Once you've got the hang of wrangling your data into manageable chunks, you can use **lambda expressions** to refactor your C# code to make it even more expressive.

Jimmy's a Captain Amazing super-fan...

Meet Jimmy, one of the most enthusiastic collectors of Captain Amazing comics, graphic novels, and paraphernalia. He knows all the Captain trivia, he's got props from all the movies, and he's got a comic collection that can only be described as, well, amazing.



...but his collection's all over the place

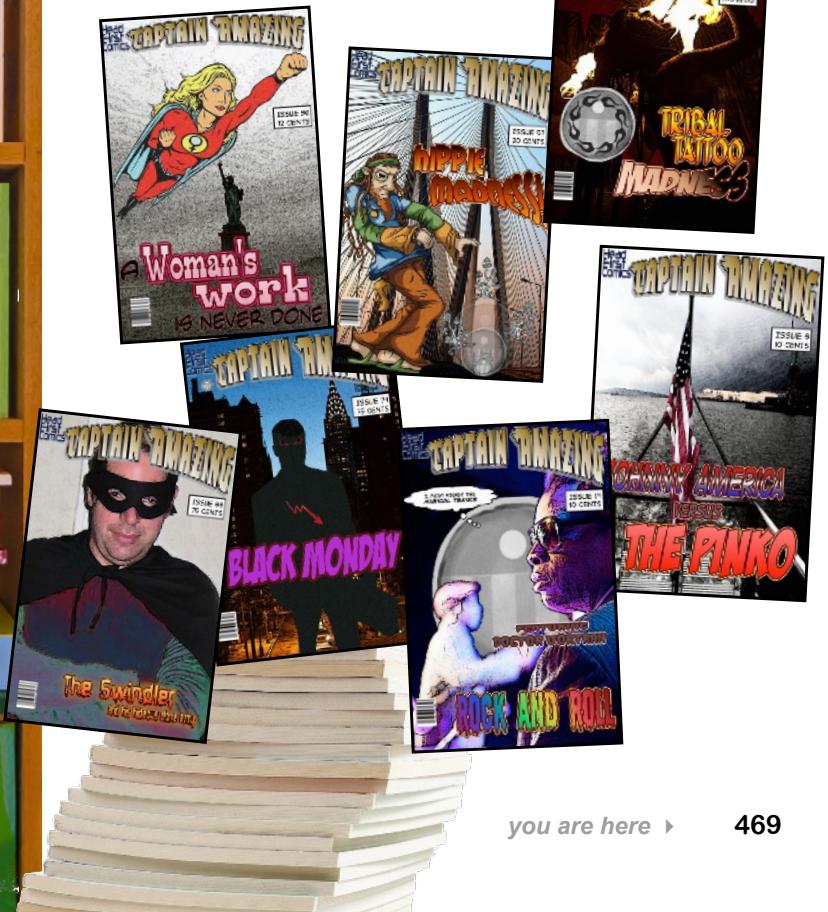
Jimmy may be passionate, but he's not exactly organized. He's trying to keep track of the most prized "crown jewel" comics of his collection, but he needs help. Can you build Jimmy an app to manage his comics?



LINQ works with values *and* objects.

We're going to use LINQ to query collections of numbers to introduce the ideas and syntax. You'll probably be thinking, "But how does this help manage comic books?" That's a great question to keep in mind in the first part of this chapter.

Later on, we'll use really similar LINQ queries to manage collections of Comic objects.



Framed cover of the legendary "Death of the Object" issue, signed by the writers.



Use LINQ to query your collections

In this chapter you'll learn about **LINQ** (or **Language-Integrated Query**). LINQ combines really useful classes and methods with powerful features built directly into C#, all created to help you work with sequences of data—like Jimmy's comic book collection.

Let's use Visual Studio to start exploring LINQ. **Create a new Console App (.NET Core)** project and give it the name **LinqTest**. Add this code, and when you get to the last line **add a period** and look at the IntelliSense window:

```
using System;

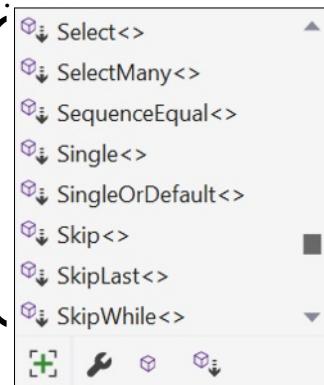
namespace LinqTest
{
    using System.Collections.Generic;
    using System.Linq; ←

    class Program
    {
        static void Main(string[] args)
        {
            List<int> numbers = new List<int>();
            for (int i = 1; i <= 99; i++)
                numbers.Add(i);
            IEnumerable<int> firstAndLastFive = numbers.
        }
    }
}
```

You've been using arrays and lists for a while now, but you haven't seen any of these methods before. Try removing `using System.Linq;` from the top of the class, then look at the IntelliSense again—all of those new methods disappeared! They only show up when you include the `using` directive.

Adding `using System.Linq;` gives your collections a bunch of new methods that let you run all sorts of queries on them.

Type "numbers" and then press . (period) to bring up the IntelliSense window. Hey, there are a lot more methods in there than there used to be!



Let's use some of those new methods to finish your console app:

```
IEnumerable<int> firstAndLastFive = numbers.Take(5).Concat(numbers.TakeLast(5));
foreach (int i in firstAndLastFive)
{
    Console.Write($"{i} ");
}
}
```

Now run your app. It prints this line of text to the console:

1 2 3 4 5 95 96 97 98 99

So what did you just do?

LINQ (or Language INtegrated Query) is a combination of C# features and .NET classes that helps you work with sequences of data.

A LINQ Query Up Close



Let's take a closer look at how you're using the LINQ methods Take, TakeLast, and Concat.

numbers

This is the original List<int> that you created with a for loop.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27			
28	29	30	31	32	33	34	35	36	37	38	39			
40	41	42	43	44	45	46	47	48	49	50	51			
52	53	54	55	56	57	58	59	60	61	62	63			
64	65	66	67	68	69	70	71	72	73	74	75			
76	77	78	79	80	81	82	83	84	85	86	87			
88	89	90	91	92	93	94	95	96	97	98	99			

numbers.Take(5)

The Take method takes the first elements from a sequence.

1	2	3	4	5
---	---	---	---	---

numbers.TakeLast(5)

The TakeLast method takes the last elements from a sequence.

96	97	98	99	100
----	----	----	----	-----

numbers.Take(5).Concat(numbers.TakeLast(5))

The Concat method concatenates two sequences together.

1	2	3	4	5	96	97	98	99	100
---	---	---	---	---	----	----	----	----	-----

Use method chaining with LINQ methods

When you add using System.Linq; to your code, the LINQ methods get added to lists. They also get added to arrays, queues, stacks...in fact, they're added to any object that extends `IEnumerable<T>`. Since almost all of the LINQ methods return an `IEnumerable<T>`, you can take the results of a LINQ method and call another LINQ method directly on them without using a variable to keep track of the results. That's called method chaining, and it lets you write very compact code.

For example, we could have used variables to store the results of Take and TakeLast:

```
IEnumerator<int> firstFive = numbers.Take(5);
IEnumerator<int> lastFive = numbers.TakeLast(5);
IEnumerator<int> firstAndLastFive = firstFive.Concat(lastFive);
```

But method chaining lets us put it all into a single line of code, calling Concat directly on the output of numbers.Take(5). They both do the same thing. Keep in mind that compact code is not necessarily better than verbose code! Sometimes breaking a chained method call into extra lines of code makes it clearer and easier to understand. It's up to you to decide which is more readable for any particular project.

LINQ works with any `IEnumerable<T>`

When you added the `using System.Linq;` directive to your code, your `List<int>` suddenly got “superpowered”—a bunch of LINQ methods repeated appeared on it. You can do the same thing for **any class that implements `IEnumerable<T>`**.

When a class implements `IEnumerable<T>`, any instance of that class is a **sequence**:

- ★ The list of numbers from 1 to 99 was a sequence.
- ★ When you called its `Take` method, it returned a reference to a sequence that contained the first five elements.
- ★ When you called its `TakeLast` method, it returned another five-element sequence.
- ★ And when you used `Concat` to combine the two five-element sequences, it created a new sequence with 10 elements and returned a reference to that new sequence.

LINQ methods enumerate your sequences

You already know that `foreach` loops work with `IEnumerable` objects. Think about what a `foreach` loop does:

```
foreach (int i in firstAndLastFive)
{
    Console.WriteLine($"{i} ");
}
```

This `foreach` loop operates on the sequence 1, 2, 3, 4, 5, 96, 97, 98, 99, 100.

It starts at the first element of the sequence (in this case, 1)...

...and it does an operation on each element in that sequence in order (writing a string to the console).

When a method goes through each item in a sequence in order, that's called **enumerating** the sequence. And that's how LINQ methods work.

Objects that implement the `IEnumerable` interface can be enumerated. That's the job objects that implement the `IEnumerable` interface do.

Any time you have an object that implements the `IEnumerable` interface, you have a sequence that you can use with LINQ. Doing an operation on that sequence in order is called enumerating the sequence.

e-nu-mer-ate, verb.
mention a number of things one by one. Suzy **enumerated** the toy cars in her collection for her dad, telling him each car's make and model.

`Enumerable.Range(8, 5);` → 8 9 10 11 12

What if you want to find the first 30 issues in Jimmy's collection starting with issue #118? LINQ provides a really useful method to help with that. The static `Enumerable.Range` method generates a sequence of integers. Calling `Enumerable.Range(8, 5)` returns a sequence of 5 numbers starting with 8: 8, 9, 10, 11, 12.

ⓘ `IEnumerable<int> Enumerable.Range(int start, int count)`
Generates a sequence of integral numbers within a specified range.

Exceptions:

`ArgumentOutOfRangeException`

↗ You'll get some practice with the `Enumerable.Range` method in the next pencil-and-paper exercise.

Sharpen your pencil



Here are just a few of the LINQ methods that you'll find on your sequences when you add the `using System.Linq;` directive to your code. They have pretty intuitive names—can you figure out just from their names what they do? **Draw a line** connecting each method call to its output.

`Enumerable.Range(1, 5)
.Sum()`

9

`Enumerable.Range(1, 6)
.Average()`

17

`new int[] { 3, 7, 9, 1, 10, 2, -3 }
.Min()`

104

`new int[] { 8, 6, 7, 5, 3, 0, 9 }
.Max()`

15

`Enumerable.Range(10, 3721)
.Count()`

3.5

`Enumerable.Range(5, 100)
.Last()`

10

`new List<int>() { 3, 8, 7, 6, 9, 6, 2 }
.Skip(4)
.Sum()`

-3

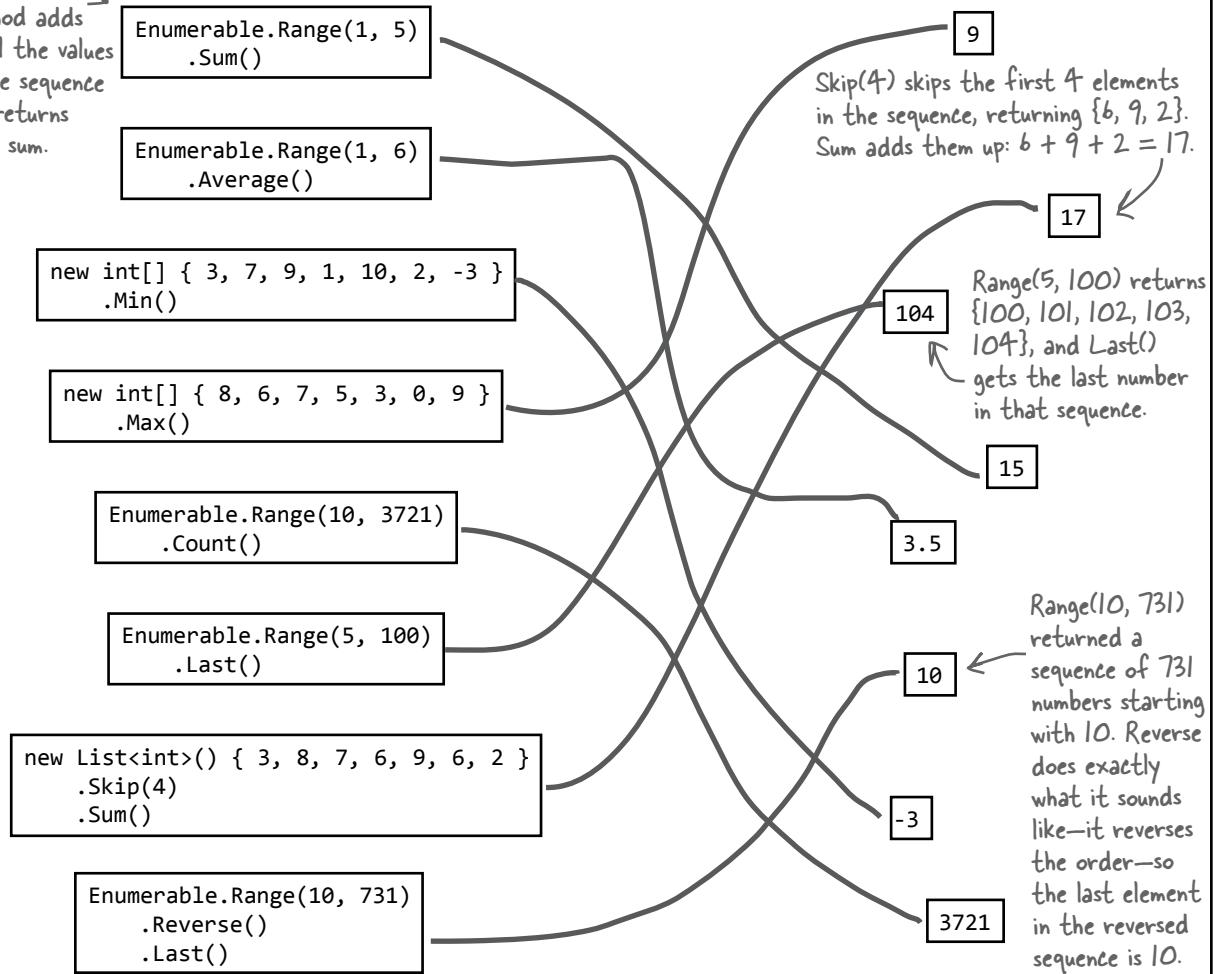
`Enumerable.Range(10, 731)
.Reverse()
.Last()`

3721

Sharpen your pencil Solution

The `Sum` method adds up all the values in the sequence and returns their sum.

Here are just a few of the LINQ methods that you'll find on your sequences when you add the `using System.Linq;` directive to your code. They have pretty intuitive names—can you figure out just from their names what they do? **Draw a line** connecting each method call to its output.



The LINQ methods in this exercise have names that make it obvious what they do. Some LINQ methods, like `Sum`, `Min`, `Max`, `Count`, `First`, and `Last`, return a single value. The `Sum` method adds up the values in the sequence. The `Average` method returns their average value. The `Min` and `Max` methods return the smallest and largest values in the sequence. The `First` and `Last` methods do exactly what it sounds like they do.

Other LINQ methods, like `Take`, `TakeLast`, `Concat`, `Reverse` (which reverses the order in a sequence), and `Skip` (which skips the first elements in a sequence), return another sequence.

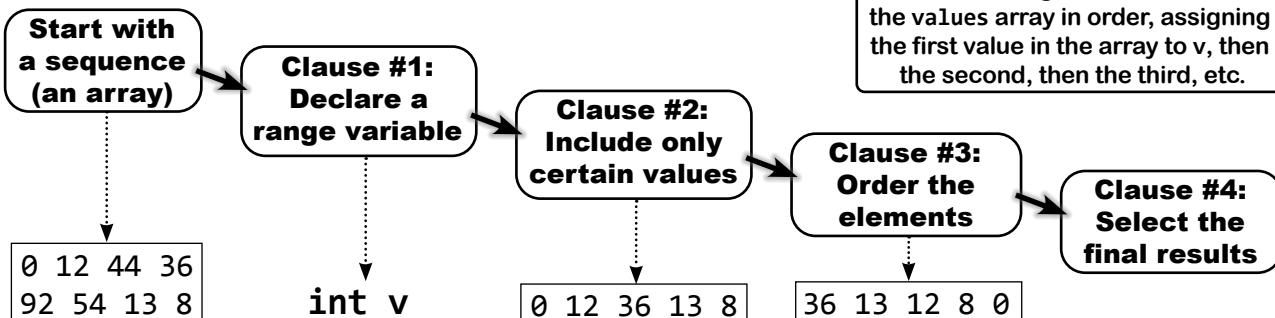
LINQ's query syntax

The LINQ methods you've seen so far may not be enough on their own to answer the kinds of questions about data that we might have—or the questions that Jimmy has about his comic collection.

And that's where the **LINQ declarative query syntax** comes in. It uses special keywords—including `where`, `select`, `groupby`, and `join`—to build **queries** directly into your code.

LINQ queries are built from clauses

Let's build a query that finds the numbers in an int array that are under 37 and puts those numbers in ascending order. It does that using four **clauses** that tell it what object to query how to determine which of its members to select, how to sort the results, and how the results should be returned.



```
int[] values = new int[] {0, 12, 44, 36, 92, 54, 13, 8};
```

```
IEnumerable<int> result =
    from v in values
    where v < 37
    orderby -v
    select v;
```

This LINQ query has four clauses: the from clause, a where clause, an orderby clause, and the select clause.

The `from` clause assigns a variable, called the **range variable**, to stand in for each value as it iterates through the array. In the first iteration the variable `v` is 0, then 12, then 44, etc.

A `where` clause contains a conditional test that the query uses to determine which values to include in the results—in this case, any value less than 37.

An `orderby` clause contains an expression used to sort the results—in this case, `-v` sorts from highest to lowest.

The query ends with a `select` clause with an expression that tells it what to put in the results.

```
// use a foreach loop to print the results
foreach(int i in result)
    Console.WriteLine("{i} ");
```

Output: 36 13 12 8 0

there are no Dumb Questions

Q: So adding a `using` directive to the top of a file “magically” adds LINQ methods to every `IEnumerable` in it?

A: Basically, yes it does. You need a `using System.Linq;` directive at the top of your file to use LINQ methods (and, later, LINQ queries). Just like you saw in the last chapter, you’ll also need `using System.Collections.Generic;` if you want to use `IEnumerable<T>`—for example, as a return value.

(Obviously, there’s no *actual* magic involved. LINQ uses a C# feature called **extension methods**, which you’ll learn about in Chapter 11. All you need to know for now is that when you add the `using` directive, you can use LINQ with any `IEnumerable<T>` reference.)

Q: I’m still not clear on what method chaining is. How does it work, exactly, and why should I use it?

A: Method chaining is a really common way of making multiple method calls in a row. Since many of the LINQ methods return a sequence that implements `IEnumerable<T>`, you can call another LINQ method on the result. Method chaining isn’t unique to LINQ, either. You can use it in your own classes.



THIS ALL
SOUNDS FINE. BUT HOW DOES
IT HELP ME MANAGE MY UNRULY COMIC
BOOK COLLECTION?

LINQ isn’t just for numbers. It works with objects, too.

When Jimmy looks at stacks and stacks of disorganized comics, he might see paper, ink, and a jumbled mess. When us developers look at them, we see something else: **lots and lots of data** just waiting to be organized. How do we organize comic book data in C#? The same way we organize playing cards, bees, or items on Sloppy Joe’s menu: we create a class, then we use a collection to manage that class. So all we need to help Jimmy out is a Comic class, and code to help us bring some sanity to his collection. And LINQ will help!

Q: Can you show me an example of a class that uses method chaining?

A: Sure. Here’s a class that has two methods built for chaining:

```
class AddSubtract
{
    public int Value { get; set; }  
    public AddSubtract Add(int i) {
        Console.WriteLine($"Value: {Value}, adding {i}");
        return new AddSubtract() { Value = Value + i };
    }
    public AddSubtract Subtract(int i) {
        Console.WriteLine(
            $"Value: {Value}, subtracting {i}");
        return new AddSubtract() { Value = Value - i };
    }
}
```

You can call it like this:

```
AddSubtract a = new AddSubtract() { Value = 5 }
    .Add(5)
    .Subtract(3)
    .Add(9)
    .Subtract(12);
Console.WriteLine($"Result: {a.Value}");
```

} Try adding the
AddSubtract
class to a new
console app,
then add this
code to the
Main method.

LINQ works with objects

Do this!

Jimmy wanted to know how much some of his prize comics are worth, so he hired a professional comic book appraiser to give him prices for each of them. It turns out that some of his comics are worth a lot of money! Let's use collections to manage that data for him.

1 Create a new console app and add a Comic class.

Use two automatic properties for the name and issue number:

```
using System.Collections.Generic;
class Comic {
    public string Name { get; set; }
    public int Issue { get; set; }

    public override string ToString() => $"{Name} (Issue #{Issue})";
```

We've never seen that => operator before! Can you figure out from its context what it does? You know how `ToString` methods work—so somehow the => makes the `ToString` return the interpolated string to the right of the operator.

2 Add a List that contains Jimmy's catalog.

Add this static Catalog field to the Comic class. It returns a sequence with Jimmy's prized comics:

```
public static readonly IEnumerable<Comic> Catalog =
    new List<Comic> {
        new Comic { Name = "Johnny America vs. the Pinko", Issue = 6 },
        new Comic { Name = "Rock and Roll (limited edition)", Issue = 19 },
        new Comic { Name = "Woman's Work", Issue = 36 },
        new Comic { Name = "Hippie Madness (misprinted)", Issue = 57 },
        new Comic { Name = "Revenge of the New Wave Freak (damaged)", Issue = 68 },
        new Comic { Name = "Black Monday", Issue = 74 },
        new Comic { Name = "Tribal Tattoo Madness", Issue = 83 },
        new Comic { Name = "The Death of the Object", Issue = 97 },
    };
```

We left the () parentheses off of the collection and object initializers after <Comic>, because you don't need 'em.

3 Use a Dictionary to manage the prices.

Add a static `Comic.Prices` field—it's a `Dictionary<int, decimal>` that lets you look up the price of each comic using its issue number (using the collection initializer syntax for dictionaries that you learned in Chapter 8). Note that we're using the **IReadOnlyDictionary interface** for encapsulation—it's an interface that includes only the methods to read values (so we don't accidentally change the prices):

```
public static readonly IReadOnlyDictionary<int, decimal> Prices =
    new Dictionary<int, decimal> {
        { 6, 3600M },
        { 19, 500M },
        { 36, 650M },
        { 57, 13525M }, ← Jimmy's rare
        { 68, 250M },
        { 74, 75M },
        { 83, 25.75M },
        { 97, 35.25M },
    };
```

We're using the IReadOnlyDictionary interface to make the dictionary to prevent code from using the Price field to change prices.

misprinted
edition of issue
#57 ("Hippie
Madness" from
1973) is worth
\$13,525. Wow!

We used a Dictionary to store the prices for the comics. We could have included a property called Price. We decided to keep information about the comic and price separate. We did this because prices for collectors' items change all the time, but the name and issue number will always be the same. Do you think we made the right choice?

Use a LINQ query to finish the app for Jimmy

We used the LINQ declarative query syntax earlier to create a query with four clauses: a **from** clause to create a range variable, a **where** clause to include only numbers under 37, an **orderby** clause to sort them in descending order, and a **select** clause to determine which elements to include in the resulting sequence.

Let's **add a LINQ query to the Main method** that works exactly the same way—except using Comic objects instead of int values, so it writes a list of comics with a price > 500 in reverse order to the console. We'll start with two **using** declarations so we can use `IEnumerable<T>` and LINQ methods. The query will return an `IEnumerable<Comic>`, then use a **foreach** loop to iterate through it and write the output.

4 Modify the Main method to use a LINQ query.

Here's the entire Program class, including the **using** directives that you needed to add to the top:

```
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        IEnumerable<Comic> mostExpensive =
            from comic in Comic.Catalog
            where Comic.Prices[comic.Issue] > 500
            orderby -Comic.Prices[comic.Issue]
            select comic;

        foreach (Comic comic in mostExpensive)
        {
            Console.WriteLine($"{comic} is worth {Comic.Prices[comic.Issue]:c}");
        }
    }
}
```

You need `System.Collections.Generic` to use the `IEnumerable<T>` interface, and you need `System.Linq` to add the LINQ methods to any object that implements it.

Keep your eye on how the "comic" range variable is used. It's a `Comic` variable that's declared in the `from` clause and used in the `where` and `orderby` clauses.

The `select` clause determines what the query returns. Since it's selecting a `Comic` variable, the result of the query is an `IEnumerable<Comic>`.

We saw in previous chapters that "`:c`" formats a number as local currency, so if you're in the UK, you'll see £ instead of \$ in the output.

Output:

```
Hippie Madness (misprinted) is worth $13,525.00
Johnny America vs. the Pinko is worth $3,600.00
Woman's Work is worth $650.00
```

5 Use the descending keyword to make your orderby clause more readable.

Your `orderby` clause uses a minus sign to negate the comic prices before sorting, causing the query to sort them in descending order. But it's easy to accidentally miss that minus sign when you're reading the code and trying to figure out how it works. Luckily, there's another way to get the same results. Remove the minus sign and **add the descending keyword** to the end of the clause:

```
orderby Comic.Prices[comic.Issue] descending ← The descending keyword makes
orderby sort in reverse order.
```



Anatomy of a query

Let's explore how LINQ works by making a couple of small changes to the query:

- ★ That minus in the `orderby` clause is easy to miss. We'll make sure to use the `descending` keyword that you just added in step 5.

Changing the `select` clause causes the query to return a sequence of strings.

- ★ The `select` clause you just wrote selected the comic, so the result of the query was a sequence of Comic references. Let's replace it with an interpolated string that uses the `comic` range variable—now the result of the query is a sequence of strings.

Here's the updated LINQ query. Each clause in the query produces a sequence that feeds into the next clause—we've added a table under each clause that shows its result.

```
IEnumerable<string> mostExpensiveComicDescriptions =
from comic in Comic.Catalog
```

{ Name = "Johnny America vs. the Pinko", Issue = 6 }
{ Name = "Rock and Roll (limited edition)", Issue = 19 }
{ Name = "Woman's Work", Issue = 36 }
{ Name = "Hippie Madness (misprinted)", Issue = 57 }
{ Name = "Revenge of the New Wave Freak (damaged)", Issue = 68 }
{ Name = "Black Monday", Issue = 74 }
{ Name = "Tribal Tattoo Madness", Issue = 83 }
{ Name = "The Death of the Object", Issue = 97 }

The `from` clause loops through `Comic.Catalog`, pulling out each value in it and assigning it to the range variable "comic". The result of the `from` clause is a sequence of Comic object references.

```
where Comic.Prices[comic.Issue] > 500
```

{ Name = "Johnny America vs. the Pinko", Issue = 6 }
{ Name = "Woman's Work", Issue = 36 }
{ Name = "Hippie Madness (misprinted)", Issue = 57 }

The `where` clause starts with the results of the `from` clause, assigning "comic" to each value and using it to apply a conditional test that checks the `Comic.Prices` dictionary for the price and only includes comics whose price is greater than 500.

```
orderby Comic.Prices[comic.Issue] descending
```

{ Name = "Hippie Madness (misprinted)", Issue = 57 }
{ Name = "Johnny America vs. the Pinko", Issue = 6 }
{ Name = "Woman's Work", Issue = 36 }

The `orderby` clause starts with the results of the `where` clause and sorts in descending order by price.

```
select $"{{comic}} is worth {Comic.Prices[comic.Issue]:c}";
```

"Hippie Madness (misprinted) is worth \$13,525.00"
"Johnny America vs. the Pinko is worth \$3,600.00"
"Woman's Work is worth \$650.00"

The `select` clause loops through the results of the `orderby` clause, using the "comic" range variable with string interpolation to return a sequence of strings.

define a variable without declaring a type

The var keyword lets C# figure out variable types for you

We just saw that when we made the small change to the `select` clause, the type of sequence that the query returned changed. When it was `select comic`; the return type was `IEnumerable<Comic>`. When we changed it to `select $"{comic} is worth {Comic.Prices[comic.Issue]:c}"`; the return type changed to `IEnumerable<string>`. When you're working with LINQ, that happens all the time—you're constantly tweaking your queries. It's not always obvious exactly what type they return. Sometimes going back and updating all of your declarations can get annoying.

Luckily, C# gives us a really useful tool to help keep variable declarations simple and readable. You can replace any variable declaration with the **var keyword**. So you can replace any of these declarations:

```
IEnumerable<int> numbers = Enumerable.Range(1, 10);
string s = $"The count is {numbers.Count()}";
IEnumerable<Comic> comics = new List<Comic>();
IReadOnlyDictionary<int, decimal> prices = Comic.Prices;
```

with these declarations, which do exactly the same thing:

```
var numbers = Enumerable.Range(1, 10);
var s = $"The count is {numbers.Count()}";
var comics = new List<Comic>();
var prices = Comic.Prices;
```

When you use the **var keyword**, you're telling C# to use an **implicitly typed** variable. We saw that same word—**implicit**—back in Chapter 8 when we talked about covariance. It means that C# figures out the types on its own.

And you don't have to change any of your code. Just replace the types with var and everything works.

When you use var, C# figures out the variable's type automatically

Go ahead—try it right now. Comment out the first line of the LINQ query you just wrote, then replace `IEnumerable<Comic>` with `var`:

```
// IEnumerable<Comic> mostExpensive =
var mostExpensive =
    from comic in Comic.Catalog
    where Comic.Prices[comic.Issue] > 500
    orderby -Comic.Prices[comic.Issue]
    select comic;
```

When you used var in the variable declaration, the IDE figured out its type based on how it was used in the code.

If you temporarily comment out the `orderby` clause in the query, that turns `mostExpensive` into an `IEnumerable<T>`.

Now hover your mouse cursor over the variable name in the `foreach` loop to see its type:

```
foreach (Comic comic in mostExpensive)
{
    Console.WriteLine($"{cc}");
```

[cc] (local variable) `IOrderedEnumerable<Comic> mostExpensive`

The IDE figured out the `mostExpensive` variable's type—and it's a type we haven't seen before. Remember in Chapter 7 when we talked about how interfaces can extend other interfaces? The `IOrderedEnumerable` interface is part of LINQ—it's used to represent a *sorted* sequence—and it extends the `IEnumerable<T>` interface. Try commenting out the `orderby` clause and hovering over the `mostExpensive` variable—you'll find that it turns into an `IEnumerable<Comic>`. That's because C# looks at the code to **figure out the type of any variable you declare with var**.



LINQ magnets

We had a nice LINQ query that used the var keyword arranged with magnets on the refrigerator—but someone slammed the door and the magnets fell off! Rearrange the magnets so they produce the output at the bottom of the page.

pigeon descending

Console.WriteLine("Get your kicks on route {0}",

weasels.Sum()

sparrow in bears

pigeon in badgers

where

from

from

select

orderby

select

var weasels =

int[] badgers =

var skunks =

var bears =

);

skunks

pigeon + 5;

sparrow - 1;

(pigeon != 36 && pigeon < 50)

{ 36, 5, 91, 3, 41, 69, 8 };

Output:

Get your kicks on route 66



LINQ Magnets Solution

Rearrange the magnets so they produce the output at the bottom of the page.

LINQ starts with some sort of sequence, collection, or array—in this case, an array of integers.

`int[] badgers =`

`{ 36, 5, 91, 3, 41, 69, 8 };`

We chose confusing names like skunks, badgers, and bears on purpose. “from pigeon in badgers” makes for a good puzzle, but the code is pretty hard to understand. Renaming the array to “numbers” and using “from number in numbers” to declare a more sensibly named range variable would make it more readable.

After this statement,
skunks contains four
numbers: 46, 13, 10, and 8.

```
var skunks =
    from pigeon in badgers
    where (pigeon != 36 && pigeon < 50)
    orderby pigeon descending
    select pigeon + 5;
```

This LINQ statement pulls all the numbers that are below 50 and not equal to 36 out of the array, adds 5 to each of them, sorts them from biggest to smallest, puts them in a new object, and points the skunks reference at it.

After this
statement, bears
contains three
numbers: 46, 13,
and 10.

```
var bears =
    skunks .Take(3);
```

Here's where we take the first three numbers in skunks and put them into a new sequence called bears.

After this statement,
weasels contains three
numbers: 45, 12, and 9.

```
var weasels =
    from sparrow in bears
    select sparrow - 1;
```

This statement just subtracts 1 from each number in bears and puts them all into weasels.

$$45 + 12 + 9 = 66$$

```
Console.WriteLine("Get your kicks on route {0}",  
    weasels.Sum() );
```

The numbers in weasels add up to 66.

Output:

Get your kicks on route 66



ARE YOU SERIOUSLY TELLING ME THAT I CAN
REPLACE THE TYPE IN ANY VARIABLE DECLARATION
WITH **VAR** AND MY CODE WILL STILL WORK?
IT CAN'T BE THAT SIMPLE.

You really can use var in your variable declarations.

And yes, it really is that simple. A lot of C# developers declare local variables using var almost all the time, and include the type only when it makes the code easier to read. As long as you're declaring the variable and initializing it in the same statement, you can use var.

But there are some important restrictions on using var. For example:

- You can only declare one variable at a time with var.
- You can't use the variable you're declaring in the declaration.
- You can't declare it equal to null.

If you create a variable named var, you won't be able to use it as a keyword anymore:

- You definitely can't use var to declare a field or a property—you can only use it as a local variable inside a method.
- If you stick to those ground rules, you can use var pretty much anywhere.

So when you did this in Chapter 4:

```
int hours = 24;
short RPM = 33;
long radius = 3;
char initial = 'S';
int balance = 345667 - 567;
```

Or this in Chapter 6:

```
SwordDamage swordDamage = new SwordDamage(RollDice(3));
ArrowDamage arrowDamage = new ArrowDamage(RollDice(1));
```

Or this in Chapter 8:

```
List<Card> cards = new List<Card>();
```

You could have done this:

```
var hours = 24;
var RPM = 33;
var radius = 3;
var initial = 'S';
var balance = 345667 - 567;
```

Or this:

```
var swordDamage = new SwordDamage(RollDice(3));
var arrowDamage = new ArrowDamage(RollDice(1));
```

Or this:

```
var cards = new List<Card>();
```

...and your code would have worked exactly the same.

But you can't use var to declare a field or property:

```
class Program
{
    static var random = new Random(); // this will cause a compiler error

    static void Main(string[] args)
{
```

there are no
Dumb Questions

Q: How does the `from` clause work?

A: It's a lot like the first line of a `foreach` loop. One thing that makes thinking about LINQ queries a little tricky is that you're not just doing one operation. A LINQ query does the same thing over and over again for each item in a collection—in other words, it enumerates the sequence. So the `from` clause does two things: it tells LINQ which collection to use for the query, and it assigns a name to use for each member of the collection that's being queried.

The way the `from` clause creates a new name for each item in the collection is really similar to how a `foreach` loop does it. Here's the first line of a `foreach` loop:

```
foreach (int i in values)
```

That `foreach` loop temporarily creates a variable called `i` which it assigns sequentially to each item in the `values` collection. Now look at a `from` clause in a LINQ query on the same collection:

```
from i in values
```

That clause does pretty much the same thing. It creates a range variable called `i` and assigns it sequentially to each item in the `values` collection. The `foreach` loop runs the same block of code for each item in the collection, while the LINQ query applies the same criteria in the `where` clause to each item in the collection to determine whether or not to include it in the results.

Q: You took a LINQ query that returned a sequence of Comic references and made it return strings. What exactly did you do to make that work?

A: We modified the `select` clause. The `select` clause includes an expression that gets applied to every item in the sequence, and that expression determines the type of the output. So if your query produces a sequence of values or object references, you can use string interpolation in the `select` clause to turn each item in that sequence into a string. The query in the exercise solution ended with `select comic`, so it returned a sequence of Comic references. In our "Anatomy of a Query" code, we replaced it with `select $"{comic} is worth {Comic.Prices[comic.Issue]:c}"`—which caused the query to return a sequence of strings instead.

Q: How does LINQ decide what goes into the results?

A: That's what the `select` clause is for. Every LINQ query returns a sequence, and every item in that sequence is of the same type. It tells LINQ exactly what that sequence should contain. When you're querying an array or list of a single type—like an array of ints or a `List<string>`—it's obvious what goes into the `select` clause. What if you're selecting from a list of `Comic` objects? You could do what Jimmy did and select the whole class. You could also change the last line of the query to `select comic.Name` to tell it to return a sequence of strings. Or you could do `select comic.Issue` and have it return a sequence of ints.

Q: I understand how to use `var` in my code, but how does it actually work?

A: `var` is a keyword that tells the compiler to figure out the type of a variable at compilation time. The C# compiler figures out the type of the local variable that you're using LINQ to query. When you build your solution, the compiler will replace `var` with the right type for the data you're working with.

So when this line is compiled:

```
var result = from v in values
```

The compiler replaces `var` with this:

```
IEnumerable<int>
```

And you can always check a variable's type by hovering over it in the IDE.

The `from` clause in a LINQ query does two things: it tells LINQ which collection to use for the query, and it assigns a name to use for each member of the collection being queried.

Q: LINQ queries use a lot of keywords I haven't seen before—from, where, orderby, select...it's like a whole different language. Why does it look so different from the rest of C#?

A: Because LINQ serves a different purpose. Most of the C# syntax was built to do one small operation or calculation at a time. You can start a loop, set a variable, do a mathematical operation, call a method...those are all single operations. For example:

```
var under10 =
    from number in sequenceOfNumbers
    where number < 10
    select number;
```

BULLET POINTS

- When a class implements `IEnumerable<T>`, any instance of that class is a **sequence**.
- When you add `using System.Linq;` to the top of your code, you can use **LINQ methods** with any reference to a sequence.
- When a method goes through each item in a sequence in order, that's called **enumerating** the sequence, which is how LINQ methods work.
- The **Take method** takes the first elements from a sequence. The **TakeLast method** takes the last elements from a sequence. The **Concat method** concatenates two sequences together.
- The **Average method** returns the average value of a sequence of numbers. The **Min and Max methods** return the smallest and largest values in the sequence.
- The **First and Last methods** return the first or last element in a sequence. The **Skip method** skips the first elements of a sequence and returns the rest.
- Many LINQ methods return a sequence, which lets you do **method chaining**, or calling another LINQ method directly on the results without using an extra variable to keep track of those results.
- The **IReadOnlyDictionary interface** is useful for encapsulation. You can assign any dictionary to it to create a reference that doesn't allow the dictionary to be updated.

That query looks pretty simple—not a lot of stuff there, right? But this is actually a pretty complex piece of code.

Think about what's got to happen for the program to actually select all the numbers from `sequenceOfNumbers` (which must be a reference to an object that implements `IEnumerable<T>`) that are less than 10. First, it needs to loop through the entire array. Then, each number is compared to 10. Then those results need to be gathered together so the rest of the code can use them.

And that's why LINQ looks a little odd: because it lets you cram a whole lot of behavior into a small—but easily readable!—amount of C# code.

- The **LINQ declarative query syntax** uses special keywords—including `where`, `select`, `groupby`, and `join`—to build queries directly into your code.
- LINQ queries start with a **from clause**, which assigns a variable to stand in for each value as it enumerates the sequence.
- The variable declared in the `from` clause is the **range variable**, and it can be used throughout the query.
- A **where clause** contains a conditional test that the query uses to determine which values to include in the results.
- An **orderby clause** contains an expression used to sort the results. You can optionally specify the **descending** keyword to reverse the sort order.
- The query ends with a **select clause** that includes an expression to specify what to include in the results.
- The **var keyword** is used to declare an implicitly typed variable, which means the C# compiler figures out the type of the variable on its own.
- You can use `var` in place of a variable type in any declaration statement that initializes a variable.
- You can include a **C# expression in a select clause**. That expression is applied to every element in the results, and determines the type of sequence returned by the query.

LINQ is versatile

With LINQ, you can do a lot more than just pull a few items out of a collection. You can modify the items before you return them. Once you've generated a set of result sequences, LINQ gives you a bunch of methods that work with them. Top to bottom, LINQ gives you the tools you need to manage your data. Let's do a quick review of some of the LINQ features that we've already seen.

You can use the var keyword to declare an implicitly typed array. Just use new[] along with a collection initializer, and the C# compiler will figure out the type of the array for you. If you mix and match types, you need to specify the array type:

```
var mixed = new object[] { 1, "x" , new Random() };
```

★ Modify every item returned from the query.

This code will add a string onto the end of each element in an array of strings.

It doesn't change the array itself—it creates a new sequence of modified strings.

```
var sandwiches = new[] { "ham and cheese", "salami with mayo",
                        "turkey and swiss", "chicken cutlet" };
var sandwichesOnRye =
    from sandwich in sandwiches
    select $"{sandwich} on rye";
foreach (var sandwich in sandwichesOnRye)
    Console.WriteLine(sandwich);
```

Now all the items returned have
"on rye" added to the end.

} You can think of "select" as a way to make the
same change to every element in a sequence—in
this case, add "on rye" to the end.

Output:

```
ham and cheese on rye
salami with mayo on rye
turkey and swiss on rye
chicken cutlet on rye
```

★ Perform calculations on sequences.

You can use the LINQ methods on their own to get statistics about a sequence of numbers.

```
var random = new Random();
var numbers = new List<int>();
int length = random.Next(50, 150);
for (int i = 0; i < length; i++)
    numbers.Add(random.Next(100));
```

```
Console.WriteLine($"Stats for these {numbers.Count()} numbers:
The first 5 numbers: {String.Join(", ", numbers.Take(5))}
The last 5 numbers: {String.Join(", ", numbers.TakeLast(5))}
The first is {numbers.First()} and the last is {numbers.Last()}
The smallest is {numbers.Min()}, and the biggest is {numbers.Max()}
The sum is {numbers.Sum()}
The average is {numbers.Average():F2}");
```

Here's the output from a sample run. The
length of the sequence and the numbers
in it will be random each time you run it.

The static `String.Join` method
concatenates all of the items in a
sequence into a string, specifying
the separator to use between them.

```
Stats for these 61 numbers:
The first 5 numbers: 85, 30, 58, 70, 60
The last 5 numbers: 40, 83, 75, 26, 75
The first is 85 and the last is 75
The smallest is 2, and the biggest is 99
The sum is 3444
The average is 56.46
```

LINQ queries aren't run until you access their results

When you include a LINQ query in your code, it uses **deferred evaluation** (sometimes called lazy evaluation). That means the LINQ query doesn't actually do any enumerating or looping until your code executes a statement that **uses the results of the query**. That sounds a little weird, but it makes a lot more sense when you see it in action. **Create a new console app** and add this code:

```
class PrintWhenGetting
{
    private int instanceNumber;
    public int InstanceNumber
    {
        set { instanceNumber = value; }
        get
        {
            Console.WriteLine($"Getting #{instanceNumber}");
            return instanceNumber;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        var listOfObjects = new List<PrintWhenGetting>();
        for (int i = 1; i < 5; i++)
            listOfObjects.Add(new PrintWhenGetting() { InstanceNumber = i });

        Console.WriteLine("Set up the query");
        var result =
            from o in listOfObjects
            select o.InstanceNumber;

        Console.WriteLine("Run the foreach");
        foreach (var number in result)
            Console.WriteLine($"Writing #{number}");
    }
}
```

Now run your app. Notice how the `Console.WriteLine` that prints "Set up the query" runs **before** the get accessor ever executes. That's because the LINQ query won't get executed until the `foreach` loop.

If you need the query to execute right now, you can force **immediate execution** by calling a LINQ method that needs to enumerate the entire list—for example, the `ToList` method, which turns it into a `List<T>`. Add this line, and change the `foreach` to use the new `List`:

```
var immediate = result.ToList();

Console.WriteLine("Run the foreach");
foreach (var number in immediate)
    Console.WriteLine($"Writing #{number}");
```

Run the app again. This time you'll see the get accessors called before the `foreach` loop starts executing—which makes sense, because `ToList` needs to access every element in the sequence to convert it to a `List`. Methods like `Sum`, `Min`, and `Max` also need to access every element in the sequence, so when you use them you'll force LINQ to do immediate execution as well.

The `Console.WriteLine` in the getter isn't called until the `foreach` loop actually executes. That's what deferred execution looks like.

Set up the query
Run the foreach
Getting #1
Writing #1
Getting #2
Writing #2
Getting #3
Writing #3
Getting #4
Writing #4

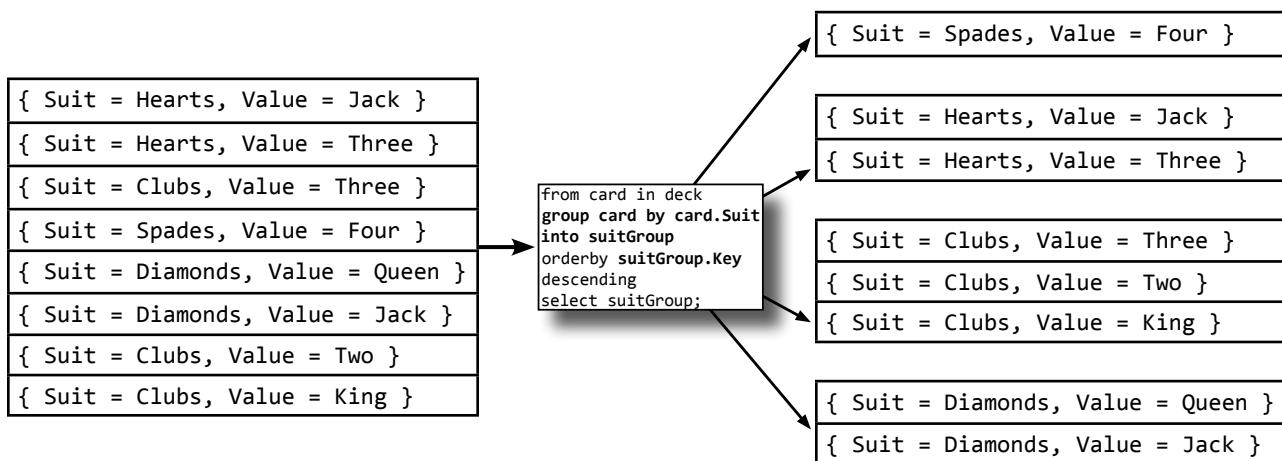
When you call `ToList` or another LINQ method that needs to access every element in the sequence, you'll get immediate evaluation.

Set up the query
Getting #1
Getting #2
Getting #3
Getting #4
Run the foreach
Writing #1
Writing #2
Writing #3
Writing #4

Use a group query to separate your sequence into groups

Sometimes you really want to slice and dice your data. For example, Jimmy might want to group his comics by the decade they were published. Or maybe he wants to separate them by price (cheap ones in one collection, expensive ones in another). There are lots of reasons you might want to group your data together. That's where the **LINQ group query** comes in handy.

Group
this!



1 Create a new console app and add the card classes and enums.

Create a new .NET Core console app named **CardLinq**. Then go to the Solution Explorer panel, right-click on the project name, and choose Add >> Existing Items (or Add >> Existing Files on a Mac). Navigate to the folder where you saved the Two Decks project from Chapter 8. Add the files with the **Suit and Value enums**, then add the **Deck, Card, and CardComparerByValue classes**.

Make sure you **modify the namespace in each file you added** to match the namespace in *Program.cs* so your Main method can access the classes you added.

You created the Deck class in the downloadable “Two Decks” project at the end of Chapter 8.

2 Make your Card class sortable with **IComparable<T>**.

We'll be using a LINQ **orderby** clause to sort groups, so we need the Card class to be sortable. Luckily, this works exactly like the **List.Sort** method, which you learned about in Chapter 7. Modify your Card class to **extend the IComparable interface**:

```

class Card : IComparable<Card>
{
    public int CompareTo(Card other)
    {
        return new CardComparerByValue().Compare(this, other);
    }

    // the rest of the class stays the same
  
```



We'll also be using the **LINQ Min** and **Max** methods to find the highest and lowest card in each group, and they also use the **IComparable interface**.

3 Modify the Deck.Shuffle method to support method chaining.

The Shuffle class shuffles the deck. All you need to do to make it support method chaining is modify it to return a reference to the Deck instance that just got shuffled:

```
public Deck Shuffle()
{
    // The rest of the class stays the same
    return this;
}
```

When you make the Shuffle method return a reference to the same Deck object being shuffled, you can call it and then chain additional method calls to the result..

4 Use a LINQ query with group...by clause to group the cards by suit.

The Main method will get 16 random cards by shuffling the deck, then using the LINQ Take method to pull the first 16 cards. Then it will use a LINQ query with a **group...by clause** to separate the deck into smaller sequences, with one sequence for each suit in the 16 cards:

```
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        var deck = new Deck()
            .Shuffle()
            .Take(16);

        var grouped =
            from card in deck
            group card by card.Suit into suitGroup
            orderby suitGroup.Key descending
            select suitGroup;

        foreach (var group in grouped)
        {
            Console.WriteLine($"Group: {group.Key}");
            Count: {group.Count()}
            Minimum: {group.Min()}
            Maximum: {group.Max()});
        }
    }
}
```

Use LINQ's Count, Min, and Max methods to get information about each group the query returns.

Now that the Shuffle method supports method chaining, you can chain the LINQ Take method right after it.

Each group has a Key property that returns its key—in this case, a suit.

A group...by clause in a LINQ query separates a sequence into groups:

group card by card.Suit into suitGroup

The group keyword tells it which sequence contains the elements to group, the by keyword specifies the criteria used to determine the groups, and the into keyword declares a new variable that the other clauses can use to refer to the groups. The output of a group query is a sequence of sequences.

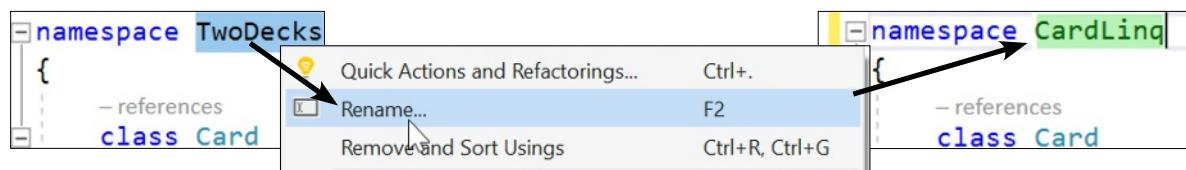
Each group is a sequence that implements the IGrouping interface: IGrouping<Suits, Card> is a group of cards that uses a suit as its group key.

Hover over the "grouped" variable to see its type.

(local variable) `IOrderedEnumerable<IGrouping<Suits, Card>> grouped`

IDE Tip: Rename anything!

When you need to change the name of a variable, field, property, namespace, or class, you can use a really handy **refactoring tool** built into Visual Studio. Just right-click on it and choose "Rename..." from the menu. When the IDE highlights it, edit its name—and the IDE will **automatically rename it everywhere in the code**.



Use Rename to change the name of a **variable, field, property, class, or namespace** (and a few other things, too!). When you rename one occurrence, the IDE changes its name everywhere it occurs in the code.



Anatomy of a group query

Let's take a closer look at how that group query works.

```
var grouped =
    from card in deck
```

This random sample happens to start with two cards with the suit Hearts, followed by a card with the suit Clubs, then a Spades card, then two Diamonds cards.

{ Suit = Hearts, Value = Jack }
{ Suit = Hearts, Value = Three }
{ Suit = Clubs, Value = Three }
{ Suit = Spades, Value = Four }
{ Suit = Diamonds, Value = Queen }
{ Suit = Diamonds, Value = Jack }
{ Suit = Clubs, Value = Two }
{ Suit = Clubs, Value = King }

group card by card.Suit into suitGroup

The group...by clause enumerates the sequence, creating new groups as it gets to each new key. So the groups are in the same order as the first occurrences of the suits that were in the random sample.

{ Suit = Hearts, Value = Jack }
{ Suit = Hearts, Value = Three }
{ Suit = Clubs, Value = Three }
{ Suit = Clubs, Value = Two }
{ Suit = Clubs, Value = King }
{ Suit = Spades, Value = Four }
{ Suit = Diamonds, Value = Queen }
{ Suit = Diamonds, Value = Jack }

orderby suitGroup.Key descending
select suitGroup;

The group...by clause grouped the cards by card.Suit, so each group's key is the suit. That means all of the cards in each group have the same suit, and all of the cards with that suit are in the group. The orderby clause sorted the groups by key, which put them in the order they appear in the Suits enum (in reverse): Spades, Hearts, Clubs, and Diamonds.

{ Suit = Spades, Value = Four }
{ Suit = Hearts, Value = Jack }
{ Suit = Hearts, Value = Three }
{ Suit = Clubs, Value = Three }
{ Suit = Clubs, Value = Two }
{ Suit = Clubs, Value = King }
{ Suit = Diamonds, Value = Queen }
{ Suit = Diamonds, Value = Jack }

The from clause works just like in the other LINQ queries you've used. It assigns the range variable "card" to each card in the sequence—in this case, the Deck that you shuffled and then pulled some cards from.

The group...by clause splits the cards into groups. It includes "by card.Suit"—that specifies that the key for each group is the card's suit. It declares a new variable called suitGroup that the remaining clauses can use to work with the groups.

The group...by clause creates a sequence of groups that implement the IGrouping interface. IGrouping extends IEnumerable and adds exactly one member: a property called Key. So each group is a sequence of other sequences—in this case, it's a group of Card sequences, where the key is the suit of the card (from the Suits enum). The full type of each group is IGrouping<Suits, Card>, which means it's a sequence of Card sequences, each of which has a Suits value as its key.

Use join queries to merge data from two sequences

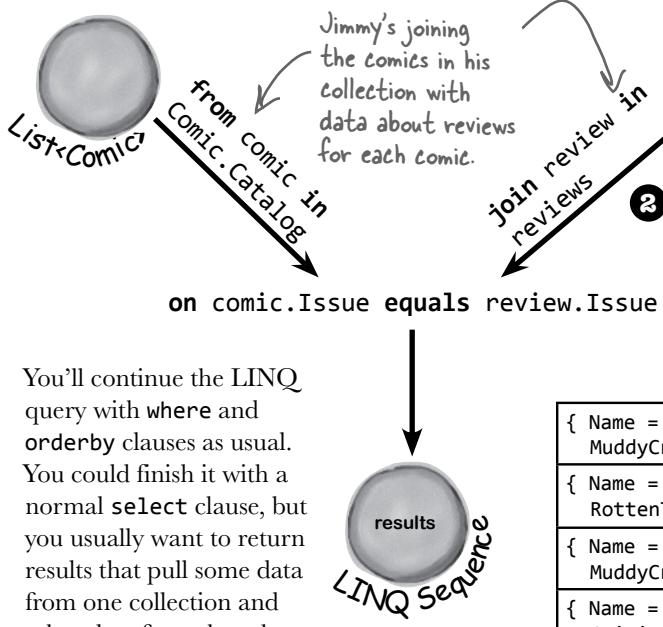
Every good collector knows that critical reviews can have a big impact on values. Jimmy's been keeping track of reviewer scores from two big comic review aggregators, MuddyCritic and Rotten Tornadoes. Now he needs to match them up to his collection. How's he going to do that?

LINQ to the rescue! Its **join** keyword lets you **combine data from two sequences** using a single query. It does it by comparing items in one sequence with their matching items in a second sequence. (LINQ is smart enough to do this efficiently—it doesn't actually compare every pair of items unless it has to.) The final result combines every pair that matches.

- 1 You'll start your query with the usual **from** clause. But instead of following it up with the criteria to determine what goes into the results, you'll add:

```
join name in collection
```

The **join** clause tells LINQ to enumerate both sequences to match up pairs with one member from each. It assigns **name** to the member it'll pull out of the joined collection in each iteration. You'll use that name in the **where** clause.



Jimmy's got his data in a collection of Review objects called "reviews".

```
class Review
{
    public int Issue { get; set; }
    public Critics Critic { get; set; }
    public double Score { get; set; }
}
```

- 2 Next you'll add the **on** clause, which tells LINQ how to match the two collections together. You'll follow it with the name of the member of the first collection you're matching, followed by **equals** and the name of the member of the second collection to match it up with.

{ Name = "Woman's Work", Issue = 36, Critic = MuddyCritic, Score = 37.6 }
{ Name = "Black Monday", Issue = 74, Critic = RottenTornadoes, Score = 22.8 }
{ Name = "Black Monday", Issue = 74, Critic = MuddyCritic, Score = 84.2 }
{ Name = "The Death of the Object", Issue = 97, Critic = MuddyCritic, Score = 98.1 }

The result is a sequence of objects that have **Name** and **Issue** properties from **Comic**, but **Critic** and **Score** properties from **Review**. The result can't be a sequence of **Comic** objects, but it also can't be a sequence of **Review** objects, because neither class has all of those properties. The result is a different kind of type: **an anonymous type**.

Use the new keyword to create anonymous types

You've been using the `new` keyword since Chapter 3 to create instances of objects. Every time you use it, you include a type (so the statement `new Guy();` creates an instance of the type `Guy`). You can also use the `new` keyword without a type to create an **anonymous type**. That's a perfectly valid type that has read-only properties, but doesn't have a name. The type we returned from the query that joined Jimmy's comics to reviews is an anonymous type. You can add properties to your anonymous type by using an object initializer. Here's what that looks like:

```
public class Program
{
    public static void Main()
    {
        var whatAmI = new { Color = "Blue", Flavor = "Tasty", Height = 37 };
        Console.WriteLine(whatAmI);
    }
}
```

Try pasting that into a new console app and running it. You'll see this output:

```
{ Color = Blue, Flavor = Tasty, Height = 37 }
```

Now hover over `whatAmI` in the IDE and have a look at the IntelliSense window:



The IDE knows exactly what the type is: it's an object type with two string properties and an int property. It just doesn't have a name for the type. That's why it's an anonymous type.

The `whatAmI` variable is a reference type, just like any other reference. It points to an object on the heap, and you can use it to access that object's members—in this case, two of its properties:

```
Console.WriteLine($"My color is {whatAmI.Color} and I'm {whatAmI.Flavor}");
```

Besides the fact that they don't have names, anonymous types are just like any other types.



I BET THAT'S WHY WE'RE LEARNING ABOUT THE VAR KEYWORD NOW. WE NEED IT TO DECLARE ANONYMOUS TYPES.

Right! You use var to declare anonymous types.

In fact, that's one of the most important uses of the `var` keyword.



Sharpen your pencil



Joe, Bob, and Alice are some of the top competitive Go Fish players in the world. This LINQ code joins two arrays with anonymous types to generate a list of their winnings. Read through the code and write the output that it writes to the console.

```
var players = new[]
{
    new { Name = "Joe", YearsPlayed = 7, GlobalRank = 21 },
    new { Name = "Bob", YearsPlayed = 5, GlobalRank = 13 },
    new { Name = "Alice", YearsPlayed = 11, GlobalRank = 17 },
};

var playerWins = new[]
{
    new { Name = "Joe", Round = 1, Winnings = 1.5M },
    new { Name = "Alice", Round = 2, Winnings = 2M },
    new { Name = "Bob", Round = 3, Winnings = .75M },
    new { Name = "Alice", Round = 4, Winnings = 1.3M },
    new { Name = "Alice", Round = 5, Winnings = .7M },
    new { Name = "Joe", Round = 6, Winnings = 1M },
};

var playerStats =
    from player in players
    join win in playerWins
    on player.Name equals win.Name
    orderby player.Name
    select new
    {
        Name = player.Name,
        YearsPlayed = player.YearsPlayed,
        GlobalRank = player.GlobalRank,
        Round = win.Round,
        Winnings = win.Winnings,
    };

foreach (var stat in playerStats)
    Console.WriteLine(stat);
```

We're using var and new[] to create arrays with anonymous types.

This code writes six lines to the console. We started you out by filling in the first two lines. Notice how both of those lines have the same name ("Alice"). A join query will find every match between the key properties in both sequences. If there are multiple matches, the results will include one element for each match. If there's a key in one input sequence that doesn't have a match in the other, it won't be included in the results.

{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 2, Winnings = 2 }
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 4, Winnings = 1.3 }

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

Joe, Bob, and Alice are some of the top competitive Go Fish players in the world. This LINQ code joins two arrays with anonymous types to generate a list of their winnings. Read through the code and write the output that it writes to the console.



```
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 2, Winnings = 2 }  
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 4, Winnings = 1.3 }  
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 5, Winnings = 0.7 }  
{ Name = Bob, YearsPlayed = 5, GlobalRank = 13, Round = 3, Winnings = 0.75 }  
{ Name = Joe, YearsPlayed = 7, GlobalRank = 21, Round = 1, Winnings = 1.5 }  
{ Name = Joe, YearsPlayed = 7, GlobalRank = 21, Round = 6, Winnings = 1 }
```

there are no Dumb Questions

Q: Can you rewind a minute and explain what var is again?

A: Yes, definitely. The var keyword solves a tricky problem that LINQ brings with it. Normally, when you call a method or execute a statement, it's absolutely clear what types you're working with. If you've got a method that returns a string, for instance, then you can only store its results in a string variable or field.

But LINQ isn't quite so simple. When you build a LINQ statement, it might return an anonymous type that *isn't defined anywhere in your program*. Yes, you know that it's going to be a sequence of some sort. But what kind of sequence will it be? You don't know—because the objects that are contained in the sequence depend entirely on what you put in your LINQ query. Take this query, for example, from the code we wrote earlier for Jimmy. We originally wrote this:

```
IEnumerable<Comic> mostExpensive =  
    from comic in Comic.Catalog  
    where Comic.Prices[comic.Issue] > 500  
    orderby -Comic.Prices[comic.Issue]  
    select comic;
```

But then we changed the first line to use the var keyword:

```
var mostExpensive =
```

And that's useful. For example, if we changed the last line to this:

```
select new {  
    Name = comic.Name,  
    IssueNumber = $"#{comic.Issue}"  
};
```

the updated query would return a different (but perfectly valid!) type—an anonymous type with two members, a string called Name and a string called IssueNumber. But we don't have a class definition for that type anywhere in our program! You don't actually need to run the program to see exactly how that type is defined, but the mostExpensive variable still needs to be declared with *some* type.

That's why C# gives us the var keyword, which tells the compiler, "OK, we know that this is a valid type, but we can't exactly tell you what it is right now. So why don't you just figure that out yourself and not bother us with it? Thanks so much."

IDE Tip: Refactoring tools

Switch variables between implicit and explicit types

When you're working with group queries, you'll often use the `var` keyword—not just because it's convenient, but because the type returned by the group query can be a little cumbersome.

```
var grouped = → [local variable] IOrderedEnumerable<IGrouping<Suits, Card>> grouped
    from card in deck
    group card by card.Suit into suitGroup
    orderby suitGroup.Key descending
    select suitGroup;
```

But sometimes our code is actually easier to understand if we use the **explicit type**. Luckily, the IDE makes it easy to switch between the implicit type (`var`) and the explicit type for any variable. Just open the Quick Actions menu and choose “**Use explicit type instead of ‘var’**” to convert the `var` to its explicit type.



You can also choose “**Use implicit type**” from the Quick Actions menu to change the variable back to `var`.

Extract methods

Your code will often be easier to read if you take a large method and break it into smaller ones. That's why one of the most common ways that developers refactor code is **extracting methods**, or taking a block of code from a large method and moving it into its own method. The IDE gives you a really useful refactoring tool to make that easy.

The screenshot shows the Visual Studio code editor with the following code selected:

```
18 static void Main(string[] args)
19 {
20     var deck = new Deck().Shuffle();
21
22     var grouped =
23         from card in deck
24         group card by card.Suit into suitGroup
25         orderby suitGroup.Key descending
26         select suitGroup;
27 }
```

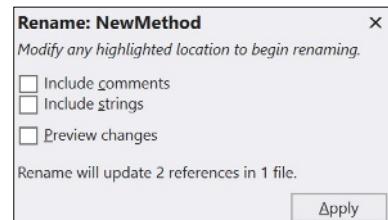
Start by selecting a block of code. Then **choose Refactor >> Extract Method** from the Edit menu (on Windows) or **Extract Method** (on a Mac) from the Quick Actions.

As soon as you do, the IDE moves the selected code into a new method called `NewMethod` with a return type that matches the type of the code that's returned. Then it immediately jumps into the Rename feature so you can start typing a new method name.

In this screenshot, we selected the entire LINQ query from the card grouping project earlier in the chapter. After we extracted the method, this is what it looked like:

```
IOrderedEnumerable<IGrouping<Suits, Card>> grouped = NewMethod(deck);
```

Notice that it left a new explicitly typed `grouped` variable—the IDE figured out that the variable is used later in the code and left a variable in place. That's yet another example of how the IDE helps write cleaner code.



there are no Dumb Questions

Q: Can you give me a little more detail on how join works?

A: join works with any two sequences. Let's say you're printing t-shirts for football players, using a collection called `players` with objects that have a `Name` property and a `Number` property. What if you need a different design for players with double-digit numbers? You could pull out the players with numbers greater than 10:

```
var doubleDigitPlayers =
    from player in players
    where player.Number > 10
    select player;
```

Now what if you need to get their shirt sizes? If you have a sequence called `jerseys` whose items have a `Number` property and a `Size` property, a join would work really well for combining the data:

```
var doubleDigitShirtSizes =
    from player in players
    where player.Number > 10
    join shirt in jerseys
    on player.Number equals shirt.Number
    select shirt;
```

Q: That query will just give me a bunch of objects. What if I want to connect each player to their shirt size, and I don't care about the number at all?

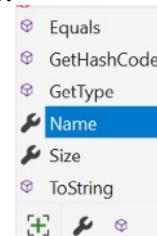
A: That's what **anonymous types** are for—you can construct an anonymous type that only has the data you want in it. It lets you pick and choose from the various collections that you're joining together, too.

So you can select the player's name and the shirt's size, and nothing else:

```
var doubleDigitShirtSizes =
    from player in players
    where player.Number > 10
    join shirt in jerseys
    on player.Number equals shirt.Number
    select new {
        player.Name,
        shirt.Size
    };
```

The IDE is smart enough to figure out exactly what results you'll be creating with your query. If you create a loop to enumerate through the results, as soon as you type the variable name the IDE will pop up an IntelliSense list:

```
foreach (var size in doubleDigitShirtSizes)
    size.
```



Notice how the list has `Name` and `Size` in it. If you added more items to the `select` clause, they'd show up in the list too, because the query would create a different anonymous type with different members.

Q: How do I write a method that returns an anonymous type?

A: You don't. Methods cannot return anonymous types. C# doesn't give you a way to do that. You can't declare a field or a property with an anonymous type, either. You also can't use an anonymous type for a parameter in a method or a constructor—that's why you can't use the `var` keyword with any of those things.

And when you think about it, these things make sense. Whenever you use `var` in a variable declaration, you always have to include a value, which the C# compiler or IDE uses to figure out the type of the variable. If you're declaring a field or a method parameter, there's no way to specify that value—which means there's no way for C# to figure out the type. (Yes, you can specify a value for a property, but that's not really the same thing—technically, the value is set just before the constructor is called.)

You can only use the var keyword when you're declaring a variable. You can't use var with a field or a property, or write a method that returns an anonymous type or takes one as a parameter.



Exercise

Use what you've learned about LINQ so far to build a new console app called **JimmyLinq** that organizes Jimmy's comic collection. Start by adding a **Critics** enum with two members, **MuddyCritic** and **RottenTornadoes**, and a **PriceRange** enum with two members, **Cheap** and **Expensive**. Then add a **Review** class with three automatic properties: int Issue, Critics Critic, and double Score.

You'll need data, so add a new static field to the **Comic** class that returns a sequence of reviews:

```
public static readonly IEnumerable<Review> Reviews = new[] {
    new Review() { Issue = 36, Critic = Critics.MuddyCritic, Score = 37.6 },
    new Review() { Issue = 74, Critic = Critics.RottenTornadoes, Score = 22.8 },
    new Review() { Issue = 74, Critic = Critics.MuddyCritic, Score = 84.2 },
    new Review() { Issue = 83, Critic = Critics.RottenTornadoes, Score = 89.4 },
    new Review() { Issue = 97, Critic = Critics.MuddyCritic, Score = 98.1 },
};
```

Here's the Main method, plus two methods it calls:

```
static void Main(string[] args) {
    var done = false;
    while (!done) {
        Console.WriteLine(
            "\nPress G to group comics by price, R to get reviews, any other key to quit\n");
        switch (Console.ReadKey(true).KeyChar.ToString().ToUpper()) {
            case "G":
                done = GroupComicsByPrice();
                break;
            case "R":
                done = GetReviews();
                break;
            default:
                done = true;
                break;
        }
    }
}

private static bool GroupComicsByPrice() {
    var groups = ComicAnalyzer.GroupComicsByPrice(Comic.Catalog, Comic.Prices);
    foreach (var group in groups) {
        Console.WriteLine($"{group.Key} comics:");
        foreach (var comic in group)
            Console.WriteLine($"#{comic.Issue} {comic.Name}: {Comic.Prices[comic.Issue]}:{c}");
    }
    return false;
}

private static bool GetReviews() {
    var reviews = ComicAnalyzer.GetReviews(Comic.Catalog, Comic.Reviews);
    foreach (var review in reviews)
        Console.WriteLine(review);
    return false;
}
```

Look closely at this while loop. It uses a switch to determine which method to call. The methods return true, setting "done" to true and the while loop to do another iteration. If the user presses any other key, it sets "done" to false and ends the loop.

The foreach loops in the GroupComicsByPrice method are nested: one loop is inside the other. The outer loop prints information about each group, and the inner one enumerates the group.

The GroupComicsByPrice and GetReviews methods call methods in the static ComicAnalyzer class (which you'll write) that run LINQ queries.

Your job is to **create a static class called ComicAnalyzer** with three static methods (two of which are public):

- A private static method called CalculatePriceRange takes a Comic reference and returns PriceRange.Cheap if the price is under 100 and PriceRange.Expensive otherwise.
- GroupComicsByPrice orders the comics by price, then groups them by CalculatePriceRange(comic) and returns a sequence of groups of comics (IEnumerable<IGrouping<PriceRange, Comic>>).
- GetReviews orders the comics by issue number, then does the join you saw earlier in the chapter and returns a sequence of strings like this: MuddyCritic rated #74 'Black Monday' 84.20



Exercise Solution

Use what you've learned about LINQ so far to build a new console app called **JimmyLing** that organizes Jimmy's comic collection. Start by adding a **Critics enum** with two members, **MuddyCritic** and **RottenTornadoes**, and a **PriceRange enum** with two members, **Cheap** and **Expensive**. Then add a **Review class** with three automatic properties: **int Issue**, **Critics Critic**, and **double Score**.

First you add the Critics and PriceRange enums: Then you add the Review class:

```
enum Critics {
    MuddyCritic,
    RottenTornadoes,
}

enum PriceRange {
    Cheap,
    Expensive,
}
```

```
class Review {
    public int Issue { get; set; }
    public Critics Critic { get; set; }
    public double Score { get; set; }
}
```

Once you have those, you can add the static ComicAnalyzer class with a private **PriceRange** method and public **GroupComicsByPrice** and **GetReviews** methods:

```
using System.Collections.Generic;
using System.Linq;

static class ComicAnalyzer
{
    private static PriceRange CalculatePriceRange(Comic comic)
    {
        if (Comic.Prices[comic.Issue] < 100)
            return PriceRange.Cheap;
        else
            return PriceRange.Expensive;
    }

    public static IEnumerable<IGrouping<PriceRange, Comic>> GroupComicsByPrice(
        IEnumerable<Comic> comics, IReadOnlyDictionary<int, decimal> prices)
    {
        IGroupable<IGrouping<PriceRange, Comic>> grouped =
            from comic in comics
            orderby prices[comic.Issue]
            group comic by CalculatePriceRange(comic) into priceGroup
            select priceGroup;

        return grouped;
    }

    public static IEnumerable<string> GetReviews(
        IEnumerable<Comic> comics, IEnumerable<Review> reviews)
    {
        var joined =
            from comic in comics
            orderby comic.Issue
            join review in reviews on comic.Issue equals review.Issue
            select $"{{review.Critic}} rated {{comic.Issue}} {{comic.Name}} {{review.Score:0.00}}";
    }
}
```

Don't forget the using directives.

We've intentionally included a bug in this CalculatePriceRange method, so make sure the code in your method matches this solution.

Can you spot the bug? It's subtle...

The refactoring tools that we showed you earlier in the chapter make it easier to get the return type of the GroupComicsByPrice method right.

We asked you to order the comics by price, then group them. That causes each group to be sorted by price, because the groups are created in order as the group-by clause enumerates the sequence.

This is really similar to the join query that we explained earlier in the chapter.

Did you run into a compiler error about "inconsistent accessibility" telling you the return type is less accessible than the method? That happens when a class is marked public but has members that are internal (the default if you leave off the access modifier). So make sure that none of the classes or enums are marked public.



BULLET POINTS

- The **group...by** clause tells LINQ to group the results together—when you use it, LINQ creates a sequence of group sequences.
- Every group contains members that have one member in common, called the group's **key**. Use the **by** keyword to specify the key for the group. Each group sequence has a **Key** member that contains the group's key.
- Join queries use an **on...equals** clause to tell LINQ how to match the pairs of items.
- Use a **join** clause to tell LINQ to combine two collections into a single query. When you do this, LINQ compares every member of the first collection with every member of the second collection, including the matching pairs in the results.
- When you're doing a join query, you usually want a set of results that includes some members from the first collection and other members from the second collection. The **select** clause lets you build custom results from both of them.
- Use **select new** to construct custom LINQ query results with an anonymous type that includes only the specific properties you want in your result sequence.
- LINQ queries use **deferred evaluation** (sometimes called lazy evaluation), which means they don't run until a statement uses the results of the query.
- Use the new keyword to create an instance of an **anonymous type**, or an object with a well-defined type that doesn't have a name. The members specified in the new statement become automatic properties of the anonymous type.
- Use the **Rename feature** in Visual Studio to easily rename every instance of a variable, field, property, class, or namespace at once.
- Use Visual Studio's Quick Actions menu to change a var declaration to an **explicit type**, or back to a var (or an implicit type) again.
- One of the most common ways that developers refactor code is **extracting methods**. Visual Studio's Extract Method feature makes it very easy to move a block of code into its own method.
- You can **only use the var keyword** when you're declaring a variable. You can't write a method that returns an anonymous type, or which takes one as a parameter, or use one with a field or a property.

Unit tests help you make sure your code works

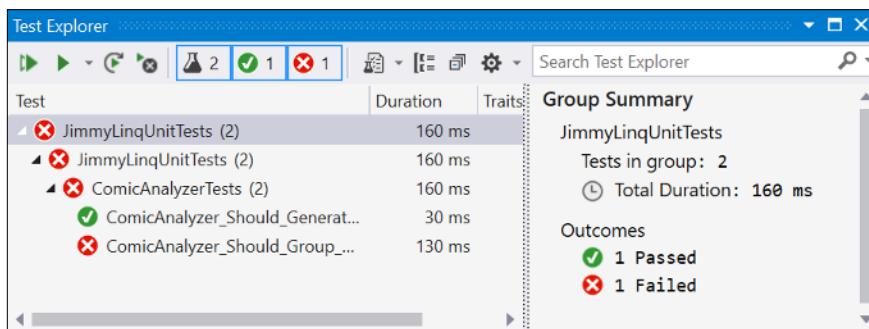
We intentionally left a bug in the code we gave you...but is that the *only* bug in the app? It's easy to write code that doesn't do exactly what you intended it to do. Luckily, there's a way for us to find bugs so we can fix them. **Unit tests** are *automated* tests that help you make sure your code does what it's supposed to do. Each unit test is a method that makes sure that a specific part of the code (the "unit" being tested) works. If the method runs without throwing an exception, it passes. If it throws an exception, it fails. Most large programs have a **suite** of tests that cover most or all of the code.

Visual Studio has built-in unit testing tools to help you write your tests and track which ones pass or fail. The unit tests in this book will use **MSTest**, a unit test framework (which means that it's a set of classes that give you the tools to write unit tests) developed by Microsoft.

Visual Studio also supports unit tests written in NUnit and xUnit, two popular open source unit test frameworks for C# and .NET code.

Visual Studio for Windows has a Test Explorer window

Open the Test Explorer window by choosing **View >> Test Explorer** from the main menu bar. It shows you the unit tests on the left, and the results of the most recent run on the right. The toolbar has buttons to run all tests, run a single test, and repeat the last run.



When you add unit tests to your solution, you can run your tests by clicking the Run All Tests button. You can debug your unit tests on Windows by choosing Tests >> Debug all tests, and on Mac by clicking Debug All Tests in the Unit Tests tool window.



Visual Studio for Mac has the Unit Tests tool window

Open the Unit Test pad by choosing **View >> Tool Windows >> Unit Tests** from the menu bar. It has buttons to run or debug your tests. When you run the unit tests, the IDE displays the results in a

A screenshot of the Visual Studio Unit Tests tool window. On the left, a tree view shows test projects like 'RefactoredJimmyLinq' and 'JimmyLinqUnitTests'. On the right, a 'Test Results' pane displays a summary of test outcomes (Passed: 1, Failed: 1, etc.) and a detailed stack trace for a failed test named 'Assert.Fail failed.' with a link to the specific line of code.

You can use the stack trace to find the specific line where a test failed.

Back in Chapter 3 you learned about prototypes, or early versions of games that you can play, test, learn from, and improve—and you saw how that idea can work with any kind of project, not just games. The same thing applies to testing. Sometimes the idea of testing software can seem a little abstract. Thinking about how game developers test their games can help us get used to the idea, and can make the concept of testing feel more intuitive.



Testing

Game design... and beyond

As soon as you've got a playable prototype of your game, you're ready to think about **video game testing**. Getting people to try out your game and give you feedback on it can make all the difference between a game that everybody loves playing and one that frustrates new users and gives an unsatisfying experience. If you've ever played a game that made you feel lost about what you were supposed to be doing, or one where the puzzles seemed unsolvable, then you know what happens when a game doesn't get enough **play testing**.

There are a few approaches to game testing you'll want to think about as you start designing and building games:

- **Individual play testing:** Ask people you know to play the game, preferably with you watching. The most informal way to do play testing is to just ask a friend to play the game and talk about their experience as they're playing. Having someone narrate what they think the game is asking them to do and what they think about the gameplay can really help you design an experience that's accessible and satisfying to other people. Don't give them very much instruction and pay special attention if your play testers get stuck. That will help you to understand if the point of your game is well understood and will make it easier for you to see if some of the game mechanics you've put in place are not obvious enough to a user. Write down all of the feedback testers give you so you can fix any design problems that they identify.

Getting individual feedback can be done informally or in a more formal setting where you've drawn up a list of tasks you want the user to perform and a **questionnaire** to capture feedback about the game. It should be done frequently as you're adding new features, and you should ask people to play test your game as early in development as possible to make sure you find design problems when they are easiest to fix.

- **Beta programs:** When you are ready to have a larger audience check out the game you can ask a larger group of people to play it before you open it up to the general public. Beta programs are great for finding problems with load and performance in your game. Usually feedback from beta programs is analyzed through game logs. You can log the time it takes for users to accomplish various activities in your game and use those logs to find problems with how the game allocates resources. Sometimes areas that are not well understood by users in the game will show up this way too. Usually, you'll have play testers sign up for a beta test, so you can ask them about their experience afterward and get their help troubleshooting issues you find in the log.
- **Structured quality assurance tests:** Most games have dedicated tests that are run as part of development. Usually these tests are based on an understanding of how the game is supposed to work, and they can be automated or manual. The goal of this kind of testing is to make sure that the product works as intended. When a quality assurance test is run, the idea is to find as many bugs as possible before a user has to deal with them. Those bugs get written down with clear step-by-step instructions so they can be replicated and fixed. Then they get triaged in order of how much they will impact the player's experience with the game and fixed based on that priority. If a game crashes every time a user walks into a room, that bug will probably get fixed before a bug where a weapon isn't rendered correctly in that same room.

Most development teams try to automate as much of their testing as possible, and run those tests before each commit. That way, they know if they inadvertently introduced a bug when they were making a fix or adding a new feature.

Add a unit test project to Jimmy's comic collection app

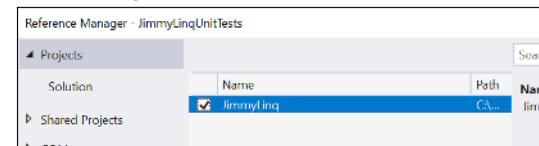
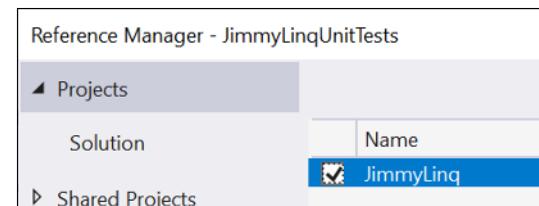
1 Add a new MSTest (.NET Core) project.

Right-click on the solution name in the Solution Explorer, then choose **Add >> New Project...** from the menubar. Make sure it's an **MSTest Test Project (.NET Core)**: on Windows use the "Search for Templates" box to search for MSTest; on macOS select Tests under "Web and Console" to see the project template. Name your project *JimmyLinqUnitTests*.

2 Add a dependency on your existing project.

You'll be building unit tests for the `ComicAnalyzer` class. When you have two different projects in the same solution, they're *independent*—by default, the classes in one project can't use classes in another project—so you'll need to set up a dependency to let your unit tests use `ComicAnalyzer`.

Expand the `JimmyLinqUnitTests` project in the Solution Explorer, then right-click on Dependencies and choose **Add Reference...** from the menu. Check the `JimmyLinq` project that you created for the exercise.



3 Make your ComicAnalyzer class public.

When Visual Studio added the unit test project, it created a class called `UnitTest1`. Edit the `UnitTest1.cs` file and try adding the `using JimmyLinq;` directive inside the namespace:

```
namespace JimmyLinqUnitTests
{
    using JimmyLinq;
    class UnitTest1
    {
        // ...
    }
}
```

This is why we asked you to remove the "public" access modifier from all the classes and enums in the `JimmyLinq` project—so you could use Visual Studio to explore how the "internal" access modifier works.

Hmm, something's wrong—the IDE won't let you add the directive. The reason is that the `JimmyLinq` project has no public classes, enums, or other members. Try modifying the `Critics` enum to make it public (`public enum Critics`), then go back and try adding the `using` directive. Now you can add it! The IDE saw that the `JimmyLinq` namespace has public members, and added it to the pop-up window.

Now change the `ComicAnalyzer` declaration to make it public: `public static class ComicAnalyzer`

Uh-oh—something's wrong. Did you get a bunch of "Inconsistent accessibility" compiler errors?

Inconsistent accessibility: return type 'IEnumerable<IGrouping<PriceRange, Comic>>' is less accessible than method 'ComicAnalyzer.GroupComicsByPrice(IEnumerable<Comic>, IReadOnlyDictionary<int, decimal>)'

The problem is that `ComicAnalyzer` is public, but it exposes members that have no access modifiers, which makes them `internal`—so other projects in the solution can't see them. **Add the public access modifier to *every class and enum* in the `JimmyLinq` project.** Now your solution will build again.

Write your first unit test

The IDE added a class called UnitTest1 to your new MSTest project. Rename the class (and the file) ComicAnalyzerTests. The class contains a test method called TestMethod1. Next, give it a very descriptive name: rename the method ComicAnalyzer_Should_Group_Comics. Here's the code for your unit test class:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace JimmyLinqUnitTests
{
    using JimmyLinq;
    using System.Collections.Generic;
    using System.Linq;

    [TestClass]
    public class ComicAnalyzerTests
    {
        IEnumerable<Comic> testComics = new[]
        {
            new Comic() { Issue = 1, Name = "Issue 1" },
            new Comic() { Issue = 2, Name = "Issue 2" },
            new Comic() { Issue = 3, Name = "Issue 3" },
        };

        [TestMethod]
        public void ComicAnalyzer_Should_Group_Comics()
        {
            var prices = new Dictionary<int, decimal>()
            {
                { 1, 20M },
                { 2, 10M },
                { 3, 1000M },
            };

            var groups = ComicAnalyzer.GroupComicsByPrice(testComics, prices);

            Assert.AreEqual(2, groups.Count());
            Assert.AreEqual(PriceRange.Cheap, groups.First().Key);
            Assert.AreEqual(2, groups.First().First().Issue);
            Assert.AreEqual("Issue 2", groups.First().First().Name);
        }
    }
}
```

Visual Studio added this using directive above the namespace declaration.

Rename the test class (and make sure the file also gets renamed).

Give your test method a descriptive name too.

The groups are sorted in ascending price order, so the first item in the first group should be issue #2.

When you run your unit tests in the IDE, it looks for any class with [TestClass] above it. That's called an attribute. A test class includes test methods, which must be marked with the [TestMethod] attribute.

MSTest unit tests use the Assert class, which has static methods that you can use to check that your code behaves the way you expect it to. This unit test uses the Assert.AreEqual method. It takes two parameters, an expected result (what you think the code should do) and an actual result (what it actually does), and throws an exception if they're not equal.

This test sets up some very limited test data: a sequence of three comics and a dictionary with three prices. Then it calls GroupComicsByPrice and uses Assert.AreEqual to validate that the results match what we expect.

Look closely at the expected results. Our test data has 3 comics: 2 priced under 100 and 1 priced over 100. So it should create two groups, a group with 2 cheap comics followed by a group with 1 expensive comic.

Run your test by choosing **Test >> Run All Tests** (Windows) or **Run >> Run Unit Tests** (Mac) from the menubar. The IDE will pop up a Test Explorer window (Windows) or Test Results panel (Mac) with the test results:

```
Test method JimmyLinqUnitTests.ComicAnalyzerTests.ComicAnalyzer_Should_Group_Comics threw exception:
System.Collections.Generic.KeyNotFoundException: The given key '2' was not present in the dictionary.
```

This is the result of a **failed unit test**. Look for the  1 icon in Windows or the **Failed: 1** message at the bottom of the IDE window in Visual Studio for Mac—that's how you see a count of your failed unit tests.

Did you expect that unit test to fail? Can you figure out what went wrong with the test?



Sleuth it out

Unit testing is all about discovering places where your code doesn't act the way you expect and sleuthing out exactly what went wrong. Sherlock Holmes once said, "It is a capital mistake to theorize before one has data." So let's get some data.

Let's start with the assertions

A good place to start is always the assertions that you included in the test, because they tell you what specific code you're testing, and how you expect it to behave. Here are the assertions from your unit test:

```
Assert.AreEqual(2, groups.Count());
Assert.AreEqual(PriceRange.Cheap, groups.First().Key);
Assert.AreEqual(2, groups.First().First().Issue);
Assert.AreEqual("Issue 2", groups.First().First().Name);
```

When you look at the test data that you're feeding into the GroupComicsByPrice method, these assertions look correct. It really should return two groups. The first one should have the key PriceRange.Cheap. The groups are sorted by ascending price, so the first comic in the first group should have Issue = 2 and Name = "Issue 2"—and that's exactly what those assertions are testing. So if there's a problem, it's not here—these assertions really do seem to be correct.

Now let's have a look at the stack trace

You've seen plenty of exceptions by now. Each exception comes with a **stack trace**, or a list of all of the method calls the program made right up to the line of code that threw it. If it's in a method, it shows what line of code called that method, and what line called that one, all the way up to the Main method. **Open the stack trace** for your failed unit test:

- Windows: open the Test Explorer (View >> Test Explorer), click on the test, and scroll down in the Test Detail Summary
- Mac: go to the Test Results panel, expand the test, then expand the Stack Trace section underneath it

The stack trace will start like this (on Mac you'll see fully qualified class names like JimmyLinq.ComicAnalyzer):

```
at Dictionary`2.get_Item(TKey key)
at CalculatePriceRange(Comic comic) in ComicAnalyzer.cs:line 11
at <>c.<GroupComicsByPrice>b_1_1(Comic comic) in ComicAnalyzer.cs:line 22
```

← Click (Windows) or double-click (Mac) in the stack trace to jump to the code.

Stack traces look a little weird at first, but once you get used to them they've got a lot of useful information. Now we know that the test failed because an exception related to dictionary keys was thrown somewhere inside CalculatePriceRange.

Use the debugger to gather clues

Add a **breakpoint** to the **first line** of the CalculatePriceRange method: `if (Comic.Prices[comic.Issue] < 100)`

Then **debug your unit tests**: on Windows choose Test >> Debug All Tests, on Mac open the Unit Tests panel (View >> Tests) and click the Debug All Tests button at the top. Hover over `comic.Issue`—its value is 2. But wait a minute! The `Comic.Prices` dictionary **doesn't have an entry** with the key 2. **No wonder it threw the exception!**

Now we know [how to fix the bug](#):

- Add a second parameter to the CalculatePriceRange method:
`private static PriceRange CalculatePriceRange(Comic comic, IReadOnlyDictionary<int, decimal> prices)`
- Modify the first line to use the new parameter: `if (prices[comic.Issue] < 100)`
- Modify the LINQ query: `group comic by CalculatePriceRange(comic, prices)` into `priceGroup`

Run your test again. This time it passes!

✓ 1 ✘ 0

Passed: 1 Failed: 0

Write a unit test for the GetReviews method

The unit test for the GroupComicsByPrice method used MSTest's static `Assert.AreEqual` method to check expected values against actual ones. The `GetReviews` method *returns a sequence of strings*, not an individual value. We *could* use `Assert.AreEqual` to compare individual elements in that sequence, just like we did with the last two assertions, using LINQ methods like `First` to get specific elements...but that would take a LOT of code.

Luckily, MSTest has a better way to compare collections: the **CollectionAssert class** has static methods for comparing expected versus actual collection results. So if you have a collection with expected results and a collection with actual results, you can compare them like this:

```
CollectionAssert.AreEqual(expectedResults, actualResults);
```

If the expected and actual results don't match the test will fail. Go ahead and **add this test** to validate the `ComicAnalyzer.GetReviews` method:

```
[TestMethod]
public void ComicAnalyzer_Should_Generate_A_List_Of_Reviews()
{
    var testReviews = new[]
    {
        new Review() { Issue = 1, Critic = Critics.MuddyCritic, Score = 14.5 },
        new Review() { Issue = 1, Critic = Critics.RottenTornadoes, Score = 59.93 },
        new Review() { Issue = 2, Critic = Critics.MuddyCritic, Score = 40.3 },
        new Review() { Issue = 2, Critic = Critics.RottenTornadoes, Score = 95.11 },
    };

    var expectedResults = new[]
    {
        "MuddyCritic rated #1 'Issue 1' 14.50",
        "RottenTornadoes rated #1 'Issue 1' 59.93",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "RottenTornadoes rated #2 'Issue 2' 95.11",
    };

    var actualResults = ComicAnalyzer.GetReviews(testComics, testReviews).ToList();
    CollectionAssert.AreEqual(expectedResults, actualResults);
}
```

Now run your tests again. You should see two unit tests pass.



What happens if you pass a sequence with
duplicate reviews to `ComicAnalyzer.GetReviews`?
What if you pass it a review with a negative score?

Write unit tests to handle edge cases and weird data

In the real world, data is messy. For example, we never really told you exactly what review data is supposed to look like. You've seen review scores between 0 and 100. Did you assume those were the only values allowed? That's definitely the way some review websites in the real world operate. What if we get some weird review scores—like negative ones, or really big ones, or zero? What if we get more than one score from a reviewer for an issue? Even if these things aren't *supposed* to happen, they *might* happen.

We want our code to be **robust**, which means that it handles problems, failures, and especially bad input data well. So let's build a unit test that passes some weird data to GetReviews and makes sure it doesn't break:

```
[TestMethod]
public void ComicAnalyzer_Should_Handle_Weird_Review_Scores()
{
    var testReviews = new[]
    {
        new Review() { Issue = 1, Critic = Critics.MuddyCritic, Score = -12.1212},
        new Review() { Issue = 1, Critic = Critics.RottenTornadoes, Score = 391691234.48931},
        new Review() { Issue = 2, Critic = Critics.RottenTornadoes, Score = 0},
        new Review() { Issue = 2, Critic = Critics.MuddyCritic, Score = 40.3},
        new Review() { Issue = 2, Critic = Critics.MuddyCritic, Score = 40.3},
        new Review() { Issue = 2, Critic = Critics.MuddyCritic, Score = 40.3},
        new Review() { Issue = 2, Critic = Critics.MuddyCritic, Score = 40.3},
    };

    var expectedResults = new[]
    {
        "MuddyCritic rated #1 'Issue 1' -12.12",
        "RottenTornadoes rated #1 'Issue 1' 391691234.49",
        "RottenTornadoes rated #2 'Issue 2' 0.00",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "MuddyCritic rated #2 'Issue 2' 40.30",
    };

    var actualResults = ComicAnalyzer.GetReviews(testComics, testReviews).ToList();
    CollectionAssert.AreEqual(expectedResults, actualResults);
}
```

Can our code handle negative numbers?
Really big numbers? Zero? These are great cases for a unit test to check.

What if we got exactly the same review from the same critic several times in a row? It seems obvious how the code should handle it, but that doesn't mean the code actually does.

The GetReviews method returns a sequence of strings and truncates the score to two decimal places.

Always take time to write unit tests for edge cases and weird data—think of them as “must-have” and not “nice-to-have” tests. The point of unit testing is to cast the widest possible net to catch bugs, and these kinds of tests are really effective for that.

It's really important to add unit tests that handle edge cases and weird data. They can help you spot problems in your code that you wouldn't find otherwise.

ro-bust, adjective.
sturdy in construction. *The bridge's robust design allowed it to handle hurricane-force winds by flexing without breaking.*

there are no Dumb Questions

Q: Why are they called *unit* tests?

A: The term “unit test” is a generic term that applies to many different languages, not just C#. It comes from the idea that your code is divided into discrete units, or small building blocks. Different languages have different units; in C# the basic unit of code is a class.

So from that perspective, the name “unit test” makes sense: you write tests for the units of code, or in our case, on a class-by-class basis.

Q: I created two projects in a single solution. How does that work, exactly?

A: When you start a new C# project in Visual Studio, it creates a solution and adds a project to it. All of the solutions that you created so far in the book had a single project—until the unit test project. A solution can actually contain many projects. We used a separate project to keep the unit tests separate from the code that they’re testing. You can also add multiple Console App, WPF, or ASP.NET projects—a solution can contain combinations of different project types.

Q: If a solution has multiple Console App, WPF, or ASP.NET projects, how does the IDE know which one to run?

A: Look at the Solution Explorer (or the Solution pad in VS for Mac)—one of the project names is boldfaced. The IDE calls that its **Startup Project**. You can right-click on any project in the solution and tell the IDE to use it as the startup project instead. Then the next time you press the Run button in the toolbar, that project will start.

Q: Can you explain how the internal access modifier works again?

A: When you mark a class or interface internal, that means it can only be accessed by code inside that project. If you don’t use an access modifier at all, a class or interface defaults to internal. That’s why you had to make sure your classes were marked public—otherwise, the unit tests wouldn’t be able to see them. Also, *be careful with access modifiers*: while classes and interfaces default to internal if you leave off the access modifier, class members like methods, fields, and properties default to private.

Q: If my unit tests are in a separate project from the code that they’re testing, how can they access private methods?

A: They don’t. Unit tests access whatever part of the unit is visible to the rest of the code—so for your C# classes, that means the public methods and fields—and use them to make sure the unit works. Unit tests are typically supposed to be **black-box tests**, which means that they only check the methods that they can see (as opposed to clear-box tests, where you can “see” the internals of the thing that you’re testing).

Q: Do all of my tests need to pass? Is it OK if some of them fail?

A: Yes, all of your tests need to pass. And no, it’s not OK if they fail. Think of it this way—is it OK if some of your code doesn’t work? Of course not! So if your test fails, then either the code is has a bug or the test does. Either way, you should fix it so it passes.

WRITING TESTS SEEMS LIKE A LOT OF EXTRA WORK. ISN’T IT FASTER TO JUST WRITE CODE AND SKIP THE UNIT TESTS?

Your projects actually go faster when you write unit tests.

We’re serious! It may seem counterintuitive that it takes *less time* to write *more code*, but if you’re in the habit of writing unit tests, your projects go a lot more smoothly because you find and fix bugs early. You’ve written a lot of code so far in the first eight and a half chapters of this book, which means you’ve almost certainly had to track down and fix bugs in your code. When you fixed those bugs, did you have to fix other code in your project too? When we find an unexpected bug, we often have to stop what we’re doing to track it down and fix it, and switching back and forth like that—losing our train of thought, having to interrupt our flow—can really slow things down. Unit tests help you find those bugs early, before they have a chance to interrupt your work.

Still wondering exactly when unit tests should be written? We included a downloadable project at the end of the chapter to help answer that question.



Use the `=>` operator to create lambda expressions

We left you hanging back at the beginning of the chapter. Remember that mysterious line we asked you to add to the Comic class? Here it is again:

```
public override string ToString() => $"{{Name}} (Issue #{Issue})";
```

You've been using that `ToString` method throughout the chapter—you know it works. What would you do if we asked you to rewrite that method the way you've been writing methods so far? You might write something like this:

```
public override string ToString() {  
    return $"{{Name}} (Issue #{Issue})";  
}
```

And you'd basically be right. So what's going on? What exactly is that `=>` operator?

The `=>` operator that you used in the `ToString` method is the **lambda operator**. You can use `=>` to define a **lambda expression**, or an *anonymous function* defined within a single statement. Lambda expressions look like this:

(input-parameters) => expression;

There are two parts to a lambda expression:

- ★ The **input-parameters** part is a list of parameters, just like you'd use when you declare a method. If there's only one parameter, you can leave off the parentheses.
- ★ The **expression** is any C# expression: it can be an interpolated string, a statement that uses an operator, a method call—pretty much anything you would put in a statement.

Lambda expressions may look a little weird at first, but they're just another way of using the **same familiar C# expressions** that you've been using throughout the book—just like the `Comic.ToString` method, which works the same way whether or not you use a lambda expression.



IF THAT `TOSTRING` METHOD WORKS
THE SAME WAY WHETHER OR NOT YOU USE
A LAMBDA EXPRESSION, ISN'T THAT A LOT
LIKE **REFACTORING** ?

Yes! You can use lambda expressions to refactor many methods and properties.

You've written a lot of methods throughout this book that contain just a single statement. You could refactor most of them to use lambda expressions instead. In many cases, that could make your code easier to read and understand. Lambdas give you options—you can decide when using them improves your code.



A lambda test drive

Let's kick the tires on lambda expressions, which give us a whole new way to write methods, including ones that return values or take parameters.

- Create a new console app.** Add this Program class with the Main method:

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

    static double GetRandomDouble(int max)
    {
        return max * random.NextDouble();
    }

    static void PrintValue(double d)
    {
        Console.WriteLine($"The value is {d:0.0000}");
    }

    static void Main(string[] args)
    {
        var value = Program.GetRandomDouble(100);
        Program.PrintValue(value);
    }
}
```



Run it a few times. Each time it prints a different random number, as in: The value is 37.8709

- Refactor the GetRandomDouble and PrintValue methods** using the => operator:

```
static double GetRandomDouble(int max) => max * random.NextDouble();
static void PrintValue(double d) => Console.WriteLine($"The value is {d:0.0000}");
```

Run your program again—it should print a different random number, just like before.

Before we do one more refactoring, **hover over the random field** and look at the IntelliSense pop-up:

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

a (field) static Random Program.random

This isn't a true refactoring, because we changed the behavior of the code, not just its structure.

- Modify the random field** to use a lambda expression:

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

The program still runs the same way. Now **hover over the random field** again:

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

a Random Program.random { get; }

Wait a minute—random isn't a field anymore. Changing it into a lambda turned it into a property! That's because **lambda expressions always work like methods**. So when random was a field, it got instantiated once when the class was constructed. When you changed the = to a => and converted it to a lambda, it became a method—which means **a new instance of Random is created every time the property is accessed**.

These lambdas each take a parameter, just like the methods they're replacing.

Which of the versions of this code do you think is easiest to read?

we really thought we were done with that clown

Refactor a clown with lambdas

Do this!

Back in Chapter 7, you created an IClown interface with two members:

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();
}
```

The IClown interface from Chapter 7 has two members: one property and one method.

And you modified this class to implement that interface:

```
class TallGuy {
    public string Name;
    public int Height;

    public void TalkAboutYourself() {
        Console.WriteLine($"My name is {Name} and I'm {Height} inches tall.");
    }
}
```

IDE Tip: Implement interface

When a class implements an interface, the Quick Actions menu's "Implement interface" option tells the IDE to add any missing interface members.

Let's do that same thing again—but this time we'll use lambdas. **Create a new Console App project** and add the IClown interface and TallGuy class. Then modify TallGuy to implement IClown:

```
class TallGuy : IClown {
```

Now open the Quick Actions menu and choose "**Implement interface.**" The IDE fills in all of the interface members, having them throw NotImplementedExceptions just like it does when you use Generate Method.

```
public string FunnyThingIHave => throw new NotImplementedException();
public void Honk()
{
    throw new NotImplementedException();
}
```

When you asked the IDE to implement the IClown interface for you, it used the `=>` operator to create a lambda to implement the property.

Let's refactor these methods so they do the same thing as before, but now use lambda expressions:

```
public string FunnyThingIHave => "big red shoes";
public void Honk() => Console.WriteLine("Honk honk!");
```

Now add the same Main method you used back in Chapter 7:

```
TallGuy tallGuy = new TallGuy() { Height = 76, Name = "Jimmy" };
tallGuy.TalkAboutYourself();
Console.WriteLine($"The tall guy has {tallGuy.FunnyThingIHave}");
tallGuy.Honk();
```

Run your app. The TallGuy class works just like it did in Chapter 7, but now that we've refactored its members to use lambda expressions it's more compact.

We think the new and improved TallGuy class is easier to read. Do you?

The IDE created a method body when it added the interface members, but you can replace it with a lambda expression. When a class member has a lambda as a body, it's called an expression-bodied member.

That mysterious ToString method at the beginning of the chapter was an expression-bodied member.



Sharpen your pencil



Here are the NectarCollector class from the Beehive Management System project in Chapter 6 and the ScaryScary class from Chapter 7. Your job is to **refactor some of the members of these classes using the lambda operator (=>)**. Write down the refactored methods.

```
class NectarCollector : Bee
{
    public const float NECTAR_COLLECTED_PER_SHIFT = 33.25f;
    public override float CostPerShift { get { return 1.95f; } }
    public NectarCollector() : base("Nectar Collector") { }

    protected override void DoJob()
    {
        HoneyVault.CollectNectar(NECTAR_COLLECTED_PER_SHIFT);
    }
}
```

Refactor the CostPerShift property as a lambda:

```
class ScaryScary : FunnyFunny, IScaryClown {
    private int scaryThingCount;
    public ScaryScary(string funnyThing, int scaryThingCount) : base(funnyThing)
    {
        this.scaryThingCount = scaryThingCount;
    }
    public string ScaryThingIHave { get { return $"{scaryThingCount} spiders"; } }
    public void ScareLittleChildren()
    {
        Console.WriteLine($"Boo! Gotcha! Look at my {ScaryThingIHave}");
    }
}
```

Refactor the ScaryThingIHave property as a lambda:

Refactor the ScareLittleChildren method as a lambda:



Here are the NectarCollector class from the Beehive Management System project in Chapter 6 and the ScaryScary class from Chapter 7. Your job is to **refactor some of the members of these classes using the lambda operator ($=>$)**. Write down the refactored methods.

Refactor the CostPerShift property as a lambda:

```
public float CostPerShift { get => 1.95f; }
```

Refactor the ScaryThingIHaves property as a lambda:

```
public string ScaryThingIHaves { get => $"{scaryThingCount} spiders"; }
```

Refactor the ScareLittleChildren method as a lambda:

```
public void ScareLittleChildren() => Console.WriteLine($"Boo! Gotcha! Look at my {ScaryThingIHaves}");
```

there are no Dumb Questions

Q: Go back to the Test Drive, where you modified the random field. You said that wasn't a "true refactoring"—can you explain what you meant by that?

A: Sure. When you refactor code, you're modifying its *structure* without changing its *behavior*. When you converted the PrintValue and GetRandomDouble methods into lambda expressions, they still worked exactly the same way—you changed their structure, but that change didn't affect their behavior.

But when you changed the equals sign (=) to a lambda operator ($=>$) in the random field declaration, you changed the way that it behaves. A field is like a variable—you declare it once, and then you reuse it. So when random was a field, a new Random instance was created as soon as the program started, and a reference to that instance was stored in the random field.

But when you use a lambda expression, you're always creating a method. So when you changed the random field to this:

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

the C# compiler no longer saw a field. Instead, it saw a property. This should make sense—you learned in Chapter 5 about how properties are called like fields, but are really methods.

And you confirmed this when you hovered over the field:

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

Random Program.random { get; }

The IDE is telling you random is a now property by showing you that it has a get accessor { get; }.

You can use the $=>$ operator to create a property with a get accessor that executes a lambda expression.

Use the ?: operator to make your lambdas make choices

What if you want your lambdas to do... more? It would be great if they could make decisions...and that's where the **conditional operator** (which some people call the **ternary operator**) comes in. It works like this:

```
condition ? consequent : alternative;
```

which may look a little weird at first, so let's have a look at an example. First of all, the ?: operator isn't unique to lambdas—you can use it anywhere. Take this **if** statement from the AbilityScoreCalculator class in Chapter 4:

```
if (added < Minimum)
    Score = Minimum;
else
    Score = added;
```

The ?: expression checks the condition (added < Minimum).

If it's true, the expression returns the value Minimum.

Otherwise it returns added.

We can refactor it using the ?: operator like this: **Score = (added < Minimum) ? Minimum : added;**

Notice how we set Score equal to the results of the ?: expression. The ?: expression **returns a value**: it checks the *condition* (added < Minimum), and then it either returns the *consequent* (Minimum) or the *alternative* (added).

When you have a method that looks like that **if/else** statement, you can **use ?: to refactor it as a lambda**. For example, take this method from the PaintballGun class in Chapter 5:

```
public void Reload()
{
    if (balls > MAGAZINE_SIZE)
        BallsLoaded = MAGAZINE_SIZE;
    else
        BallsLoaded = balls;
}
```

This "if" conditional (balls > MAGAZINE_SIZE) executes the then-statement (BallsLoaded = MAGAZINE_SIZE) if it's true, or the else-statement (BallsLoaded = balls) if it's not.

We converted the "if" to an expression-bodied member that uses ?:—both the consequent and alternative both return a value—and used that ✓ to set the BallsLoaded property.

Let's rewrite that as a more concise lambda expression:

```
public void Reload() => BallsLoaded = balls > MAGAZINE_SIZE ? MAGAZINE_SIZE : balls;
```

Notice the slight change—in the **if/else** version, the BallsLoaded property was set inside the then- and else-statements. We changed this to use a conditional operator that checked balls against MAGAZINE_SIZE and returned the correct value, and used that return value to set the BallsLoaded property.



There's an easy way to remember how the conditional operator works.

A lot of people have trouble remembering the order of the question mark and colon in the ?: ternary operator. Luckily, there's an easy way to remember it.

The conditional operator is like asking a question, and you always ask a question *before* you find out the answer. So just ask yourself:

is this condition true ? yes : no

and you'll know that the ? appears before the : in your expression.

And here's a fun fact—we learned this tip from Microsoft's great documentation page for the ?: operator: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-operator>.

Lambda expressions and LINQ

Add this small LINQ query to any C# app, then hover over the `select` keyword in the code:

```
var array = new[] { 1, 2, 3, 4 };
var result = from i in array select i * 2;
```

This looks like a method declaration, just like when you hover over any other method.

ⓘ (extension) `IEnumerable<int>.Select<int, int>(Func<int, int> selector)`
Projects each element of a sequence into a new form.

Returns:

An `IEnumerable<out T>` whose elements are the result of invoking the transform function on each element of source.

Exceptions:

`ArgumentNullException`

The IDE pops up a tooltip window just like it does **when you hover over a method**. Let's take a closer look at the first line, which shows the method declaration:

`IEnumerable<int>.Select<int, int>(Func<int, int> selector)`

We can learn a few things from that method declaration:

- ★ The `IEnumerable<int>.Select` method returns an `IEnumerable<int>`.
- ★ It takes a single parameter of type `Func<int, int>`.

The `IEnumerable<int>.Select` method takes a single parameter whose type is `Func<int, int>`, which means you can use a lambda that takes an `int` parameter and returns an `int`.

Use lambda expressions with methods that take a Func parameter

When a method takes a `Func<int, int>` parameter, you can **call it with a lambda expression** that takes an `int` parameter and returns an `int` value. So you could refactor the `select` query like this:

```
var array = new[] { 1, 2, 3, 4 };
var result = array.Select(i => i * 2);
```

Go ahead—try that yourself in a console app. Add a `foreach` statement to print the output:

```
foreach (var i in result) Console.WriteLine(i);
```

When you print the results of the refactored query, you'll get the sequence `{ 2, 4, 6, 8 }`—exactly the same result as you got with the LINQ query syntax before you refactored it.



Any time you see a `Func` in a LINQ method, it just means you can use a lambda.

You'll learn a lot more about `Func` in the **downloadable chapter on events and delegates**, which is available as a PDF on our GitHub page. For now, when you use a LINQ method parameter with the type `Func<TSource, TResult>` it means you can call the method by passing it a lambda with a parameter of type `TSource` that returns type `TResult`.

LINQ queries can be written as chained LINQ methods

Take this LINQ query from earlier and **add it to an app** so we can explore a few more LINQ methods:

```
int[] values = new int[] { 0, 12, 44, 36, 92, 54, 13, 8 };
IQueryable<int> result =
    from v in values
    where v < 37
    orderby -v
    select v;
```

A sequence of ints has a LINQ method called `OrderBy` with a lambda that takes an int and returns an int. It works just like the comparer methods we saw in Chapter 8.

The OrderBy LINQ method sorts a sequence

Hover over the `orderby` keyword and take a look at its parameter:

 (extension) `IOrderedEnumerable<int> I Enumerable<int>.OrderBy<int, int>(Func<int, int> keySelector)`
 Sorts the elements of a sequence in ascending order according to a key.
 Returns:
 An `IOrderedEnumerable<out TElement>` whose elements are sorted according to a key.

When you use an `orderby` clause in a LINQ query, it calls a LINQ `OrderBy` method that sorts the sequence. In this case, we can pass it a lambda expression with an int parameter that **returns the sort key**, or any value (which must implement `IComparer`) that it can use to sort the results.

The LINQ `Where` method uses a lambda that takes a member of a sequence and returns true if it should be kept, or false if it should be removed.

The Where LINQ method pulls out a subset of a sequence

Now hover over the `where` keyword in the LINQ query:

 (extension) `IEnumerable<int> I Enumerable<int>.Where<int>(Func<int, bool> predicate)`
 Filters a sequence of values based on a predicate.
 Returns:
 An `IEnumerable<out T>` that contains elements from the input sequence that satisfy the condition.

The `where` clause in a LINQ query calls a LINQ `Where` method that can use a lambda that returns a Boolean. **The Where method calls that lambda for each element in the sequence.** If the lambda returns true, the element is included in the results. If the lambda returns false, the element is removed.

Here's a lambda challenge for you! Can you figure out how to refactor this LINQ query into a set of chained LINQ methods? Start with `result`, then chain the `Where` and `OrderBy` methods to produce the same sequence.

```
IEnumerable<int> result =
    from v in values
    where v < 37
    orderby -v
    select v;
```





LINQ Methods Up Close

LINQ queries can be rewritten as a series of chained LINQ methods—and many of those methods can use lambda expressions to determine the sequence that they produce.

Here's the miniexercise solution—the LINQ query could be refactored like this:

```
var result = values.Where(v => v < 37).OrderBy(v => -v);
```



Let's take a closer look at how the LINQ query is turned into chained methods:

```
IEnumerable<int> result = ——Use var to declare the variable————— var result =
    from v in values ———Start with the values sequence—————values
        where v < 37—Call Where with a lambda that includes values under 37————.Where(v => v < 37)
            orderby -v ——Call OrderBy with a lambda that negates the value————.OrderBy(v => -v);
                select v; No need for a .Select method because the select clause doesn't modify the value
```

Use the OrderByDescending method where you'd use the descending keyword in a LINQ query

Remember how you used the `descending` keyword to change the `orderby` clause in the query? There's an equivalent LINQ method, `OrderByDescending`, that does exactly the same thing:

```
var result = values.Where(v => v < 37).OrderByDescending(v => v);
```

Notice how we're using the lambda expression `v => v`—that's a lambda that will always return whatever is passed to it (sometimes referred to as an *identity function*). So, `OrderByDescending(v => v)` reverses a sequence.

Use the GroupBy method to create group queries from chained methods

We saw this group query earlier in the chapter:

```
var grouped =
    from card in deck
    group card by card.Suit into suitGroup
    orderby suitGroup.Key descending
    select suitGroup;
```

Should you use LINQ declarative query syntax or chained methods? They both accomplish the same thing. Sometimes one way leads to clearer code, sometimes the other way does, so it's valuable to know how to use both.

Hover over `group` and you'll see that it calls the LINQ `GroupBy` method, which returns the same type we saw earlier in the chapter. You can use a lambda to group by the card's suit: `card => card.Suit`

(extension) `IEnumerable<IGrouping<Suits, Card>> IEnumerable<Card>
 .GroupBy<Card, Suits>(Func<Card, Suits> keySelector)`
Groups the elements of a sequence according to a specified key selector function.

And then another lambda to order the groups by key: `group => group.Key`

Here's the LINQ query refactored into chained `GroupBy` and `OrderByDescending` methods:

```
var grouped =
    deck.GroupBy(card => card.Suit)
    .OrderByDescending(group => group.Key);
```

Try going back to the app earlier in the chapter where you used that query and replace it with the chained methods. You'll see exactly the same output. It's up to you to decide which version of your code is clearer and easier to read.

Use the => operator to create switch expressions

You've been using `switch` statements since Chapter 6 to check a variable against several options. It's a really useful tool... but have you noticed its limitations? For example, try adding a case that tests against a variable:

```
case myVariable:
```

You'll get a C# compiler error: *A constant value is expected.* That's because you can only use constant values—like literals and variables defined with the `const` keyword—in the `switch` statements that you've been using.

But that all changes with the `=>` operator, which lets you create **switch expressions**. They're similar to the `switch` statements that you've been using, but they're *expressions* that return a value. A switch expression starts with a value to check and the `switch` keyword followed by a series of *switch arms* in curly brackets separated by commas. Each switch arm uses the `=>` operator to check the value against an expression. If the first arm doesn't match, it moves on to the next one, returning the value for the matching arm.

```
var returnValue = valueToCheck switch
{
    pattern1 => returnValue1,
    pattern2 => returnValue2,
    ...
    _ => defaultReturnValue,
}
```

A switch expression starts with a value to check followed by the `switch` keyword.

The body of the switch expression is a series of switch arms that use the `=>` operator to check `valueToCheck` and return a value if it matches a pattern.

Switch expressions must be exhaustive, which means their patterns must match every possible value. The `_` pattern will match any value that hasn't been matched by any other arm.

Let's say you're working on a card game that needs to assign a certain score based on suit, where spades are worth 6, hearts are worth 4, and other cards are worth 2. You could write a `switch` statement like this:

```
var score = 0;
switch (card.Suit)
{
    case Suits.Spades:
        score = 6;
        break;
    case Suits.Hearts:
        score = 4;
        break;
    default:
        score = 2;
        break;
}
```

Every case in this switch statement sets the `score` variable. That makes it a great candidate for switch expressions.

The whole goal of this `switch` statement is to use the cases to set the `score` variable—and a lot of our `switch` statements work that way. We can use the `=>` operator to create a switch expression that does the same thing:

```
var score = card.Suit switch
{
    Suits.Spades => 6,
    Suits.Hearts => 4,
    _ => 2,
};
```

This switch expression checks `card.Suit`—if it's equal to `Suits.Spades` the expression returns 6, if it's equal to `Suits.Hearts` it returns 4, and for any other value it returns 2.



Sharpen your pencil

This console app uses the Suit, Value, and Deck classes that you used earlier in the chapter and writes six lines to the console. Your job is to **write down the output of the program**. When you're done, add the program to a console app to check your answer.

```
class Program
{
    static string Output(Suits suit, int number) =>
        $"Suit is {suit} and number is {number}";

    static void Main(string[] args)
    {
        var deck = new Deck();
        var processedCards = deck
            .Take(3)
            .Concat(deck.TakeLast(3))
            .OrderByDescending(card => card)
            .Select(card => card.Value switch
            {
                Values.King => Output(card.Suit, 7),
                Values.Ace => $"It's an ace! {card.Suit}",
                Values.Jack => Output((Suits)card.Suit - 1, 9),
                Values.Two => Output(card.Suit, 18),
                _ => card.ToString(),
            });
        foreach(var output in processedCards)
        {
            Console.WriteLine(output);
        }
    }
}
```

These LINQ methods are just like the ones you saw at the beginning of the chapter.

This lambda expression takes two parameters, a Suit and an int, and returns an interpolated string.

You can use OrderByDescending because you made your Card class implement IComparable<Card> earlier in the chapter.

The Select method uses a switch expression to check the card's value and generate a string.

Write down the output of the program. We won't give you a solution—add the code to a console app for the answer.

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

This is a serious lambda challenge! There's a lot going on here—you're using lambda expressions, a switch expression, LINQ methods, enum casting, chained methods, and more. Really take your time and figure out how this code works before writing down your solution, and then run the program. If your solution didn't match the output, it's a great opportunity to sleuth out why it worked differently than you expected.



Exercise

Use everything you've learned about lambda expressions, switch expressions, and LINQ methods to refactor the ComicAnalyzer class and Main method, using your unit tests to make sure your code still works exactly the way you expect it to.

Replace the LINQ queries in ComicAnalyzer

ComicAnalyzer has two LINQ queries:

- The GroupComicsByPrice method has a LINQ query that uses the group keyword to group comics by price.
- The GetReviews method has a LINQ query that uses the join keyword to join a sequence of Comic objects to a dictionary of issue prices.

Modify the LINQ queries in these methods to use the LINQ OrderBy, GroupBy, Select, and Join methods. There's one catch: **we haven't shown you the Join method yet!** But we showed you examples of how to use the IDE to explore LINQ methods. The Join method is a little more complex—but we'll help you break it down. It takes four parameters:

```
sequence.Join(sequence to join,
              Lambda expression for the 'on' part of the join,
              Lambda expression for the 'equals' part of the join, This lambda is passed every
              Lambda expression that takes two parameters and returns the 'select' output); pair of items from the two sequences being joined.
```

Look closely at the "on" and "equals" parts of the LINQ query to come up with the first two lambdas. The Join will be the last method in the chain. Here's a *hint*—the last parameter's lambda starts like this: (comic, review) =>

Once both unit tests pass, then you're done refactoring the ComicAnalyzer class.

Replace the switch statement in the Main method with a switch expression

The Main method has a switch statement that calls private methods and assigns their return values to the done variable. Replace this with a switch expression with three switch arms. You can test it by running the app—if you can press the correct key and see the right output, you're done.

This exercise is about learning to use unit tests to safely refactor your code

Refactoring can be messy, even nerve-wracking business. You're taking code that works and making changes to improve its structure, readability, reusability. When you're modifying your code, it's really easy to accidentally mess it up so it doesn't quite work right anymore—and sometimes the bugs you introduce can be subtle and difficult to track down, or even detect. That's where unit tests can help. One of the most important ways that developers use unit tests is to make refactoring a much safer activity. Here's how it works:

- Before you start refactoring, write tests that make sure your code works—like the tests you added earlier in the chapter to validate the ComicAnalyzer class.
- When you're refactoring a class, just run the tests for that class as you make the changes. That gives you a much shorter feedback loop for developing—you can do your normal debugging, but it goes a lot faster because you're executing the code in the class directly (rather than starting up your program and using its user interface to run the code that uses the class).
- When you're refactoring a method, you can even start by just running the specific test or tests that execute that method. Then, once it works, you can run the entire suite to make sure you didn't break anything else.
- If a test fails, don't feel bad—that's actually good news! It's telling you something's broken, and now you can fix it.

Behind the Scenes





Exercise Solution

Use everything you've learned about lambda expressions, switch expressions, and LINQ methods to refactor the ComicAnalyzer class and Main method, using your unit tests to make sure your code still works exactly the way you expect it to.

Here are the refactored GroupComicsByPrice and GetReviews methods from the ComicAnalyzer class:

```
public static IEnumerable<IGrouping<PriceRange, Comic>> GroupComicsByPrice(
    IEnumerable<Comic> comics, IReadOnlyDictionary<int, decimal> prices)
{
    var grouped =
        comics
            .OrderBy(comic => prices[comic.Issue])
            .GroupBy(comic => CalculatePriceRange(comic, prices));
    return grouped;
}

public static IEnumerable<string> GetReviews(
    IEnumerable<Comic> comics, IEnumerable<Review> reviews)
{
    var joined =
        comics
            .OrderBy(comic => comic.Issue)
            .Join(
                reviews,
                comic => comic.Issue,
                review => review.Issue,
                (comic, review) =>
                    $"{review.Critic} rated #{comic.Issue} '{comic.Name}' {review.Score:0.00}");
    return joined;
}
```

The join query starts "join reviews" so the first argument passed to the Join method is reviews.

Compare the OrderBy and GroupBy lambdas with the orderby and group...by clauses in the LINQ query. They're almost identical.

Compare the middle two arguments passed to the Join method against the "on" and "equals" parts of the join query: on comic.Issue equals review.Issue.

This last lambda is called with every matched comic and review pair from the two joined sequences and returns the string to include in the output.

Here's the refactored Main method with a switch expression instead of a switch statement:

```
static void Main(string[] args)
{
    var done = false;
    while (!done)
    {
        Console.WriteLine(
            "\nPress G to group comics by price, R to get reviews, any other key to quit\n");
        done = Console.ReadKey(true).KeyChar.ToString().ToUpper() switch
        {
            "G" => GroupComicsByPrice(),
            "R" => GetReviews(),
            _ => true,
        };
    }
}
```

The switch expression is a lot more compact than the equivalent switch statement. Not all switch statements can be refactored into switch expressions—this one could because each of its cases sets the same variable (done) to a value.

Explore the `Enumerable` class

We've been using sequences for a while. We know they work with `foreach` loops and LINQ. But what, exactly, makes sequences tick? Let's take a deeper dive to find out. We'll start with the **Enumerable class**—specifically, with its three static methods, `Range`, `Empty`, and `Repeat`. You already saw the `Enumerable.Range` method earlier in the chapter. Let's use the IDE to discover how the other two methods work. Type `Enumerable.` and then hover over `Range`, `Empty`, and `Repeat` in the IntelliSense pop-up to see their declarations and comments.

The screenshot shows the IntelliSense pop-up for the `Enumerable` class. It lists three static methods:

- `Range`: Returns an empty `IEnumerable<T>` that has the specified type argument.
- `Empty`: Generates a sequence that contains one repeated value.
- `Repeat`: Returns an `IEnumerable<TResult>` `Enumerable.Repeat<TResult>(TResult element, int count)`.

`Enumerable.Empty` creates an empty sequence of any type

Sometimes you need to pass an empty sequence to a method that takes an `IEnumerable<T>` (for example, in a unit test). The **Enumerable.Empty method** comes in handy in these cases:

```
var emptyInts = Enumerable.Empty<int>(); // an empty sequence of ints
var emptyComics = Enumerable.Empty<Comic>(); // an empty sequence of Comic references
```

`Enumerable.Repeat` repeats a value a number of times

Let's say you need a sequence of 100 3s, or 12 "yes" strings, or 83 identical anonymous objects. You'd be surprised at how often that happens! You can use the **Enumerable.Repeat method** for this—it returns a sequence of repeated values:

```
var oneHundredThrees = Enumerable.Repeat(3, 100);
var twelveYesStrings = Enumerable.Repeat("yes", 12);
var eightyThreeObjects = Enumerable.Repeat(
    new { cost = 12.94M, sign = "ONE WAY", isTall = false }, 83);
```

So what exactly is an `IEnumerable<T>`?

We've been using `IEnumerable<T>` for a while now. We haven't really answered the question of what an enumerable sequence *actually is*. A really effective way to understand something is to build it ourselves, so let's finish the chapter by building some sequences from the ground up.



If you had to design an `IEnumerable<T>` interface yourself, what members would you put in it?

Create an enumerable sequence by hand

Let's say we have some sports:

```
enum Sport { Football, Baseball, Basketball, Hockey, Boxing, Rugby, Fencing }
```

Obviously, we could create a new `List<Sport>` and use a collection initializer to populate it. But for the sake of exploring how sequences work, we'll build one manually. Let's create a new class called `ManualSportSequence` and make it implement the `IEnumerable<Sport>` interface. It just has two members that return an `IEnumerator`:

```
class ManualSportSequence : IEnumerable<Sport> {
    public IEnumerator<Sport> GetEnumerator() {
        return new ManualSportEnumerator();
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}
```

When we used the "Implement interface" Quick Actions menu item, it used these fully qualified class names for `IEnumerator` and `IEnumerable`.

OK, so what's an `IEnumerator`? It's an interface that lets you enumerate a sequence, moving through each item in the sequence one after another. It has a property, `Current`, which returns the current item being enumerated. Its `MoveNext` method moves to the next element in the sequence, returning false if the sequence has run out. After `MoveNext` is called, `Current` returns that next element. Finally, the `Reset` method resets the sequence back to the beginning. Once you have those methods, you have an enumerable sequence.

IEnumerator<T>
Current
MoveNext
Reset
Dispose

So let's implement an `IEnumerator<Sport>`:

```
using System.Collections.Generic;
class ManualSportEnumerator : IEnumerator<Sport> {
    int current = -1;

    public Sport Current { get { return (Sport)current; } }

    public void Dispose() { return; } // You'll meet the Dispose method in Chapter 10

    object System.Collections.IEnumerator.Current { get { return Current; } }

    public bool MoveNext() {
        var maxEnumValue = Enum.GetValues(typeof(Sport)).Length;
        if ((int)current >= maxEnumValue - 1)
            return false;
        current++;
        return true;
    }

    public void Reset() { current = 0; }
}
```

We also need to implement the `IDisposable` interface, which you'll learn about in the next chapter. It just has one method, `Dispose`.

Our manual sport enumerator takes advantage of casting an int to an enum. It uses the static `Enum.GetValues` method to get the total number of members in the enum, and uses an int to keep track of the index of the current value.

And that's all we need to create our own `IEnumerable`. Go ahead—give it a try. **Create a new console app**, add `ManualSportSequence` and `ManualSportEnumerator`, and then enumerate the sequence in a `foreach` loop:

```
var sports = new ManualSportSequence();
foreach (var sport in sports)
    Console.WriteLine(sport);
```

Use yield return to create your own sequences

C# gives you a much easier way to create enumerable sequences: the **`yield return` statement**. The `yield return` statement is a kind of all-in-one automatic enumerator creator. A good way to understand it is to see an example. Let's use a **multiproject solution**, just to give you a little more practice with that.

Add a new **Console App project to your solution**—this is just like what you did when you added the MSTest project earlier in the chapter, except this time instead of choosing the project type MSTest choose the same Console App project type that you've been using for most of the projects in the book. Then right-click on the project under the solution and **choose “Set as startup project.”** Now when you launch the debugger in the IDE, it will run the new project. You can also right-click on any project in the solution and run or debug it.

Here's the code for the new console app:

```
static IEnumerable<string> SimpleEnumerable() {
    yield return "apples";
    yield return "oranges";
    yield return "bananas";
    yield return "unicorns";
}

static void Main(string[] args) {
    foreach (var s in SimpleEnumerable()) Console.WriteLine(s);
}
```

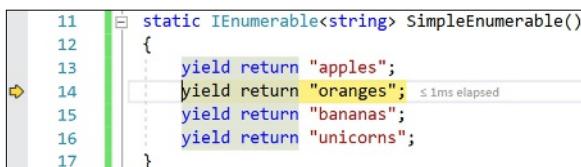
This method returns an `IEnumerable<string>` so every `yield return` returns a string value.

Run the app—it prints four lines: `apples`, `oranges`, `bananas`, and `unicorns`. So how does that work?

Use the debugger to explore `yield return`

Set a breakpoint on the first line of the `Main` method and launch the debugger. Then use **Step Into** (F11 / ⌘I) to debug the code line by line, right into the iterator:

- ★ Step into the code, and keep stepping into it until you reach the first line of the `SimpleEnumerable` method.
- ★ Step into that line again. It acts just like a `return` statement, returning control back to the statement that called it—in this case, back to the `foreach` statement, which calls `Console.WriteLine` to write `apples`.
- ★ Step two more times. Your app will jump back into the `SimpleEnumerable` method, but ***it skips the first statement in the method*** and goes right to the second line:



Every time the `foreach` loop gets an item from the sequence returned by the `SimpleEnumerable` method, it jumps back into the method right after the last `yield return` that was called.

- ★ Keep stepping. The app returns to the `foreach` loop, then back to the ***third line*** of the method, then returns to the `foreach` loop, and goes back to the ***fourth line*** of the method.

So `yield return` makes a method **return an enumerable sequence** by returning the next element in the sequence each time it's called, and keeping track of where it returned from so it can pick up where it left off.

Use **yield return** to refactor **ManualSportSequence**

You can create your own `IEnumerable<T>` by **using `yield return` to implement the `GetEnumerator` method**. For example, here's a `BetterSportSequence` class that does exactly the same thing as `ManualSportSequence` did. This version is much more compact because it uses `yield return` in its `GetEnumerator` implementation:

```
using System.Collections.Generic;
class BetterSportSequence : IEnumerable<Sport> {
    public IEnumerator<Sport> GetEnumerator() {
        int maxEnumValue = Enum.GetValues(typeof(Sport)).Length - 1;
        for (int i = 0; i <= maxEnumValue; i++) {
            yield return (Sport)i;
        }
    }
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}
```

You can use `yield return` to implement the `GetEnumerator` method in `IEnumerable<T>` and create your own enumerable sequence.

Go ahead and **add a new Console App project to your solution**. Add this new `BetterSportSequence` class, and modify the `Main` method to create an instance of it and enumerate the sequence.

Add an indexer to **BetterSportSequence**

You've seen that you can use `yield return` in a method to create an `IEnumerable<T>`. You can also use it to create a class that implements `IEnumerable<T>`. One advantage of creating a separate class for your sequence is that you can add an **indexer**. You've already used indexers—any time you use brackets `[]` to retrieve an object from a list, array, or dictionary (like `myList[3]` or `myDictionary["Steve"]`), you're using an indexer. An indexer is just a method. It looks a lot like a property, except it's got a single named parameter.

The IDE has an **especially useful code snippet** to help you add your indexer. Type **indexer** followed by two tabs, and the IDE will add the skeleton of an indexer for you automatically.

Here's an indexer for the `SportCollection` class:

```
public Sport this[int index] {
    get => (Sport)index;
}
```

Calling the indexer with `[3]` returns the value `Hockey`:

```
var sequence = new BetterSportSequence();
Console.WriteLine(sequence[3]);
```

Take a close look when you use the snippet to create the indexer—it lets you set the type. You can define an indexer that takes different types, including strings and even objects. While our indexer only has a getter, you can also include a setter (just like the ones you've used to set items in a `List`).



Sequences are not collections.

Watch it!

Try creating a class that implements `ICollection<int>` and use the Quick Actions menu to implement its members. You'll see that a collection not only has to implement the `IEnumerable<T>` methods, but it also needs additional properties (including `Count`) and methods (including `Add` and `Clear`). That's how you know a collection does a different job than an enumerable sequence.



Exercise

Create an enumerable class that, when enumerated, returns a sequence of ints that contains all of the powers of 2, starting at 0 and ending with the largest power of 2 that can fit inside an int.

Use `yield return` to create a sequence of powers of 2

Create a class called `PowersOfTwo` that implements `IEnumerable<int>`. It should have a `for` loop that starts at 0 and uses `yield return` to return a sequence that contains each power of 2.

The app should write the following output to the console: 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144 524288 1048576 2097152 4194304 8388608 16777216 33554432 67108864 134217728 268435456 536870912 1073741824

Return the desired sequence of values

You'll use methods from the static `System.Math` class in your app to:

- Compute a specific power of 2: `Math.Pow(power, 2)`
- Find the maximum power of 2 that can fit inside an int: `Math.Round(Math.Log(int.MaxValue, 2))`

there are no Dumb Questions

Q: I think I get what's going on with `yield return`, but can you explain again exactly why it jumps right into the middle of a method?

A: When you use `yield return` to create an enumerable sequence, it does something that you haven't seen anywhere else in C#. Normally when your method hits a `return` statement, it causes your program to execute the statement right after the one that called the method. It does the same thing when it's enumerating a sequence created with `yield return`—with one difference: it remembers the last `yield return` statement it executed in the method. Then when it moves to the next item in the sequence, instead of starting at the beginning of the method your program executes the next statement after the most recent `yield return` that was called. That's why you can build a method that returns an `IEnumerable<T>` with just a series of `yield return` statements.

Q: When I added a class that implemented `IEnumerable<T>`, I had to add a `MoveNext` method and `Current` property. When I used `yield return`, how was I able to implement that interface without having to implement those two members?

A: When the compiler sees a method with a `yield return` statement that returns an `IEnumerable<T>`, it automatically adds the `MoveNext` method and `Current` property. When it executes, the first `yield return` that it encounters causes it to return the first value to the `foreach` loop. When the `foreach` loop continues (by calling the `MoveNext` method), it resumes execution with the statement immediately after the last `yield return` that it executed. Its `MoveNext` method returns `false` when the enumerator is positioned after the last element in the collection. This may be a little hard to follow on paper, but it's much easier to follow if you load it into the debugger—which is why the first thing we had you do was step through a simple sequence that uses `yield return`.



Exercise Solution

```
class PowersOfTwo : IEnumerable<int> {
    public IEnumerator<int> GetEnumerator() {
        var maxPower = Math.Round(Math.Log(int.MaxValue, 2));
        for (int power = 0; power < maxPower; power++)
            yield return (int)Math.Pow(2, power);
    }
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}

class Program {
    static void Main(string[] args) {
        foreach (int i in new PowersOfTwo())
            Console.WriteLine($" {i}");
    }
}
```

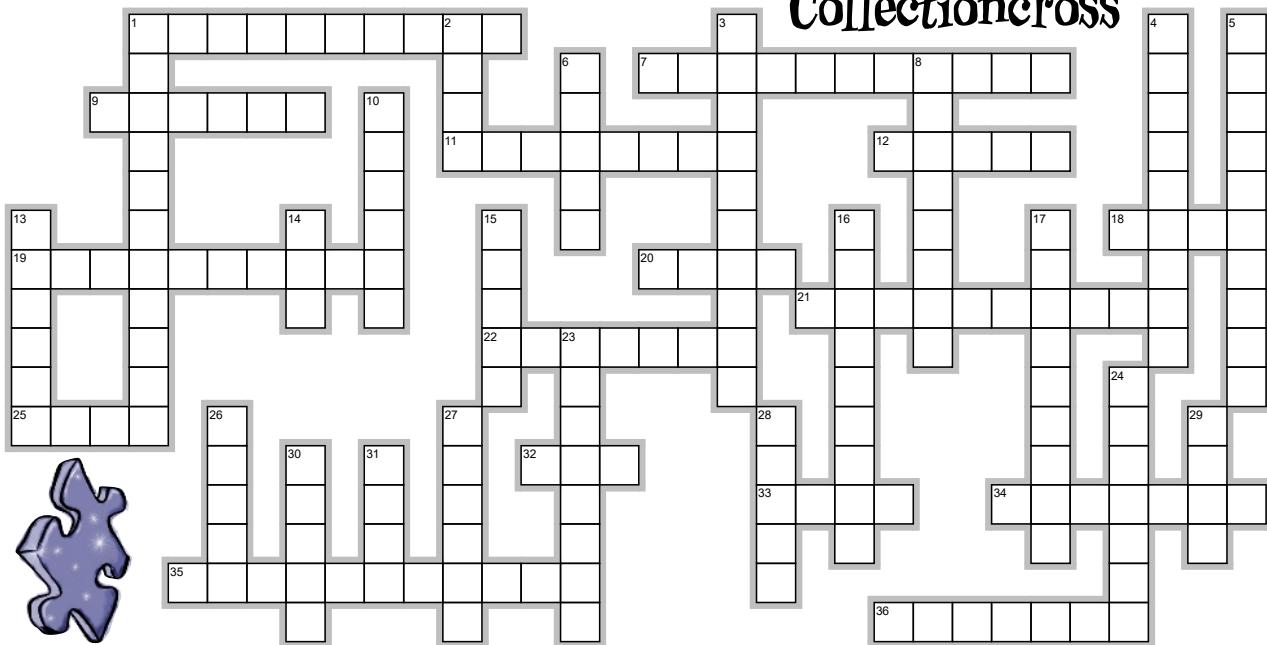
Don't forget your using directives:

```
using System;
using System.Linq;
using System.Collections;
using System.Collections.Generic;
```

BULLET POINTS

- **Unit tests** are automated tests that help you make sure your code does what it's supposed to do, and help you **safely refactor** your code.
- MSTest is a **unit test framework**, or a set of classes that give you the tools to write unit tests. Visual Studio has tools to execute and see the results of unit tests.
- Unit tests use **assertions** to validate specific behaviors.
- The **internal keyword** makes classes in one project accessible to another project in a multiproject build.
- Add unit tests to handle edge cases and weird data to make your code more **robust**.
- Use the lambda operator `=>` to define **lambda expressions**, or anonymous functions defined within a single statement that looks like this: `(input-parameters) => expression;`
- When a class implements an interface, the “**Implement interface**” option in the Quick Actions menu tells the IDE to add any missing interface members.
- `orderby` and `where` clauses in LINQ queries can be rewritten using the **OrderBy** and **Where** LINQ methods.
- You can use the `=>` operator to **turn a field into a property** with a get accessor that executes a lambda expression.
- The **? : operator** (called the conditional or ternary operator) lets you create a single expression that executes an if/else condition.
- LINQ methods that take a **Func<T1, T2> parameter** can be called with a lambda that takes a T1 parameter and returns a T2 value.
- Use the `=>` operator to create **switch expressions**, which are like switch statements that return a value.
- The Enumerable class has static **Range**, **Empty**, and **Repeat methods** to help you create enumerable sequences.
- Use **yield return statements** to create methods that return enumerable sequences.
- When a method executes a `yield return`, it returns the next value in the sequence. The next time the method is called, it **resumes execution** at the next statement after the last `yield return` that was executed.

Collectioncross



EclipseCrossword.com

Across

1. Use the var keyword to declare an _____ typed variable
7. A collection _____ combines the declaration with items to add
9. What you're trying to make your code when you have lots of tests for weird data and edge cases
11. LINQ method to return the last elements in a sequence
12. A last-in, first-out (LIFO) collection
18. LINQ method to return the first elements in a sequence
19. A method that has multiple constructors with different parameters
20. The type of parameter that tells you that you can use a lambda
21. What you take advantage of when you upcast an entire list
22. What you're using when you call myArray[3]
25. What T gets replaced with when you see <T> in a class or interface definition
32. The keyword you use to create an anonymous object
33. A data type that only allows certain values
34. The kind of collection that can store any type
35. The interface that all sequences implement
36. Another name for the ?: conditional operator

Down

1. If you want to sort a List, its members need to implement this
2. A collection class for storing items in order
3. A collection that stores keys and values
4. What you pass to List.Sort to tell it how to sort its items

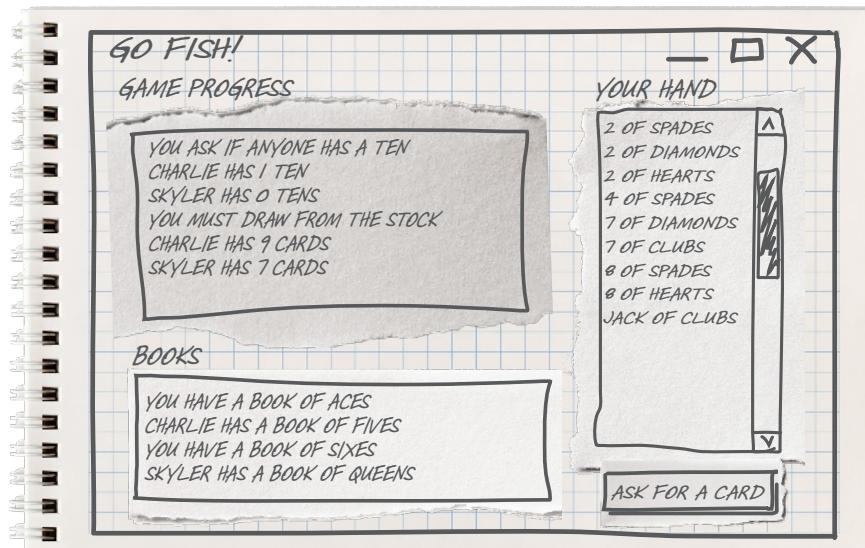
Down

5. What goes in the parentheses: (_____) => expression;
6. You can't use the var keyword to declare one of these
8. The access modifier for a class that can't be accessed by another project in a multiproject solution
10. The kind of expression the => operator creates
13. LINQ method to append the elements from one sequence to the end of another
14. Every collection has this method to put a new element into it
15. What you can do with methods in a class that return the type of that class
16. What kind of type you're looking at when the IDE tells you this: 'a' is a new string Color, int Height
17. An object's namespace followed by a period followed by the class is a fully _____ class name
23. The kind of evaluation that means a LINQ query isn't run until its results are accessed
24. The clause in a LINQ query that sorts the results
26. Type of variable created by the from clause in a LINQ query
27. The Enumerable method that returns a sequence with many copies of the same element
28. The clause in a LINQ query that determines which elements in the input to use
29. A LINQ query that merges data from two sequences
30. A first-in, first-out (FIFO) collection
31. The keyword a switch statement has that a switch expression doesn't



Downloadable exercise: Go Fish

In this next exercise you'll build a Go Fish card game where you play against computer players. Unit testing will play an important part, because you'll be doing **test-driven development**, a technique where you write your unit tests before you write the code that they test.

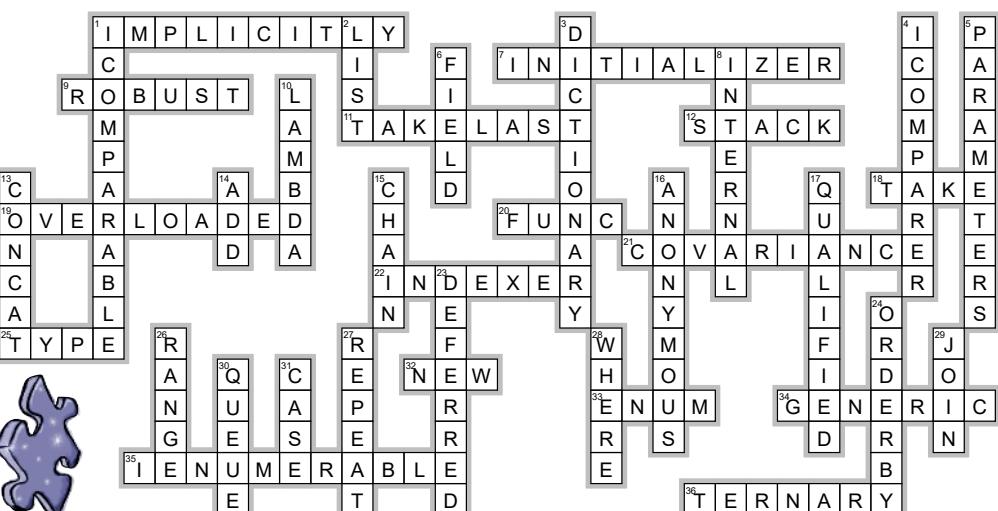


Go to the GitHub page for the book and download the project PDF:
<https://github.com/head-first-csharp/fourth-edition>

Collection crossword solution



EclipseCrossword.com



10 reading and writing files



Save the last byte for me!



Sometimes it pays to be a little persistent.

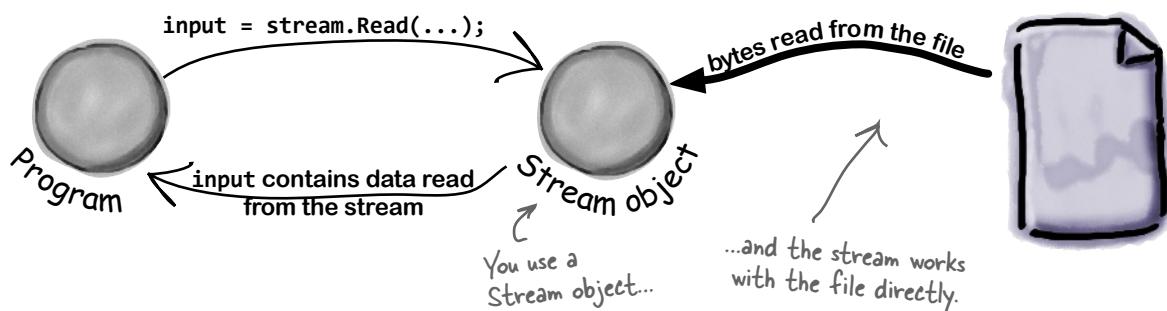
So far, all of your programs have been pretty short-lived. They fire up, run for a while, and shut down. But that's not always enough, especially when you're dealing with important information. You need to be able to **save your work**. In this chapter, we'll look at how to **write data to a file**,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

.NET uses streams to read and write data

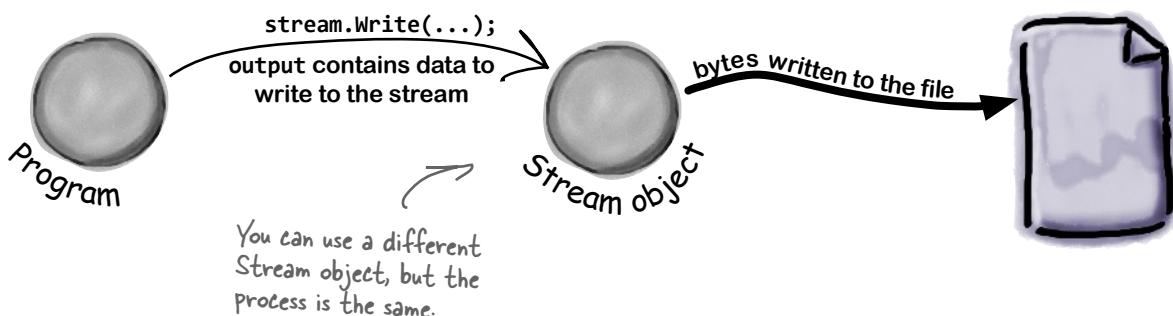
A **stream** is the .NET Framework's way of getting data into and out of your program. Any time your program reads or writes a file, connects to another computer over a network, or generally does anything where it **sends or receives bytes**, you're using streams. Sometimes you're using streams directly, other times indirectly. Even when you're using classes that don't directly expose streams, under the hood they're almost always using streams.

Let's say you have a simple app that needs to read data from a file. A really basic way to do that is to use a Stream object.

Whenever you want to read data from a file or write data to a file, you'll use a Stream object.

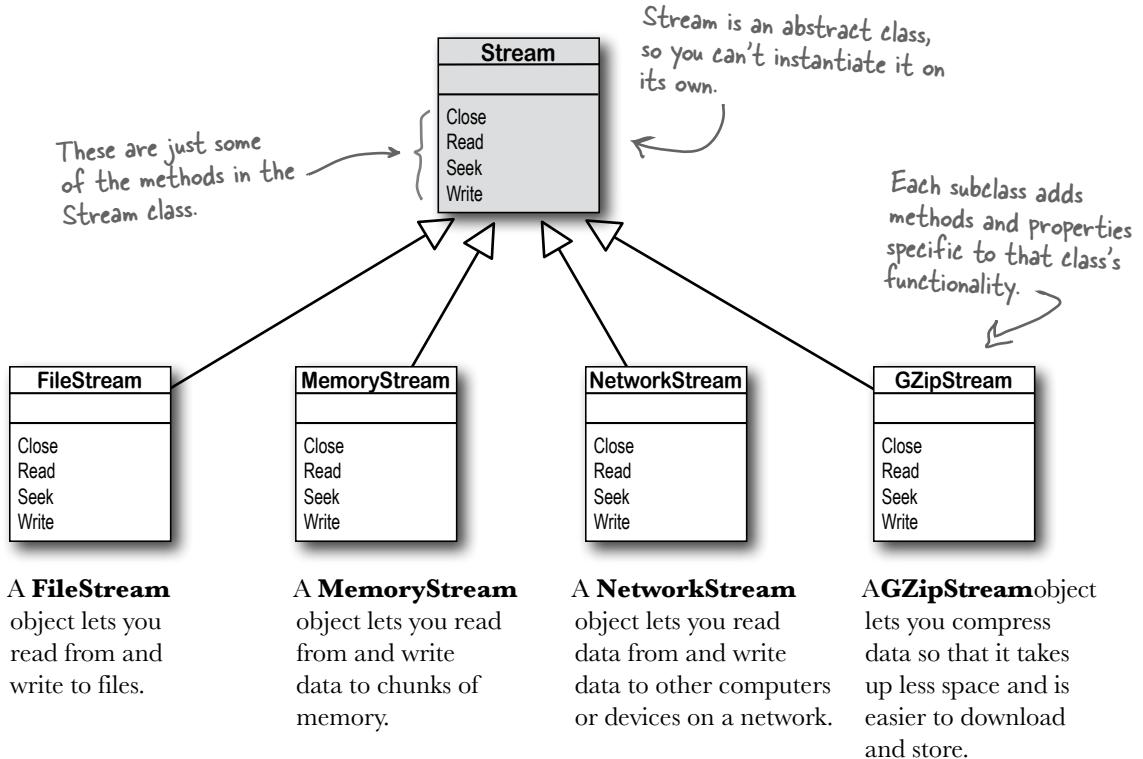


And if your app needs to write data out to the file, it can use another Stream object.



Different streams read and write different things

Every stream is a subclass of the **abstract Stream class**, and there are many subclasses of Stream that do different things. We'll be concentrating on reading and writing regular files, but everything you learn about streams in this chapter can apply to compressed or encrypted files, or network streams that don't use files at all.



Things you can do with a stream:

1 Write to the stream.

You can write your data to a stream through a stream's **Write method**.

2 Read from the stream.

You can use the **Read method** to get data from a file, or a network, or memory, or just about anything else using a stream. You can even read data from **really big** files, even if they're too big to fit into memory.

3 Change your position within the stream.

Most streams support a **Seek method** that lets you find a position within the stream so you can read or insert data at a specific place. However, not every Stream class supports Seek—which makes sense, because you can't always backtrack in some sources of streaming data.

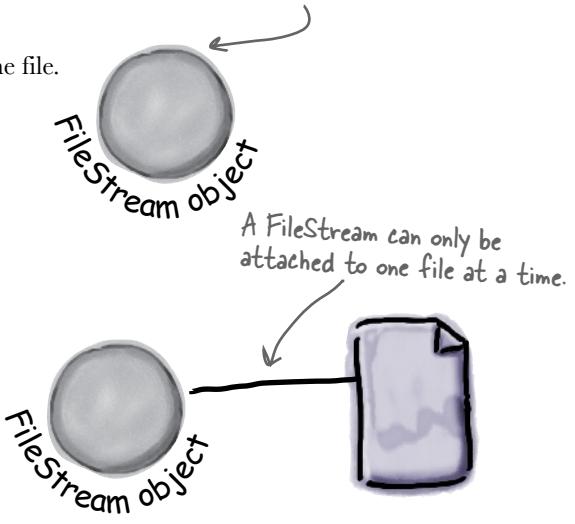
Streams let you read and write data. Use the right kind of stream for the data you're working with.

A FileStream reads and writes bytes in a file

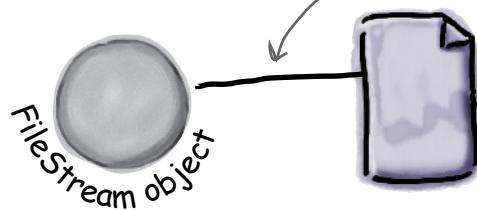
When your program needs to write a few lines of text to a file, there are a lot of things that have to happen:

Make sure you add "using System.IO;" to any program that uses FileStreams.

- 1 Create a new FileStream object and tell it to write to the file.



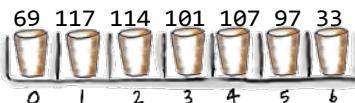
- 2 The FileStream attaches itself to a file.



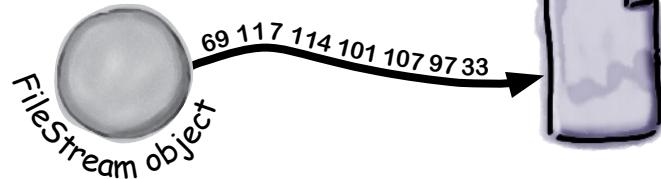
- 3 Streams write bytes to files, so you'll need to convert the string that you want to write to an array of bytes.

This is called encoding, and we'll talk more about it later on...

Eureka!

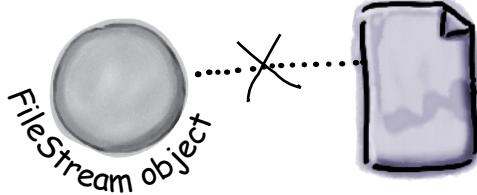


- 4 Call the stream's Write method and pass it the byte array.



- 5 Close the stream so other programs can access the file.

Forgetting to close a stream is a big deal. The file will be locked and other programs won't be able to use it until you close your stream.



Write text to a file in three simple steps

C# comes with a convenient class called **StreamWriter** that simplifies those things for you. All you have to do is create a new StreamWriter object and give it a filename. It **automatically** creates a FileStream and opens the file. Then you can use the StreamWriter's Write and WriteLine methods to write everything to the file you want.

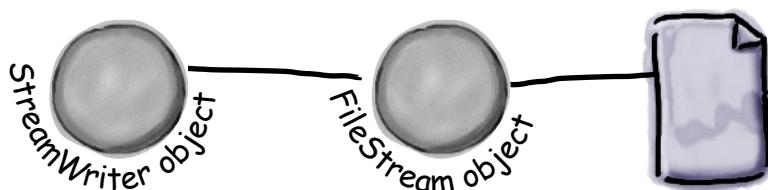
StreamWriter creates and manages a FileStream object for you automatically.

1

Use the StreamWriter's constructor to open or create a file.

You can pass a filename to the StreamWriter's constructor. When you do, the writer automatically opens the file. StreamWriter also has an overloaded constructor that lets you specify its *append* mode: passing it **true** tells it to add data to the end of an existing file (or append), while **false** tells the stream to delete the existing file and create a new file with the same name.

```
var writer = new StreamWriter("toaster oven.txt", true);
```

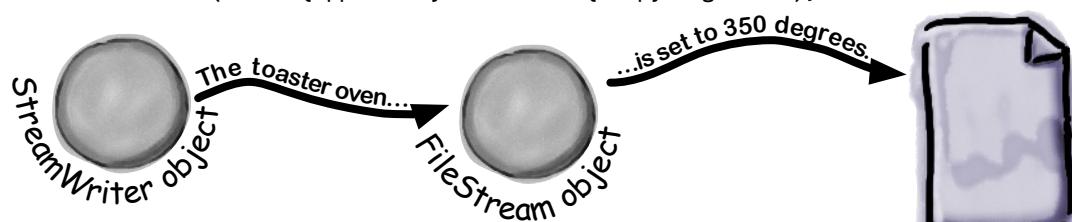


2

Use the Write and WriteLine methods to write to the file.

These methods work just like the ones in the Console class: Write writes text, and WriteLine writes text and adds a line break to the end.

```
writer.WriteLine($"The {appliance} is set to {temp} degrees.");
```



3

Call the Close method to release the file.

If you leave the stream open and attached to a file, then it'll keep the file locked and no other program will be able to use it. So make sure you always close your files!

```
writer.Close();
```

The Swindler launches another diabolical plan

The citizens of Objectville have long lived in fear of the Swindler, Captain Amazing's archnemesis. Now he's using a StreamWriter to implement another evil plan. Let's take a look at what's going on. Create a new Console App project and **add this Main code**, starting with a using declaration because StreamWriter is in the **System.IO namespace**:

```
using System.IO;           ← StreamWriter is in the
                           System.IO namespace
class Program
{
    static void Main(string[] args)
    {
        StreamWriter sw = new StreamWriter("secret_plan.txt");
        sw.WriteLine("How I'll defeat Captain Amazing");
        sw.WriteLine("Another genius secret plan by The Swindler");
        sw.WriteLine("I'll unleash my army of clones upon the citizens of Objectville.");
        string location = "the mall";
        for (int number = 1; number <= 5; number++)
        {
            sw.WriteLine("Clone #{0} attacks {1}", number, location);
            location = (location == "the mall") ? "downtown" : "the mall";
        }
        sw.Close();
    }
}
```

This line creates the StreamWriter object and tells it where the file will be.

See if you can figure out what's going on with the location variable and the ?: ternary operator.

It's really important that you call Close when you're done with the StreamWriter—that frees up any connections to the file, and any other resources that the StreamWriter instance is using. If you don't close the stream, some of the text won't get written (maybe none of it!).

Since you didn't include a full path in the filename, it wrote the output file **to the same folder as the binary**—so if you're running your app inside Visual Studio, check the bin\Debug\netcoreapp3.1 folder underneath your solution folder.

If you're using a different version of .NET, the subdirectory under Debug may be different.

Here's the output that it writes to *secret_plan.txt*:

```
How I'll defeat Captain Amazing
Another genius secret plan by The Swindler
I'll unleash my army of clones upon the citizens of
Objectville.
Clone #1 attacks the mall
Clone #2 attacks downtown
Clone #3 attacks the mall
Clone #4 attacks downtown
Clone #5 attacks the mall
```

Output

The Swindler is Captain Amazing's arch-nemesis, a shadowy supervillain bent on the domination of Objectville.





StreamWriter Magnets

Oops! These magnets were nicely arranged on the fridge with the code for the Flobbo class, but someone slammed the door and they all fell off. Can you rearrange them so the Main method produces the output below?

```
static void Main(string[] args) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

Assume all code files have
using System.IO;
at the top.

We added an extra challenge.

Something weird is going on with the Blobbo method. See how it has two different declarations in the first two magnets? We defined Blobbo as an **overloaded method**—there are two different versions, each with its own parameters, just like the overloaded methods you've used in previous chapters.

```
public bool Blobbo
    (bool Already, StreamWriter sw)
{
```

```
public bool Blobbo(StreamWriter sw) {
```

```
    if (Already) {
```

```
private string zap;
public Flobbo(string zap) {
    this.zap = zap;
}
```

```
public StreamWriter Snobbo() {
```

```
} else
{
```

```
    sw.WriteLine(zap);
    zap = "green purple";
    return false;
```

```
return new
StreamWriter("macaw.txt");
```

```
class Flobbo
{
```

```
    sw.WriteLine(zap);
    sw.Close();
    return false;
```

```
    sw.WriteLine(zap);
    zap = "red orange";
    return true;
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

Here's the output of
the app that gets
written to a file
called macaw.txt.

Output

```
blue yellow
green purple
red orange
```



StreamWriter Magnets Solution

Your job was to construct the Flobbo class from the magnets to create the desired output.

```
static void Main(string[] args) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

**Add the code to a console app.
The output will be written to
macaw.txt in the same folder
as the binary—a subdirectory
under the *bin\Debug* folder
inside the project folder.**

```
class Flobbo
{
```

Assume all code files have
using System.IO;
at the top.

```
private string zap;
public Flobbo(string zap) {
    this.zap = zap;
```

```
public StreamWriter Snobbo() {
    return new
        StreamWriter("macaw.txt");
}
```

```
public bool Blobbo(StreamWriter sw) {
    sw.WriteLine(zap);
    zap = "green purple";
    return false;
}
```

```
public bool Blobbo
    (bool Already, StreamWriter sw)
{
```

```
if (Already) {
    sw.WriteLine(zap);
    sw.Close();
    return false;
}
```

```
} else {
    sw.WriteLine(zap);
    zap = "red orange";
    return true;
}
```

Output

blue yellow
green purple
red orange

Defining overloaded methods

In Chapter 8 you learned how the Random.Next method is overloaded—there are three versions of it, each with a different set of parameters. The Blobbo method is overloaded too—it has two declarations with different parameters:

public bool Blobbo(StreamWriter sw)

and

public bool Blobbo(bool Already, StreamWriter sw)

The two overloaded Blobbo methods are completely separate from each other. They behave differently, just like the different overloaded versions of Random.Next behave differently. If you add those two methods to a class, the IDE will show them as overloaded methods, just like it did with Random.Next.

Make sure you close files when you're done with them. Take a minute and figure out why this is called after all of the text is written.

Just a reminder: we picked intentionally weird variable names and methods in these puzzles because if we used really good names, the puzzles would be too easy! Don't use names like this in your code, OK?

Use a StreamReader to read a file

Let's read the Swindler's secret plans with **StreamReader**, a class that's a lot like StreamWriter—except instead of writing a file, you create a StreamReader and pass it the name of the file to read in its constructor. Its ReadLine method returns a string that contains the next line from the file. You can write a loop that reads lines from it until its EndOfStream field is true—that's when it runs out of lines to read. Add this console app that uses a StreamReader to read one file, and a StreamWriter to write another file:

```
using System.IO;

class Program
{
    static void Main(string[] args)
    {
        var folder = Environment.GetFolderPath(Environment.SpecialFolder.Personal);

        var reader = new StreamReader($"{folder}{Path.DirectorySeparatorChar}secret_plan.txt");
        var writer = new StreamWriter($"{folder}{Path.DirectorySeparatorChar}emailToCaptainA.txt");

        writer.WriteLine("To: CaptainAmazing@objectville.net");
        writer.WriteLine("From: Commissioner@objectville.net");
        writer.WriteLine("Subject: Can you save the day... again?");
        writer.WriteLine();
        writer.WriteLine("We've discovered the Swindler's terrible plan:");

        while (!reader.EndOfStream) ←
        {
            var lineFromThePlan = reader.ReadLine();
            writer.WriteLine($"The plan -> {lineFromThePlan}");
        }
        writer.WriteLine();
        writer.WriteLine("Can you help us?");

        writer.Close(); } The StreamReader and StreamWriter each created their own
        reader.Close(); } streams. Calling their Close methods tells them to close those streams.
    }
}
```

This returns the path of the user's Documents folder on Windows, or the user's home directory on macOS. Make sure that you copy secret_plan.txt into that folder! Check out the **SpecialFolder** enum to see what other folders you can find.

Pass the file you want to read to the StreamReader's constructor.

The EndOfStream property is true if the reader has finished reading all the data in the file.

This loop reads a line from the reader and writes it out to the writer.

Output

```
To: CaptainAmazing@objectville.net
From: Commissioner@objectville.net
Subject: Can you save the day... again?

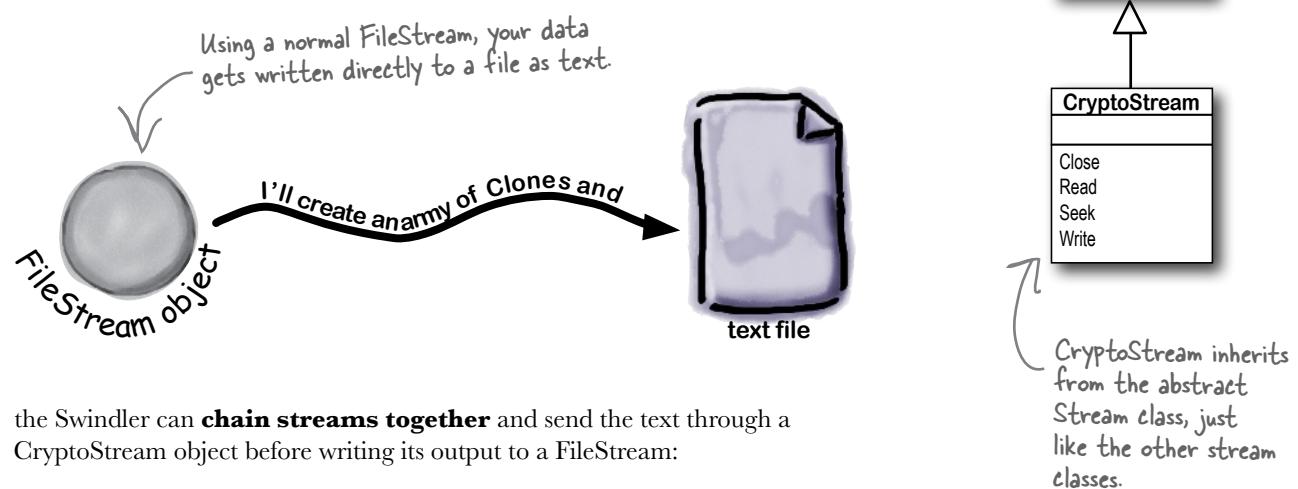
We've discovered the Swindler's terrible plan:
The plan -> How I'll defeat Captain Amazing
The plan -> Another genius secret plan by The Swindler
The plan -> I'll unleash my army of clones upon the citizens of Objectville.
The plan -> Clone #1 attacks the mall
The plan -> Clone #2 attacks downtown
The plan -> Clone #3 attacks the mall
The plan -> Clone #4 attacks downtown
The plan -> Clone #5 attacks the mall

Can you help us?
```

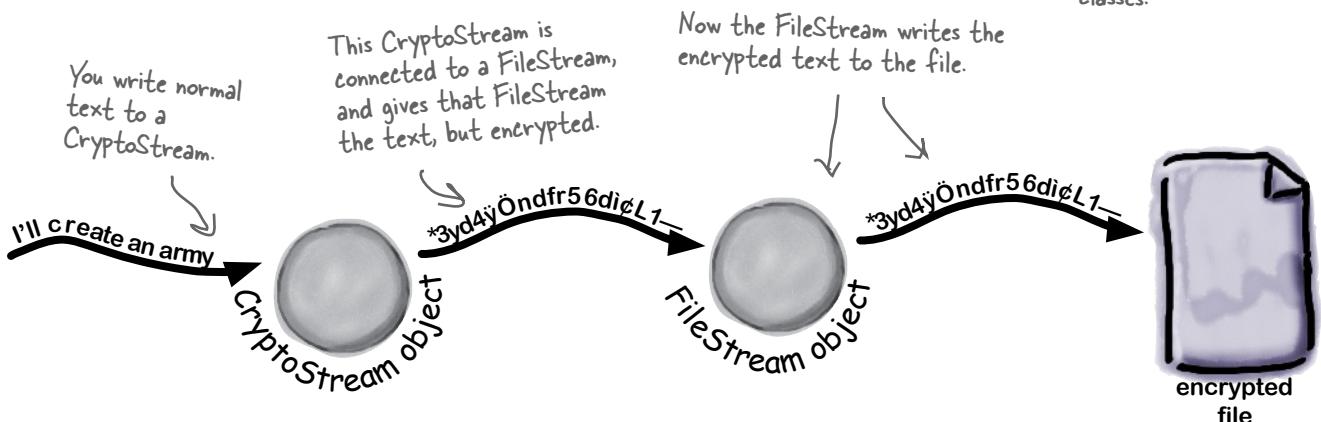
StreamReader is a class that reads characters from streams, but it's not a stream itself. When you pass a filename to its constructor, it creates a stream for you, and closes it when you call its Close method. It also has an overloaded constructor that takes a reference to a Stream.

Data can go through more than one stream

One big advantage to working with streams in .NET is that you can have your data go through more than one stream on its way to its final destination. One of the many types of streams in .NET Core is the CryptoStream class. This lets you encrypt your data before you do anything else with it. So instead of writing plain text to a regular old text file:



the Swindler can **chain streams together** and send the text through a CryptoStream object before writing its output to a FileStream:



You can **CHAIN** streams. One stream can write to another stream, which writes to another stream...often ending with a network or file stream.

Pool Puzzle



Your **job** is to take code snippets from the pool and place them into the blank lines in the Pineapple, Pizza, and Party classes. You can use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make the program write a file called *order.txt* with the five lines listed in the output box below.

```
class Pineapple {
    const _____d = "delivery.txt";
    public _____
        { North, South, East, West, Flamingo }
    public static void Main(string[] args) {
        _____o = new _____("order.txt");
        var pz = new _____(new _____(d, true));
        pz._____ (Fargo.Flamingo);
        for (_____w = 3; w >= 0; w--) {
            var i = new _____(new _____(d, false));
            i.Idaho((Fargo)w);
            Party p = new _____(new _____(d));
            p.HowMuch(o);
        }
        o._____ ("That's all folks!");
        o._____();
    }
}
```

The code writes these lines to the file *order.txt*.

order.txt

West
East
South
North
That's all folks!

Note: each snippet from the pool can be used more than once!

HowMany
HowMuch
HowBig
HowSmall

int
long
string
enum
class
ReadLine
WriteLine
Pizza
Party

Stream
reader
writer
StreamReader
StreamWriter
Open
Close

public
private
this
class
static

for	=	Fargo
while	>=	Utah
foreach	<=	Idaho
var	!=	Dakota
	==	Pineapple
	++	
	--	

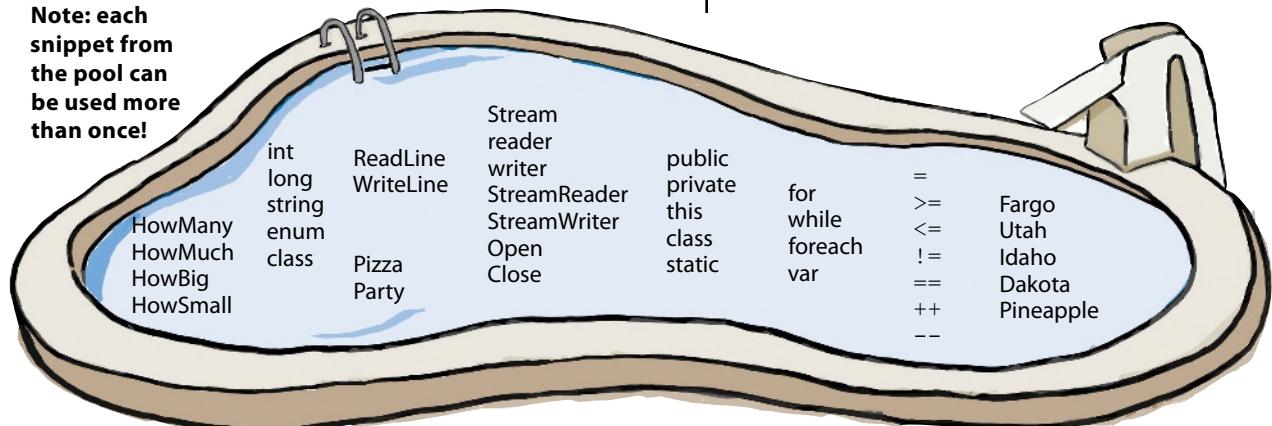
Here's a bonus question.
MINI Sharpen your pencil

What text does the app write to *delivery.txt*?

.....

```
class Pizza {
    private _____writer;
    public Pizza(_____) {
        this.writer = writer;
    }
    public void Idaho(_____.Fargo f) {
        writer._____(f);
        writer._____();
    }
}

class Party {
    private _____reader;
    public Party(_____) {
        this.reader = reader;
    }
    public void HowMuch(_____.q) {
        q._____(reader._____());
        reader._____();
    }
}
```





Poo! Puzzle Solution

Here's the entry point for the program. It creates a `StreamWriter` that it passes to the `Party` class. Then it loops through the `Fargo` members, passing each of them to the `Pizza.Idaho` method to print.

```
class Pineapple {
    const string d = "delivery.txt";
    public enum Fargo
        { North, South, East, West, Flamingo }
    public static void Main(string[] args) {
        var o = new StreamWriter("order.txt");
        var pz = new Pizza(new StreamWriter(d, true));
        pz.Idaho(Fargo.Flamingo);
        for (int w = 3; w >= 0; w--) {
            var i = new Pizza(new StreamWriter(d, false));
            i.Idaho((Fargo)w);
            Party p = new Party(new StreamReader(d));
            p.HowMuch(o);
        }
        o.WriteLine("That's all folks!");
        o.Close();
    }
}
```

This enum is used to print a lot of the output. We learned in Chapter 8 that an enum's `ToString` method returns the equivalent string, so `Fargo.North.ToString()` returns the string "North".

```
class Pizza {
    private StreamWriter writer;
    public Pizza(StreamWriter writer) {
        this.writer = writer;
    }
    public void Idaho(Pineapple.Fargo f) {
        writer.WriteLine(f);
        writer.Close();
    }
}
```

The `Pizza` class keeps a `StreamWriter` as a private field, and its `Idaho` method writes `Fargo` enums to the file using their `ToString` methods, which `WriteLine` calls automatically.

The `Party` class has a `StreamReader` field, and its `HowMuch` method reads a line from that `StreamReader` and writes it to a `StreamWriter`.

```
class Party {
    private StreamReader reader;
    public Party(StreamReader reader) {
        this.reader = reader;
    }
    public void HowMuch(StreamWriter q) {
        q.WriteLine(reader.ReadLine());
        reader.Close();
    }
}
```

Here's the output the app writes to the `order.txt` file.

order.txt

```
West
East
South
North
That's all folks!
```

MINI Sharpen your pencil
Solution

What text does the app write to `delivery.txt`?

North

there are no Dumb Questions

Q: Can you explain what you were doing with {0} and {1} when you called the StreamWriter Write and WriteLine methods?

A: When you're printing strings to a file, you'll often find yourself in the position of having to print the contents of a bunch of variables. For example, you might have to write something like this:

```
writer.WriteLine("My name is " + name +
    "and my age is " + age);
```

It gets really tedious and somewhat error-prone to have to keep using + to combine strings. It's easier to use **composite formatting**, where you use a **format string with placeholders** like {0}, {1}, {2}, etc., and follow it with variables to replace the placeholders:

```
writer.WriteLine(
    "My name is {0} and my age is {1}", name, age);
```

You're probably thinking, isn't that really similar to string interpolation? And you're right—it is! In some cases string interpolation may be easier to read, and in other cases using a format string is cleaner. Just like string interpolation, **format strings support formatting**. For example, {1:0.00} means format the second argument as a number with two decimal places, while {3:c} says to format the fourth argument in the local currency.

Oh, and one more thing—format strings work with Console.Write and Console.WriteLine too!

Q: What was that Path.DirectorySeparatorChar field that you used in the console app that used StreamReader?

A: We wrote that code to work on both Windows and macOS, so we took advantage of some of .NET Core's tools to help with that. Windows uses backslash characters as a path separator (C:\Windows), while macOS uses a forward slash (/Users).

Path.DirectorySeparatorChar is a read-only field that's set to the correct path separator character for the operating system: a \ on Windows and / on macOS and Linux.

We also used the Environment.GetFolderPath method, which returns the path of one of the special folders for the current user—in that case, the user's Documents folder on Windows or home directory on macOS.

Q: Near the beginning of the chapter you talked about converting a string to a byte array. How would that even work?

A: You've probably heard many times that files on disk are represented as bits and bytes. What that means is that when you write a file to a disk, the operating system treats it as one long sequence of bytes. The StreamReader and StreamWriter are converting from *bytes* to *characters* for you—that's called *encoding* and *decoding*. Remember from Chapter 4 how a byte variable can store any number between 0 and 255? Every file on your hard drive is one long sequence of numbers between 0 and 255. It's up to the programs that read and write those files to interpret those bytes as meaningful data. When you open a file in Notepad, it converts each individual byte to a character—for example, E is 69 and a is 97 (but this depends on the encoding...you'll learn more about encodings in just a minute). When you type text into Notepad and save it, Notepad converts each character back into a byte and saves it to disk. If you want to write a string to a stream, you'll need to do the same.

Q: If I'm just using a StreamWriter to write to a file, why do I really care if it's creating a FileStream for me?

A: If you're only reading or writing lines to or from a text file in order, then all you need are StreamReader and StreamWriter. As soon as you need to do anything more complex than that, you'll need to start working with other streams. If you ever need to write data like numbers, arrays, collections, or objects to a file, a StreamWriter just won't do. We'll go into a lot more detail about how that will work in just a minute.

Q: Why do I need to worry about closing streams after I'm done with them?

A: Have you ever had a word processor tell you it couldn't open a file because it was "busy"? When one program uses a file, Windows locks it and prevents other programs from using it. Your programs are no exception—Windows will do that for your apps when they open files, too. If you don't call the Close method, then it's possible for your program to keep a file locked until it ends.

Both Console and StreamWriter can use composite formatting, which replaces placeholders with values of parameters passed to Write or WriteLine.

Use the static File and Directory classes to work with files and directories

Like StreamWriter, the File class creates streams that let you work with files behind the scenes. You can use its methods to do most common actions without having to create the FileStreams first. The Directory class lets you work with whole directories full of files.

Things you can do with the static File class:

1 Find out if the file exists.

You can check to see if a file exists using the File.Exists method. It'll return true if it does, and false if it doesn't.

2 Read from and write to the file.

You can use the File.OpenRead method to get data from a file, or the File.Create or File.OpenWrite method to write to the file.

3 Append text to the file.

The File.AppendAllText method lets you append text to an already created file. It even creates the file if it's not there when the method runs.

4 Get information about the file.

The File.GetLastAccessTime and File.GetLastWriteTime methods return the date and time when the file was last accessed and modified.

FileInfo works just like File

If you're going to be doing a lot of work with a file, you might want to create an instance of the FileInfo class instead of using the File class's static methods.

The FileInfo class does just about everything the File class does, except you have to instantiate it to use it. You can create a new instance of FileInfo and access its Exists method or its OpenRead method in just the same way.

The big difference is that the File class is faster for a small number of actions, and FileInfo is better suited for big jobs.

Things you can do with the static Directory class:

1 Create a new directory.

Create a directory using the Directory.CreateDirectory method. All you have to do is supply the path; this method does the rest.

2 Get a list of the files in a directory.

You can create an array of files in a directory using the Directory.GetFiles method; just tell the method which directory you want to know about, and it will do the rest.

3 Delete a directory.

Need to delete a directory? Call the Directory.Delete method.

File is a static class, so it's just a set of methods that let you work with files. FileInfo is an object that you instantiate, and its methods are the same as the ones you see on File.



.NET has classes with a bunch of static methods for working with files and folders, and their method names are intuitive. The File class gives you methods to work with files, and the Directory class lets you work with directories. Write down what you think each of these lines of code does, then answer the two additional questions at the end.

Code	What the code does
<pre>if (!Directory.Exists(@"C:\SYP")) { Directory.CreateDirectory(@"C:\SYP"); }</pre>	
<pre>if (Directory.Exists(@"C:\SYP\Bonk")) { Directory.Delete(@"C:\SYP\Bonk"); }</pre>	
Directory.CreateDirectory(@"C:\SYP\Bonk");	
Directory.SetCreationTime(@"C:\SYP\Bonk", new DateTime(1996, 09, 23));	
<pre>string[] files = Directory.GetFiles(@"C:\SYP\", "*.log", SearchOption.AllDirectories);</pre>	
<pre>File.WriteAllText(@"C:\SYP\Bonk\weirdo.txt", @"This is the first line and this is the second line and this is the last line");</pre>	
<pre>File.Encrypt(@"C:\SYP\Bonk\weirdo.txt"); <i>See if you can guess what this one does—you haven't seen it yet.</i></pre>	
<pre>File.Copy(@"C:\SYP\Bonk\weirdo.txt", @"C:\SYP\copy.txt");</pre>	
<pre>DateTime myTime = Directory.GetCreationTime(@"C:\SYP\Bonk");</pre>	
<pre>File.SetLastWriteTime(@"C:\SYP\copy.txt", myTime);</pre>	
File.Delete(@"C:\SYP\Bonk\weirdo.txt");	

Why did we put @ in front of each of the strings that we passed as arguments to the methods above?

The filenames above all start with C:\ to work on Windows. What happens if you run that code on macOS or Linux?



.NET has classes with a bunch of static methods for working with files and folders, and their method names are intuitive. The File class gives you methods to work with files, and the Directory class lets you work with directories. Your job was to write down what each bit of code did.

Code	What the code does
<pre>if (!Directory.Exists(@"C:\SYP")) { Directory.CreateDirectory(@"C:\SYP"); }</pre>	Check if the C:\SYP folder exists. If it doesn't, create it.
<pre>if (Directory.Exists(@"C:\SYP\Bonk")) { Directory.Delete(@"C:\SYP\Bonk"); }</pre>	Check if the C:\SYP\Bonk folder exists. If it does, delete it.
<pre>Directory.CreateDirectory(@"C:\SYP\Bonk");</pre>	Create the directory C:\SYP\Bonk.
<pre>Directory.SetCreationTime(@"C:\SYP\Bonk", new DateTime(1996, 09, 23));</pre>	Set the creation time for the C:\SYP\Bonk folder to September 23, 1996.
<pre>string[] files = Directory.GetFiles(@"C:\SYP\", "*.log", SearchOption.AllDirectories);</pre>	Get a list of all files in C:\SYP that match the *.log pattern, including all matching files in any subdirectory.
<pre>File.WriteAllText(@"C:\SYP\Bonk\weirdo.txt", @"This is the first line and this is the second line and this is the last line");</pre>	Create a file called "weirdo.txt" (if it doesn't already exist) in the C:\SYP\Bonk folder and write three lines of text to it.
<pre>File.Encrypt(@"C:\SYP\Bonk\weirdo.txt"); ↗ This is an alternative to using a CryptoStream.</pre>	Take advantage of built-in Windows encryption to encrypt the file "weirdo.txt" using the logged-in account's credentials.
<pre>File.Copy(@"C:\SYP\Bonk\weirdo.txt", @"C:\SYP\Copy.txt");</pre>	Copy the C:\SYP\Bonk\weirdo.txt file to C:\SYP\Copy.txt.
<pre>DateTime myTime = Directory.GetCreationTime(@"C:\SYP\Bonk");</pre>	Declare the myTime variable and set it equal to the creation time of the C:\SYP\Bonk folder.
<pre>File.SetLastWriteTime(@"C:\SYP\copy.txt", myTime);</pre>	Alter the last write time of the copy.txt file in C:\SYP\ so it's equal to whatever time is stored in the myTime variable.
<pre>File.Delete(@"C:\SYP\Bonk\weirdo.txt");</pre>	Delete the C:\SYP\Bonk\weirdo.txt file.

Why did we put @ in front of each of the strings that we passed as arguments to the methods above?

The @ keeps the backslashes in the string from being interpreted as escape sequences.

The filenames above all start with C:\ to work on Windows. What happens if you run that code on macOS or Linux?

It will create a filename that starts with "C:\" and put it in the same folder as the binary.

IDisposable makes sure objects are closed properly

A lot of .NET classes implement a particularly useful interface called IDisposable. It **has only one member**: a method called Dispose. Whenever a class implements IDisposable, it's telling you that there are important things that it needs to do in order to shut itself down, usually because it's **allocated resources** that it won't give back until you tell it to. The Dispose method is how you tell the object to release those resources.

Use the IDE to explore IDisposable

You can use the Go To Definition feature in the IDE (or “Go to Declaration” on a Mac) to see the definition of IDisposable. Go to your project and type **IDisposable** anywhere inside a class. Then right-click on it and select Go To Definition from the menu. It'll open a new tab with code in it. Expand all of the code and this is what you'll see:

```
namespace System
{
    /// <summary>
    /// Provides a mechanism for releasing unmanaged resources.
    /// </summary>
    public interface IDisposable
    {
        /// <summary>
        /// Performs application-defined tasks associated with
        /// freeing, releasing, or resetting unmanaged resources.
        /// </summary>
        void Dispose();
    }
}
```

Any class that implements IDisposable must immediately release any resources that it took over as soon as you call its Dispose method. It's almost always the last thing you do before you're done with the object.

A lot of classes allocate important resources, like memory, files, and other objects. That means they take them over, and don't give them back until you tell them you're done with those resources.

al-lo-cate, verb.
to distribute resources or duties for a particular purpose. *The programming team was irritated at their project manager because he allocated all of the conference rooms for a useless management seminar.*

Avoid filesystem errors with using statements

Throughout the chapter we've been stressing that you need to **close your streams**. That's because some of the most common bugs that programmers run across when they deal with files are caused when streams aren't closed properly. Luckily, C# gives you a great tool to make sure that never happens to you: IDisposable and the Dispose method. When you **wrap your stream code in a using statement**, it automatically closes your streams for you. All you need to do is **declare your stream reference** with a using statement, followed by a block of code (inside curly brackets) that uses that reference. When you do that, C# **automatically calls the Dispose method** as soon as it finishes running the block of code.

← These "using" statements are different from the ones at the top of your code.

A using statement is always followed by an object declaration...

...and then a block of code within curly braces.

```
using (var sw = new StreamWriter("secret_plan.txt")) {  
    sw.WriteLine("How I'll defeat Captain Amazing");  
    sw.WriteLine("Another genius secret plan");  
    sw.WriteLine("by The Swindler");  
}
```

After the last statement in the using block executes, it calls the Dispose method of the object being used.

In this case, the object being used is pointed to by sw—which was declared in the using statement—so the Dispose method of the Stream class is run... which closes the stream.

This using statement declares a variable sw that references a new StreamWriter and is followed by a block of code. After all of the statements in the block are executed, the using block will automatically call sw.Dispose.

Use multiple using statements for multiple objects

You can pile using statements on top of each other—you don't need extra sets of curly brackets or indents:

```
using (var reader = new StreamReader("secret_plan.txt"))  
using (var writer = new StreamWriter("email.txt"))  
{  
    // statements that use reader and writer  
}
```

Every stream has a Dispose method that closes the stream. When you declare your stream in a using statement, it will always get closed! And that's important, because some streams don't write all of their data until they're closed.

When you declare an object in a using block, that object's Dispose method is called automatically.

Use a **MemoryStream** to stream data to memory

We've been using streams to read and write files. What if you want to read data from a file and then, well, do something with it? You can use a **MemoryStream**, which keeps track of all data streamed to it by storing it in memory. For example, you can create a new **MemoryStream** and pass it as an argument to a **StreamWriter** constructor, and then any data you write with the **StreamWriter** will be sent to that **MemoryStream**. You can retrieve that data using the **MemoryStream.ToArray** **method**, which returns all of the data that's been streamed to it in a byte array.

Use **Encoding.UTF8.GetString** to convert byte arrays to strings

One of the most common things that you'll do with byte arrays is convert them to strings. For example, if you have a byte array called `bytes`, here's one way to convert it to a string:

```
var converted = Encoding.UTF8.GetString(bytes);
```

*We mentioned "encoding" earlier.
What do you think that means?*

Here's a small console app that uses composite formatting to write a number to a **MemoryStream**, and then convert it to a byte array and then to a string. Just one problem...**it doesn't work!**

Create a new console app and add this code to it. Can you sleuth out the problem and fix it?

```
using System;
using System.IO;
using System.Text;

class Program
{
    static void Main(string[] args)
    {
        using (var ms = new MemoryStream())
        using (var sw = new StreamWriter(ms))
        {
            sw.WriteLine("The value is {0:0.00}", 123.45678);
            Console.WriteLine(Encoding.UTF8.GetString(ms.ToArray()));
        }
    }
}
```

The `MemoryStream.ToArray` method returns all of the streamed data as a byte array. The `GetString` method converts that byte array to a string.

Do this!

This app doesn't work! When you run it, it's supposed to write a line of text to the console, but it doesn't write anything at all. We'll explain what's wrong, but before we do first see if you can sleuth it out on your own.

Here's a hint: can you figure out when the streams are closed?

Q: Remind me why you put an @ in front of the strings that contained filenames in that "Sharpen your pencil" exercise?

A: Because if we didn't, the \S in "C:\SYP" would be interpreted as an invalid escape sequence and throw an exception. When you add a string literal to your program, the compiler converts escape sequences like \n and \r to special characters. Windows filenames have backslash characters in them, but C# strings normally use backslashes to start escape sequences. If you put @ in front of a string, it tells C# not to interpret escape sequences. It also tells C# to include line breaks in your string, so you can hit Enter halfway through the string and it'll include that as a line break in the output.

Q: And what exactly are escape sequences?

A: An escape sequence is a way to include special characters in your strings. For example, \n is a line feed, \t is a tab, and \r is a return character, or half of a Windows return (in Windows text files, lines have to end with \r\n; for macOS and Linux, lines just end in \n). If you need to include a quotation mark inside a string, you can use \" and it won't end the string. If you want to use an actual backslash in your string and not have C# interpret it as the beginning of an escape sequence, just do a double backslash: \\.

We gave you a chance to sleuth this problem out on your own. Here's how we fixed it.



Sleuth it out

Sherlock Holmes once said, "Data! Data! Data! I can't make bricks without clay." Let's start at the scene of the crime: our code that's not working. We'll scour it for all of the data that we can find by digging up clues.

How many of these clues did you spot?

- ★ We instantiate a StreamWriter that feeds data into a new MemoryStream.
- ★ The StreamWriter writes a line of text to the MemoryStream.
- ★ The contents of the MemoryStream are copied to an array and converted to a string.
- ★ This all happens inside a using block, so the streams are definitely closed.

If you spotted all of those clues, then congratulations—you've been sharpening your code detective skills! But like in every great mystery, there's always one last clue, some fact we learned earlier that proves to be the key to unraveling the whole crime and finding the culprit.

We used a using block, so we know that the streams definitely get closed. **But when are they closed?** Which leads us to the keystone to this mystery, the all-important clue that we learned just moments before the crime: **some streams don't write all of their data until they're closed**.

The StreamWriter and MemoryStream are declared in the same using block, so both Dispose methods are called *after the last line in the block is executed*. What does that mean? It means the MemoryStream.ToArray method is called *before the StreamWriter is closed*.

So we can fix the problem by adding a **nested** using block to *first* close the StreamWriter and *then* call ToArray:

```
using System;
using System.IO;
using System.Text;

class Program
{
    static void Main(string[] args)
    {
        using (var ms = new MemoryStream())
        {
            using (var sw = new StreamWriter(ms))
            {
                sw.WriteLine("The value is {0:0.00}", 123.45678);
            }
            Console.WriteLine(Encoding.UTF8.GetString(ms.ToArray()));
        }
    }
}
```

The MemoryStream is declared in the outer using block so it can stay open even after the StreamWriter is closed.

The inner using block makes sure the StreamWriter is closed before the MemoryStream.ToArray method gets called.

Stream objects often have data in memory that's **buffered**, or waiting to be written. When the stream empties all of that data, it's called **flushing**. If you need to flush the buffered data without closing the stream, you can also call its **Flush** method.



Exercise

In Chapter 8 you created a Deck class that kept track of a sequence of Card objects, with methods to reset it to a 52-card deck in order, shuffle the cards to randomize their order, and sort the cards to put them back in order. Now you'll add a method to write the cards to a file, and a constructor that lets you initialize a new deck by reading cards from a file.

Start by reviewing the Deck and Card classes that you wrote in Chapter 8

You created your Deck class by extending a generic collection of Card objects. This allowed you to use some useful members that Deck inherited from Collection<Card>:

- ★ The Count property returns the number of cards in the deck.
- ★ The Add method adds a card to the top of the deck.
- ★ The RemoveAt method removes a card at a specific index from the deck.
- ★ The Clear method removes all cards from the deck.

That gave you a solid starting point to add a Reset method that clears the deck and then adds 52 cards in order (Ace through King in each suit), a Deal method to remove the card from the top of the deck and return it, a Shuffle method to randomize the order of the cards, and a Sort method to put them back in order.

Add a method to write all of the cards in the deck to a file

Your Card class has a Name property that returns a string like “Three of Clubs” or “Ace of Hearts”. Add a method called WriteCards that takes a string with a filename as a parameter and writes the name of each card to a line in that file—so if you reset the deck and then call WriteCards, it will write 52 lines to the file, one for each card.

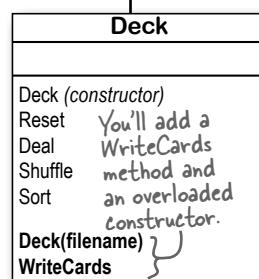
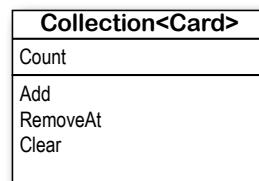
Add an overloaded Deck constructor that reads a deck of cards in from a file

Add a second constructor to the Deck class. Here's what it should do:

```
public Deck(string filename)
{
    // Create a new StreamReader to read the file.
    // For each line in the file, do the following four things:
    // Use the String.Split method: var cardParts = nextCard.Split(new char[] { ' ' });
    // Use a switch expression to get each card's suit: var suit = cardParts[2] switch {
    // Use a switch expression to get each card's value: var value = cardParts[0] switch {
    // Add the card to the deck.
}
```

In Chapter 9 you learned that switch expressions must be exhaustive, so add a default case that **throws a new InvalidDataException** if it encounters a suit or value that it doesn't recognize—this will make sure each card is valid.

Here's a Main method that you can use to test your app. It creates a deck with 10 random cards, writes it to a file, and then reads that file into a second deck and writes each of its cards to the console.



The String.Split method lets you specify an array of separator characters (in this case, a space), uses them to split the string into parts, and returns an array with each part.

```
static void Main(string[] args) {
    var filename = "deckofcards.txt";
    Deck deck = new Deck();
    deck.Shuffle();
    for (int i = deck.Count - 1; i > 10; i--)
        deck.RemoveAt(i);
    deck.WriteCards(filename);

    Deck cardsToRead = new Deck(filename);
    foreach (var card in cardsToRead)
        Console.WriteLine(card.Name);
}
```



Here are the two methods that you added to the Deck class. The WriteCards method uses a StreamWriter to write each card to a file, and the overloaded Deck constructor uses a StreamReader to read each card from a file. Since you're using a StreamWriter and StreamReader, make sure you add using System.IO; to the top of the file.

```

public void WriteCards(string filename)
{
    using (var writer = new StreamWriter(filename))
    {
        for (int i = 0; i < Count; i++)
        {
            writer.WriteLine(this[i].Name);
        }
    }
}

public Deck(string filename)
{
    using (var reader = new StreamReader(filename))
    {
        while (!reader.EndOfStream)
        {
            var nextCard = reader.ReadLine();
            var cardParts = nextCard.Split(new char[] { ' ' });
            var value = cardParts[0] switch
            {
                "Ace" => Values.Ace,
                "Two" => Values.Two,
                "Three" => Values.Three,
                "Four" => Values.Four,
                "Five" => Values.Five,
                "Six" => Values.Six,
                "Seven" => Values.Seven,
                "Eight" => Values.Eight,
                "Nine" => Values.Nine,
                "Ten" => Values.Ten,
                "Jack" => Values.Jack,
                "Queen" => Values.Queen,
                "King" => Values.King,
                _ => throw new InvalidDataException($"Unrecognized card value: {cardParts[0]}")
            };
            var suit = cardParts[2] switch
            {
                "Spades" => Suits.Spades,
                "Clubs" => Suits.Clubs,
                "Hearts" => Suits.Hearts,
                "Diamonds" => Suits.Diamonds,
                _ => throw new InvalidDataException($"Unrecognized card suit: {cardParts[2]}"),
            };
            Add(new Card(value, suit));
        }
    }
}

```

This line tells C# to split the nextCard string using a space as a separator character. That splits the string "Six of Diamonds" into the array {"Six", "of", "Diamonds"}.

This switch expression checks the first word in the line to see if it matches a value. If it does, the right Value enum is assigned to the "value" variable.

The default cases in the switch expressions throw an exception if the file contains an invalid card.

We do the same thing for the third word in the line, except we convert this one to a Suit enum.

BULLET POINTS

- Whenever you want to read data from a file or write data to a file, you'll use a **Stream** object. Stream is an abstract class, with subclasses that do different things.
- A **FileStream** lets you read from and write to files. A **MemoryStream** reads or writes data in memory.
- You can write your data to a stream through a stream's **Write method**, and read data using its **Read method**.
- A **StreamWriter** is a quick way to write data to a file. StreamWriter creates and manages a FileStream object for you automatically.
- A **StreamReader** reads characters from streams, but it's not a stream itself. It creates a stream for you, reads from it, and closes it when you call its Close method.
- The Write and WriteLine methods of StreamWriter and Console use **composite formatting**, which takes a format string with placeholders like {0}, {1}, {2} that support formatting like {1:0.00} and {3:c}.
- **Path.DirectorySeparatorChar** is a read-only field that's set to the path separator character for the operating system: a "\" on Windows and "/" on macOS and Linux.
- The **Environment.GetFolderPath method** returns the path of one of the special folders for the current user, such as the user's Documents folder on Windows or home directory on macOS.
- The **File class** has static methods including Exists (which checks if a file exists), OpenRead and OpenWrite (to get streams to read from or write to the file), and AppendAllText (to write text to a file in one statement).
- The **Directory class** has static methods including CreateDirectory (to create folders), GetFiles (to get the list of files), and Delete (to remove a folder).
- The **FileInfo class** is similar to the File class, except instead of using static methods it's instantiated.
- Remember to **always close a stream** after you're done with it. Some streams don't write all of their data until they're closed or their **Flush** methods are called.
- The **IDisposable interface** makes sure objects are closed properly. It includes one member, the **Dispose** method, which provides a mechanism for releasing unmanaged resources.
- Use a **using statement** to instantiate a class that implements IDisposable. The using statement is followed by a block of code; the object instantiated in the using statement is disposed of at the end of the block.
- Use **multiple using statements** in a row to declare objects that are disposed of at the end of the same block.

Windows and macOS have different line endings

If you're running Windows, open Notepad. If you're running macOS, openTextEdit. Create a file with two lines—the first line has the characters L1 and the second has the characters L2.

If you used Windows, it will contain these six bytes: 76 49 13 10 76 50

If you used macOS, it will contain these five bytes: 76 49 10 76 50

Can you spot the difference? You can see that the first and second lines are encoded with the same bytes: L is 76, 1 is 49, and 2 is 50. The line break is encoded differently: on Windows it's encoded with two bytes, 13 and 10, while on macOS it's encoded with one byte, 10. This is the difference between Windows-style and Unix-style line endings (macOS is a flavor of Unix). If you need to write code that runs on different operating systems and writes files with line endings, you can use the static **Environment.NewLine** property, which returns "\r\n" on Windows and "\r" on macOS or Linux.



ALL THAT CODE JUST TO READ IN ONE SIMPLE CARD? THAT'S WAY TOO MUCH WORK! WHAT IF MY OBJECT HAS A **WHOLE BUNCH OF PROPERTIES?** ARE YOU TELLING ME I NEED TO WRITE A SWITCH STATEMENT FOR EACH OF THEM?

There's an easier way to store your objects in files. It's called serialization.

Serialization means writing an object's entire state to a file or string. **Deserialization** means reading the object's state back from that file or string. So instead of painstakingly writing out each field and value to a file line by line, you can save your object the easy way by serializing it out to a stream. **Serializing** an object is like **flattening it out** so you can slip it into a file. On the other end, **deserializing** an object is like taking it out of the file and **inflating** it again.

OK, just to come clean here: there's also a method called `Enum.Parse` that will convert the string "Spades" to the enum value `Suits.Spades`. It even has a companion, `Enum.TryParse`, that works just like the `int.TryParse` method you've used throughout this book. But serialization still makes a lot more sense here. You'll find out more about that shortly...

What happens to an object when it's serialized?

It seems like something mysterious has to happen to an object in order to copy it off of the heap and put it into a file, but it's actually pretty straightforward.

1 Object on the heap

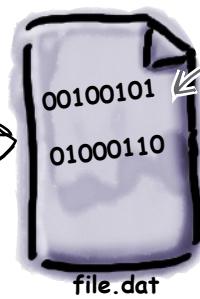
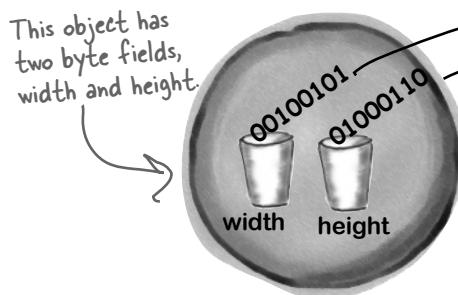


When you create an instance of an object, it has a **state**. Everything that an object “knows” is what makes one instance of a class different from another instance of the same class.

2 Object serialized



When C# serializes an object, it **saves the complete state of the object**, so that an identical instance (object) can be brought back to life on the heap later.



The field values for width and height are saved to the file.dat file, along with a little more info that code needs to restore the object later (like the types of the object and each of its fields).

Object on the heap again



3 And later on...

Later—maybe days later, and in a different program—you can go back to the file and **deserialize** it. That pulls the original class back out of the file and restores it **exactly as it was**, with all of its fields and values intact.

But what exactly IS an object's state? What needs to be saved?

We already know that **an object stores its state in its fields and properties**.

So when an object is serialized, each of those values needs to be saved to the file.

Serialization starts to get interesting when you have more complicated objects.

Chars, ints, doubles, and other value types have bytes that can just be written out to a file as is. What if an object has an instance variable that's an object *reference*?

What about an object that has five instance variables that are object references?

What if those object instance variables themselves have instance variables?

Think about it for a minute. What part of an object is potentially unique? Think about what needs to be restored in order to get an object that's identical to the one that was saved. Somehow everything on the heap has to be written to the file.

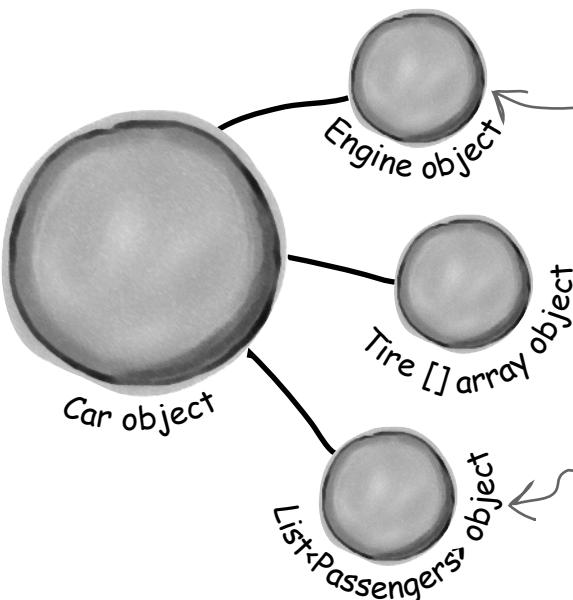
Brain Barbell is like a
“juiced up” Brain Power.
Take a few minutes and
really think about this one.



BRAIN BARBELL

What has to happen for this Car object to be saved so that it can be restored to its original state? Let's say the car has three passengers and a three-liter engine and all-weather radial tires...aren't those things all part of the Car object's state? What should happen to them?

The Car object has references to an Engine object, an array of Tire objects, and a List<> of Passenger objects. Those are part of its state, too—what happens to them?



The Engine object is private.
Should it be saved, too?

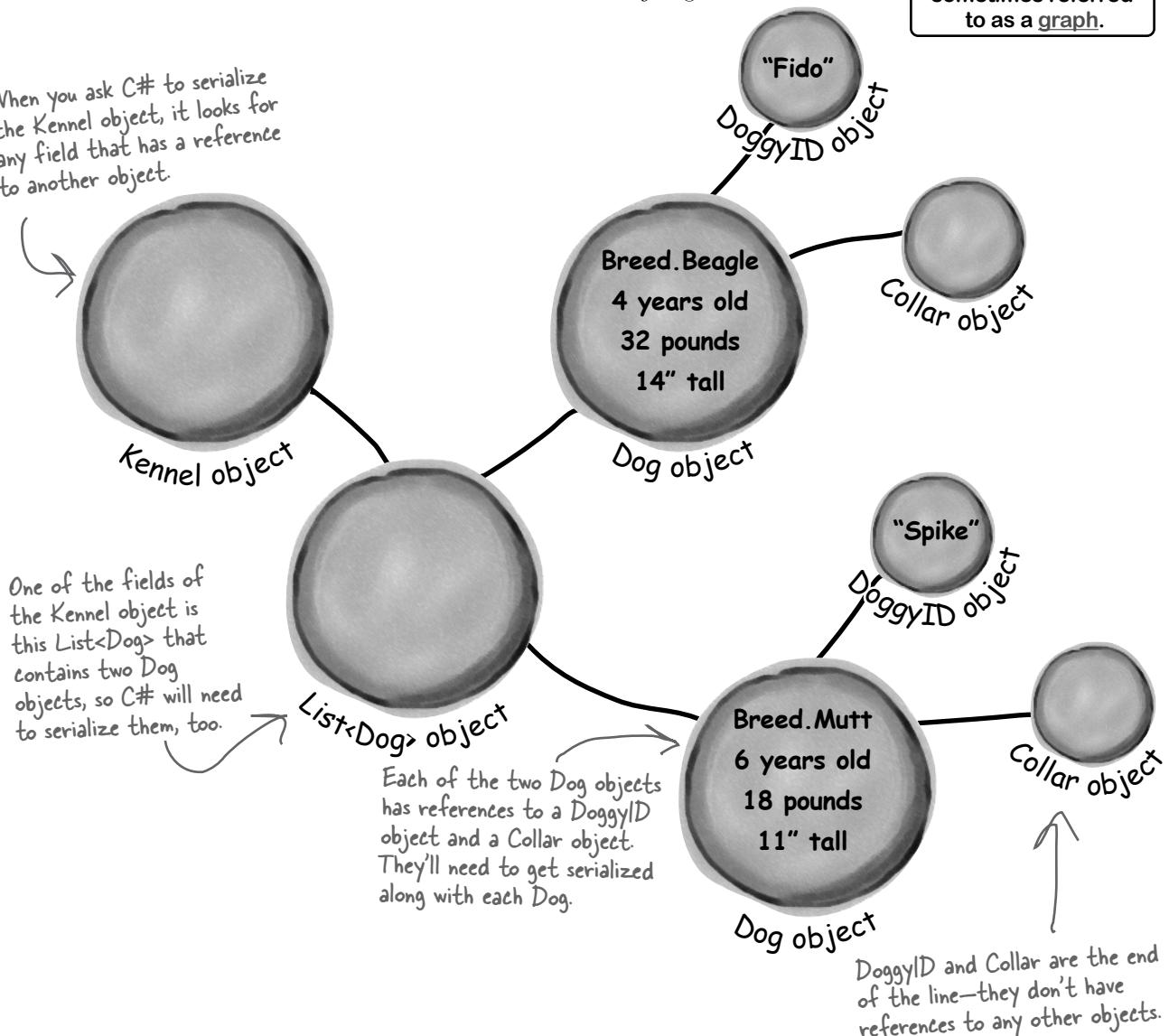
Each of the Passenger objects has its own references to other objects. Do those need to be saved, too?

When an object is serialized, all of the objects it refers to get serialized, too...

...and all of the objects *they* refer to, and all of the objects *those other objects* refer to, and so on and so on. Don't worry—it may sound complicated, but it all happens automatically. C# starts with the object you want to serialize and looks through its fields for other objects. Then it does the same for each of them. Every single object gets written out to the file, along with all the information C# needs to reconstitute it all when the object gets deserialized.

A group of objects connected to each other by references is sometimes referred to as a graph.

When you ask C# to serialize the Kennel object, it looks for any field that has a reference to another object.



Use **JsonSerialization** to serialize your objects

You're not just limited to reading and writing lines of text to your files. You can use **JSON serialization** to let your programs **copy entire objects** to strings (which you can write to files!) and read them back in...all in just a few lines of code! Let's take a look at how this works. Start by **creating a new console app**.

Do this!

1 Design some classes for your object graph.

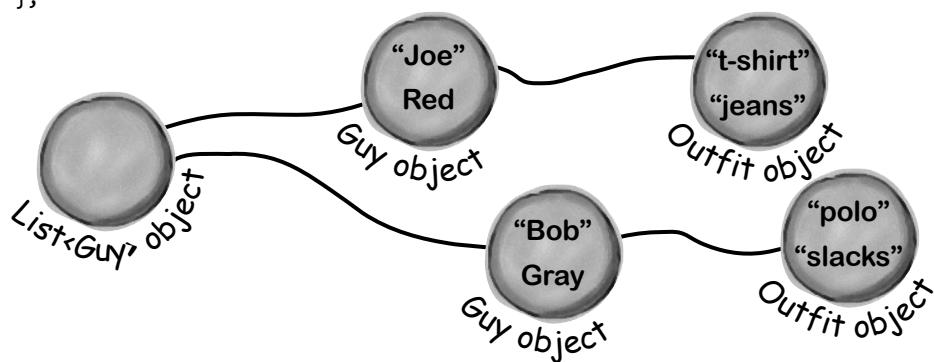
Add this HairColor enum and these Guy, Outfit, and HairStyle classes to your new console app:

```
class Guy {  
    public string Name { get; set; }  
    public HairStyle Hair { get; set; }  
    public Outfit Clothes { get; set; }  
    public override string ToString() => $"{Name} with {Hair} wearing {Clothes}";  
}  
  
class Outfit {  
    public string Top { get; set; }  
    public string Bottom { get; set; }  
    public override string ToString() => $"{Top} and {Bottom}";  
}  
  
enum HairColor {  
    Auburn, Black, Blonde, Blue, Brown, Gray, Platinum, Purple, Red, White  
}  
  
class HairStyle {  
    public HairColor Color { get; set; }  
    public float Length { get; set; }  
    public override string ToString() => $"{Length:0.0} inch {Color} hair";  
}
```

2 Create a graph of objects to serialize.

Now create a small graph of objects to serialize: a new List<Guy> pointing to a couple of Guy objects. Add this code to your Main method. It uses a collection initializer and object initializers to build the object graph:

```
static void Main(string[] args) {  
    var guys = new List<Guy>() {  
        new Guy() { Name = "Bob", Clothes = new Outfit() { Top = "t-shirt", Bottom = "jeans" },  
            Hair = new HairStyle() { Color = HairColor.Red, Length = 3.5f }  
        },  
        new Guy() { Name = "Joe", Clothes = new Outfit() { Top = "polo", Bottom = "slacks" },  
            Hair = new HairStyle() { Color = HairColor.Gray, Length = 2.7f }  
        },  
    };  
}
```



3 Use JsonSerializer to serialize the objects to a string.

First add a `using` directive to the top of your code file:

```
using System.Text.Json;
```

Now you can **serialize the entire graph** with a single line of code:

```
var jsonString = JsonSerializer.Serialize(guys);
Console.WriteLine(jsonString);
```

Run your app and look closely at what it prints to the console:

```
[{"Name": "Bob", "Hair": {"Color": 8, "Length": 3.5}, "Clothes": {"Top": "t-shirt", "Bottom": "jeans"}}, {"Name": "Joe", "Hair": {"Color": 5, "Length": 2.7}, "Clothes": {"Top": "polo", "Bottom": "slacks"}}]
```

That's your object graph **serialized to JSON** (which some people pronounce “Jason” and others pronounce “JAY-sahn”). It’s a *human-readable data interchange format*, which means that it’s a way to store complex objects using strings that a person can make sense of. Because it’s human readable, you can see that it has all of the parts of the graph: names and clothes are encoded as strings (“Bob”, “t-shirt”) and enums are encoded as their integer values.

4 Use JsonSerializer to deserialize the JSON to a new object graph.

Now that we have a string that contains the object graph serialized to JSON, we can **deserialize** it. That just means using it to create new objects. JsonSerializer lets us do that in one line of code, too. Add this to the Main method:

```
var copyOfGuys = JsonSerializer.Deserialize<List<Guy>>(jsonString);
foreach (var guy in copyOfGuys)
    Console.WriteLine("I serialized this guy: {0}", guy);
```

Run your app again. It deserializes the guys from the JSON string and writes them to the console:

```
I serialized this guy: Bob with 3.5 inch Red hair wearing t-shirt and jeans
I serialized this guy: Joe with 2.7 inch Gray hair wearing polo and slacks
```

JSON Up Close



Let's take a closer look at how JSON actually works. Go back to your app with the Guy object graph and replace the line that serializes the graph to a string with this:

```
var options = new JsonSerializerOptions() { WriteIndented = true };
var jsonString = JsonSerializer.Serialize(guys, options);
```

That code calls an overloaded `JsonSerializer.Serialize` method that takes a `JsonSerializerOptions` object that lets you set options for the serializer. In this case, you're telling it to write the JSON as indented text—in other words, it adds line breaks and spaces to make the JSON easier for people to read.

Now run the program again. The output should look like this: →

Let's break down exactly what we're seeing:

- ★ The JSON starts and ends with square brackets `[]`. This is how a list is serialized in JSON. A list of numbers would look like this: `[1, 2, 3, 4]`.
- ★ This particular JSON represents a list with two objects. Each object starts and ends with curly braces `{ }`—and if you look at the JSON, you can see that the second line is an opening curly brace `{`, the second-to-last line is a closing curly brace `}`, and in the middle there's a line with `,`, followed by a line with `{`. That's how JSON represents two objects—in this case, the two Guy objects.
- ★ Each object contains a set of keys and values that correspond to the serialized object's properties, separated by commas. For example, `"Name": "Joe"`, represents the first Guy object's Name property.
- ★ The Guy.Clothes property is an object reference that points to an Outfit object. It's represented by a nested object with values for Top and Bottom.

```
[  
  {  
    "Name": "Bob",  
    "Hair": {  
      "Color": 8,  
      "Length": 3.5  
    },  
    "Clothes": {  
      "Top": "t-shirt",  
      "Bottom": "jeans"  
    }  
  },  
  {  
    "Name": "Joe",  
    "Hair": {  
      "Color": 5,  
      "Length": 2.7  
    },  
    "Clothes": {  
      "Top": "polo",  
      "Bottom": "slacks"  
    }  
  }]
```

When you use `JsonSerializer` to serialize an object graph to JSON, it generates a (somewhat) readable text representation of the data in each object.

JSON only includes data, not specific C# types

When you were looking through the JSON data, you saw human-readable versions of the data in your objects: strings like “Bob” and “slacks”, numbers like 8 and 3.5, and even lists and nested objects. What *didn’t* you see when you looked at the JSON data? **JSON does not include the names of types.** Look inside a JSON file and you won’t see class names like Guy, Outfit, HairColor, or HairStyle, or even the names of basic types like int, string, or double. That’s because JSON just contains the data, and JsonSerializer will do its best to deserialize the data into whatever properties it finds.

Let’s put this to the test. Add a new class to your project:

```
class Dude
{
    public string Name { get; set; }
    public HairStyle Hair { get; set; }
}
```

We’re deserializing the data from a List of Guy objects into a Stack of Dude objects.

Now add this code to the end of your Main method:

```
var dudes = JsonSerializer.Deserialize<Stack<Dude>>(jsonString);
while (dudes.Count > 0)
{
    var dude = dudes.Pop();
    Console.WriteLine($"Next dude: {dude.Name} with {dude.Hair} hair");
}
```

And run your code again. Since the JSON just has a list of objects, JsonSerializer.Deserialize will happily stick them into a Stack (or a Queue, or an array, or another collection type). Since Dude has public Name and Hair properties that match the data, it will fill in any data that it can. Here’s what it prints to the output:

```
Next dude: Joe with 2.7 inch Gray hair hair
Next dude: Bob with 3.5 inch Red hair hair
```



Let’s use JsonSerializer to explore how strings are translated into JSON. Add the following code to a console app, then write down what each line of code writes to the console. The last line serializes the elephant animal emoji. ←

Console.WriteLine(JsonSerializer.Serialize(3));	You used the emoji panel in Chapter panel in Chapter to enter emoji.
Console.WriteLine(JsonSerializer.Serialize((long)-3));
Console.WriteLine(JsonSerializer.Serialize((byte)0));
Console.WriteLine(JsonSerializer.Serialize(float.MaxValue));
Console.WriteLine(JsonSerializer.Serialize(float.MinValue));
Console.WriteLine(JsonSerializer.Serialize(true));
Console.WriteLine(JsonSerializer.Serialize("Elephant"));
Console.WriteLine(JsonSerializer.Serialize("Elephant".ToCharArray()));
Console.WriteLine(JsonSerializer.Serialize("🐘"));

One more thing! We showed you basic serialization with JsonSerializer.
There are just a couple more things you need to know about it. ↴



Watch it!

JsonSerializer only serializes public properties (not fields), and requires a parameterless constructor.

Remember the SwordDamage class from Chapter 5? Its Damage property has a private set accessor:

```
public int Damage { get; private set; }
```

It also has a constructor that takes an int parameter:

```
public SwordDamage(int startingRoll)
```

JsonSerializer uses an object's setters when it deserializes data, so if an object has a private setter it won't be able to set the data.

You'll be able to use JsonSerializer to serialize a SwordDamage object without any problems. If you try to deserialize one, JsonSerializer will throw an exception—at least, it will if you use the code we've shown you. If you want to serialize objects that save their state in fields, or private properties, or use constructors with parameters, you'll need to create a converter. You can read more about that in the .NET Core serialization documentation: <https://docs.microsoft.com/en-us/dotnet/standard/serialization>.

BULLET POINTS

- **Serialization** means writing an object's entire state to a file or string. **Deserialization** means reading the object's state back from that file or string.
- A group of objects connected to each other by references is sometimes referred to as a **graph**.
- When an object is serialized, the **entire graph** of objects it refers to is serialized along with it so they can all be deserialized together.
- The **JsonSerializer class** has a static Serialize method that serializes an object graph to JSON, and a static Deserialize method that instantiates an object graph using serialized JSON data.
- JSON data is **human readable** (for the most part). Values are serialized as plain text: strings are written in "quotes" and other literals (like numbers and Boolean values) are encoded without quotes.
- JSON represents **arrays** of values using square brackets [].
- JSON represents **objects** inside curly brackets { }, with members and their values represented as key/value pairs separated by a colon.
- JSON **does not store specific types** like string or int, or specific class names. Instead, it relies on "smart" classes like JsonSerializer to do their best to match the data to the type that the data is being deserialized into.

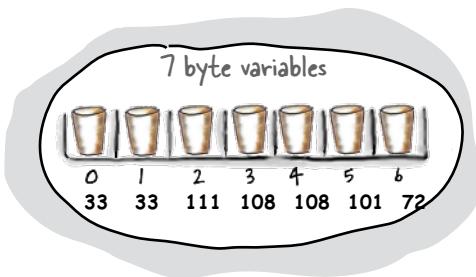
Next up: we'll take a deep dive into our data

You've been writing lots of code using value types like int, bool, and double, and creating objects that store data in fields. Now it's time to take a lower-level view of things. The rest of this chapter is all about getting to know your data better by understanding the actual bytes that C# and .NET use to represent it.

Here's what we'll do.



We'll explore how C# strings are encoded with Unicode—.NET uses Unicode to store characters and text.



We'll write values as binary data, then we'll read them back in and see what bytes were written.

0000:	45	6c	65	6d	65	6e	74	61	Elementa
0005:	72	79	2c	20	6d	79	20	64	ry, my d
0010:	65	61	72	20	57	61	74	73	ear Wats
0015:	6f	6e	21						on!

We'll build a hex dumper that lets us take a closer look at the bits and bytes in files.



Accessibility

Game design... and beyond

How would you play your favorite game if you were visually impaired? What if you had a motor or dexterity impairment that made it difficult for you to hold a controller? The goal of **accessibility** is to make sure that the programs that we build are designed so they can be used by people with disabilities or impairments. Game accessibility is all about making our games accessible to everyone, regardless of any impairments.

There are a few approaches to game testing you'll want to think about as you start designing and building games:

- “**Wait, what? There are blind gamers?**” Yes! Did you think that the “video” part of video games means they’re inaccessible to people with visual impairments? Take a few minutes to search YouTube for “blind gamer” and watch videos of people with visual impairments, including folks who are totally blind, showcase serious gaming skills.
- One really important thing you can do as a developer to make your game accessible is to take time to **understand players with impairments**. So what did you learn from watching those videos?
- One thing we learned from watching blind gamers play is that they use **sounds from the games** to understand what’s going on. In a fighting game, different moves may make different sounds. In a platformer, enemies coming towards the player might make telltale chirps or clicks. We also learned that while some games provide adequate **audio cues**, others don’t—and few (if any) games are *designed* to give totally blind players adequate audio cues to play the game.
- On the other hand, many gamers have **hearing impairments**, so it’s important that you don’t rely only on audio cues. Try playing your game with the sound on mute. Does that keep important information from being conveyed to the player? Are there visual cues that go along with the audio? Are there subtitles for all of the dialog? Make sure your game is playable without sound.
- 1 in 12 men and 1 in 200 women have some form of **color blindness** (including one of the authors of this book!). Many high-budget games include a *color blind mode* that does sophisticated color adjustments. You can make your game more accessible to color-blind people by using colors that have a **high contrast** with each other.
- Many gamers have a wide range of **motor impairments**, from repetitive strain injuries to paralysis. Players who are unable to use conventional input devices like a mouse and keyboard or a controller may use **assistive hardware** like an eye tracker or modified controller. One way you can accommodate these players is to make it easy for your games to accept keyboard mappings by letting players set up a profile where different keys map to the various game controls.
- You may have noticed that many games start with a warning screen about seizures. That’s because many gamers with **epilepsy** are photosensitive, which means that certain flashing or flickering patterns can cause them to have seizures. While seizure warnings are important, we can do better. As developers, it’s important for us to put effort into understanding and avoiding the kinds of strobe lights, flashing, and other visual patterns most likely to trigger seizures. Take the time to read this editorial by video game reviewer Cathy Vice **about her experience as a gamer with epilepsy**: <https://indiegamerchick.com/2013/08/06/the-epilepsy-thing>.

Accessibility is often abbreviated as #*a11y*—that’s a “numeronym,” or a kind of shorthand that says a followed by 11 letters (“ccessibilit”) followed by *y*. It’s a great way to remind us that **we can all be responsible allies**.

Accessibility features are often included as an afterthought, but your games—and any other kind of program!—come out better if you think about these things from the very beginning.

C# strings are encoded with Unicode

You've been using strings since you typed "Hello, world!" into the IDE at the start of Chapter 1. Because strings are so intuitive, we haven't really needed to dissect them and figure out what makes them tick. But ask yourself...***what exactly is a string?***

A C# string is a **read-only collection of chars**. So if you actually look at how a string is stored in memory, you'll see the string "Elephant" stored as the chars 'E', 'l', 'e', 'p', 'h', 'a', 'n', and 't'. Now ask yourself...***what exactly is a char?***

A char is a character represented with **Unicode**. Unicode is an industry standard for **encoding** characters, or converting them into bytes so they can be stored in memory, transmitted across networks, included in documents, or pretty much anything else you want to do with them—and you're guaranteed that you'll always get the correct characters.

This is especially important when you consider just how many characters there are. The Unicode standard supports over 150 **scripts** (sets of characters for specific languages), including not just Latin (which has the 26 English letters and variants like é and ç) but scripts for many languages used around the world. The list of supported scripts is constantly growing, as the Unicode Consortium adds new ones every year (here's the current list: <http://www.unicode.org/standard/supported.html>).

Unicode supports another really important set of characters: **emoji**. All of the emoji, from the winking smiley face 😊 to the ever-popular pile of poo 💩, are Unicode characters.



Your animal matching game in Chapter 1 treated emoji characters like any other C# chars.



Watch it!

Assistive technology can be thrown off by Unicode characters.

Accessibility is *extremely* important. We felt that it's especially valuable to bring up accessibility here because—like many of our "Game design... and beyond" topics—we can use it to teach a lesson that applies to all software development. You've probably seen posts on social media that use emoji or other "funny-looking" characters, and many that use boldfaced, cursive, or upside-down characters. On some platforms, these are done with Unicode characters, and when done like that they can be very problematic for assistive technology.

Take a social media post like this: I'm using hand claps to emphasize points

On screen, those hand claps look fine. But a screen reader like Windows Narrator or macOS VoiceOver might read that message aloud like this: "I'm clapping hands using clapping hands hand clapping hands claps clapping hands to clapping hands emphasize clapping hands points."

You might see "fonts" that look like this: this is a Me\$%#@GE in a really pri@M FONT

A screen reader will read **m** as "mathematical bold Fraktur small m" and other characters as "script letter n" or "mathematical double stroke small a," or just leave them out entirely. The experience could be just as bad for someone using a braille reader. Are those assistive technologies somehow broken? Not at all—they're doing their job. Those are the real names for those Unicode characters, and the assistive technologies are accurately describing the text. As you go through the next part of this chapter, keep these examples in mind—they'll help you better understand how Unicode works.



WHEN I SERIALIZED THE ELEPHANT EMOJI TO JSON,
IT CAME OUT AS "\uD83D\uDC18"—I BET UNICODE HAS
SOMETHING TO DO WITH THAT, RIGHT?

Every Unicode character—including emoji—has a unique number called a code point.

The number for a Unicode character is called a **code point**. You can download a list of all of the Unicode characters here: <https://www.unicode.org/Public/UNIDATA/UnicodeData.txt>.

That's a large text file with a line for every Unicode character. Download it and search for “ELEPHANT” and you'll find a line that starts like this: **1F418; ELEPHANT**. The numbers 1F418 represent a **hexadecimal** (or **hex**) value. Hex values are written with the numbers 0 to 9 and letters A to F, and it's often easier to work with Unicode values (and binary values in general) in hex than in decimal. You can create a hex literal in C# by adding 0x to the beginning, like this: `0x1F418`.

1F418 is the Elephant emoji's **UTF-8 code point**. UTF-8 is the most common way to **encode** a character as Unicode (or represent it as a number). It's a variable-length encoding, using between 1 and 4 bytes. In this case, it uses 3 bytes: 0x01 (or 1), 0xF4 (or 244), and 0x18 (or 24).

But that's not what the JSON serializer printed. It printed a longer hex number: D83DDC18. That's because **the C# char type uses UTF-16**, which uses code points that consist of either one or two 2-byte numbers. The UTF-16 code point of the elephant emoji is 0xD83D 0xDC18. UTF-8 is much more popular than UTF-16, especially on the Web, so when you look up code points you're much more likely to find UTF-8 than UTF-16.

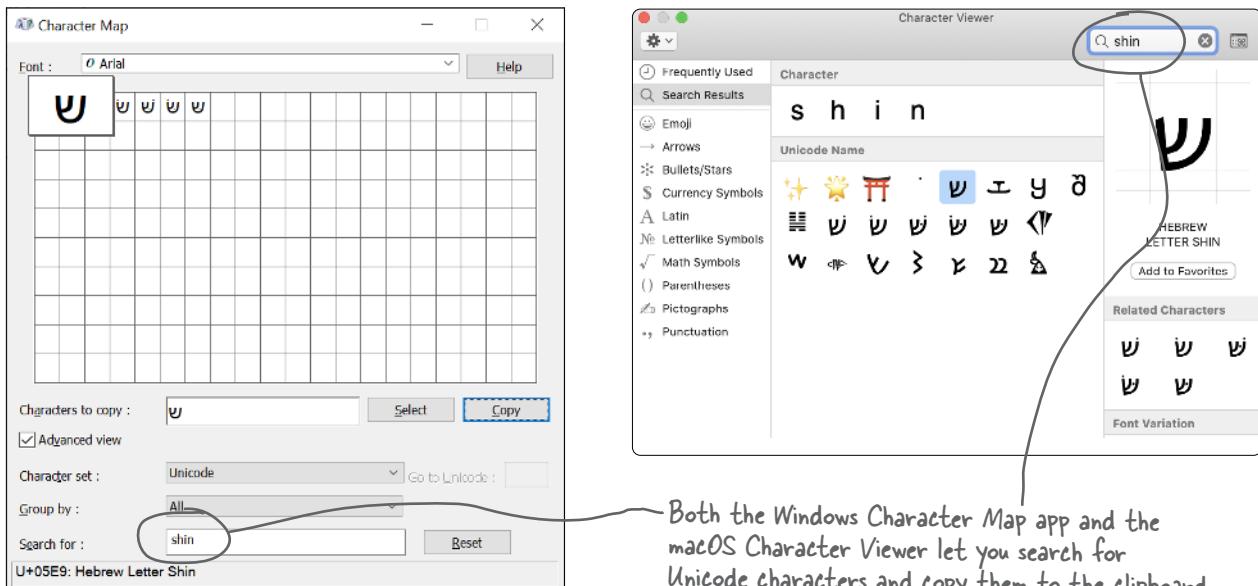
UTF-8 is a variable-length encoding used by most web pages and many systems. It can store characters using one, two, three, or more bytes. UTF-16 is a fixed-length encoding that always uses either one or two 2-byte numbers. .NET stores char values in memory as UTF-16 values.

Visual Studio works really well with Unicode

Let's use Visual Studio to see how the IDE works with Unicode characters. You saw back in Chapter 1 that you can use emoji in code. Let's see what else the IDE can handle. Go to the code editor and enter this code:

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

If you're using Windows, open up the Character Map app. If you're using a Mac, press **Ctrl-⌘-space** to pop up the Character Viewer. Then search for the Hebrew letter shin (ש) and copy it to the clipboard.



Place your cursor at the end of the string between the space and the quotation mark, and paste in the shin character that you copied to the clipboard. Hmm, something looks weird:

```
Console.WriteLine("Hello ψ");
```

Did you notice that the cursor is positioned to the *left* of the pasted letter? Well, let's continue. Don't click anywhere in the IDE—keep the cursor where it is, then switch over to Character Map or Character Viewer to search for the Hebrew letter lamed (ל). Switch back to the IDE—make sure the cursor is still positioned just left of the shin—and paste in the lamed:

```
Console.WriteLine("Hello þו");
```

When you pasted the lamed, the IDE added it to the left of the shin. Now search for the Hebrew letters vav (ו) and final mem (ם). Paste each of them into the IDE—it will insert them to the left of the cursor:

```
Console.WriteLine("Hello þוּוֹם");
```

The IDE knows that **Hebrew is read right to left**, so it's behaving accordingly. Click to select the text near the beginning of the statement, and slowly drag your cursor right to select Hello and then שׁוּם. Watch carefully what happens when the selection reaches the Hebrew letters. It skips to the shin (ש) and then selects from right to left—and that's exactly what a Hebrew reader would expect it to do.

.NET uses Unicode to store characters and text

The two C# types for storing text and characters—string and char—keep their data in memory as Unicode. When that data’s written out as bytes to a file, each of those Unicode numbers is written out to the file. Let’s get a sense of exactly how Unicode data is written out to a file. **Create a new console app.** We’ll use the File.WriteAllText and File.ReadAllBytes methods to start exploring Unicode.

Do this!

1 Write a normal string out to a file and read it back.

Add the following code to the Main method—it uses File.WriteAllText to write the string “Eureka!” out to a file called *eureka.txt* (so you’ll need `using System.IO;`). Then it creates a new byte array called `eurekaBytes`, reads the file into it, and prints out all of the bytes it read:

```
File.WriteAllText("eureka.txt", "Eureka!");  
byte[] eurekaBytes = File.ReadAllBytes("eureka.txt");  
foreach (byte b in eurekaBytes)  
    Console.Write("{0} ", b);  
Console.WriteLine(Encoding.UTF8.GetString(eurekaBytes));
```

The `ReadAllBytes` method returns a reference to a new array of bytes that contains all of the bytes that were read in from the file.

You’ll see these bytes written to the output: 69 117 114 101 107 97 33. The last line calls the method `Encoding.UTF8.GetString`, which converts a byte array with UTF-8-encoded characters to a string. Now **open the file in Notepad** (Windows) or **TextEdit** (Mac). It says “Eureka!”

2 Then add code to write the bytes as hex numbers.

When you’re encoding data you’ll often use hex, so let’s do that now. Add this code to the end of the Main method that writes the same bytes out, using `{0:x2}` to **format each byte as a hex number**:

```
foreach (byte b in eurekaBytes)  
    Console.Write("{0:x2} ", b);  
Console.WriteLine();
```

Hex uses the numbers 0 through 9 and letters A through F to represent numbers in base 16, so 6B is equal to 107.

That tells Write to print parameter 0 (the first one after the string to print) as a two-character hex code. So it writes the same seven bytes in hex instead of decimal: 45 75 72 65 6b—61 21.

3 Modify the first line to write the Hebrew letters “שְׁלִימָה” instead of “Eureka!”

You just added the Hebrew text שְׁלִימָה to another program using either Character Map (Windows) or Character Viewer (Mac). **Comment out the first line of the Main method and replace it with the following code** that writes “שְׁלִימָה” to the file instead of “Eureka!” We’ve added an extra `Encoding.Unicode` parameter so it writes UTF-16 (the Encoding class is in the `System.Text` namespace, so also add `using System.Text;` to the top):

```
File.WriteAllText("eureka.txt", "שְׁלִימָה", Encoding.Unicode);
```

Now run the code again, and look closely at the output: ff fe e9 05 dc 05 d5 05 dd 05. The first two characters are “FF FE”, which is the Unicode way of saying that we’re going to have a string of 2-byte characters. The rest of the bytes are the Hebrew letters—but they’re reversed, so U+05E9 appears as e9 05. Now open the file up in Notepad or TextEdit to make sure it looks right.

4

Use JsonSerializer to explore UTF-8 and UTF-16 code points.

When you serialized the elephant emoji, JsonSerializer generated `\uD83D\uDC18` – which we now know is the 4-byte UTF-16 code point in hex. Now let's try that with the Hebrew letter shin. Add using `System.Text.Json`; to the top of your app and then add this line:

```
Console.WriteLine(JsonSerializer.Serialize("₪"));
```

Run your app again. This time it printed a value with two hex bytes, “`\u05E9`”—that's the UTF-16 code point for the Hebrew letter shin. It's also the UTF-8 code point for the same letter.

But wait a minute—we learned that the UTF-8 code point for the elephant emoji is `0x1F418`, which is **different** than the UTF-16 code point (`0xD83D 0xDC18`). What's going on?

It turns out that most of the characters with 2-byte UTF-8 code points have the same code points in UTF-16. Once you reach the UTF-8 values that require three or more bytes—which includes the familiar emoji that we've used in this book—they differ. So while the Hebrew letter shin is `0x05E9` in both UTF-8 and UTF-16, the elephant emoji is `0x1F418` in UTF-8 and `0xD8ED 0xDC18` in UTF-16.

Use \u escape sequences to include Unicode in strings

When you serialized the elephant emoji, JsonSerializer generated `\UD83D\UDC18`—the 4-byte UTF-16 code point for the emoji in hex. That's because both JSON and C# strings use **UTF-16 escape sequences**—and it turns out JSON uses the same escape sequences.

Characters with 2-byte code points like ♂ are represented with a `\U` followed by the hex code point (`\u05E9`); characters with 4-byte code points like ⚪ are represented with `\U` and the highest two bytes, followed by `\U` and the lowest two bytes (`\UD83D\UDC18`).

C# also has another Unicode escape sequence: `\U` (with an UPPERCASE U) followed by eight hex bytes lets you embed a **UTF-32 code point**, which is always four bytes long. That's yet another Unicode encoding, and it's really useful because you can convert UTF-8 to UTF-32 by just padding the hex number with zeros—so the UTF-32 code point for ♂ is `\U000005E9`, and for ⚪ it's `\U0001F418`.

5

Use Unicode escape sequences to encode 🐘

Add these lines to your Main method to write the elephant emoji to two files using both the UTF-16 and UTF-32 escape sequences:

```
File.WriteAllText("elephant1.txt", "\uD83D\uDC18");
File.WriteAllText("elephant2.txt", "\U0001F418");
```

Run your app again, then open both of those files in Notepad orTextEdit. You should see the correct character written to the file.

You used UTF-16 and UTF-32 escape sequences to create your emoji, but the `WriteAllText` method writes a **UTF-8 file**. The `Encoding.UTF8.GetString` method you used in step 1 converts a byte array with **UTF-8 encoded data back to a string**.

write bytes not just text

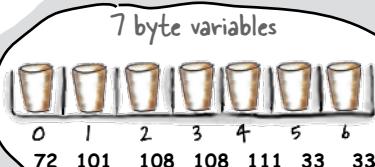
C# can use byte arrays to move data around

Since all your data ends up encoded as **bytes**, it makes sense to think of a file as one **big byte array**...and you already know how to read and write byte arrays.

Here's the code to create a byte array, open an input stream, and read the text "Hello!!" into bytes 0 through 6 of the array.



```
byte[] greeting;  
greeting = File.ReadAllBytes(filename);
```



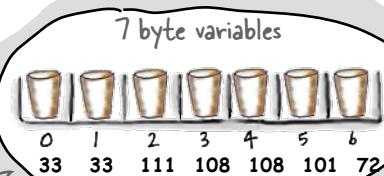
This is a static method in the `Array` class that reverses the order of the bytes. We're just using it to show that the changes you make to the byte array get written out to the file exactly.

```
Array.Reverse(greeting);  
File.WriteAllBytes(filename, greeting);
```

These numbers are the Unicode values for the characters in "Hello!!"



When the program writes the byte array out to a file, the text is in reverse order too.



Now the bytes are in reverse order.

Reversing the bytes in "Hello!!" only works because each of those characters is one byte long. Can you figure out why this won't work for 𠂇𠂇?

StreamWriter also encodes your data. It just specializes in text and text encoding—it defaults to UTF-8.

Use a BinaryWriter to write binary data

You **could** encode all of your strings, chars, ints, and floats into byte arrays before writing them out to files, but that would get pretty tedious. That's why .NET gives you a very useful class called **BinaryWriter** that **automatically encodes your data** and writes it to a file. All you need to do is create a FileStream and pass it into the BinaryWriter's constructor (they're in the System.IO namespace, so you'll need `using System.IO;`). Then you can call its methods to write out your data. Let's practice using BinaryWriter to write binary data to a file.

Do this!

- Start by creating a console app and setting up some data to write to a file:

```
1 int intValue = 48769414;
  string stringValue = "Hello!";
  byte[] byteArray = { 47, 129, 0, 116 };
  float floatValue = 491.695F;
  char charValue = 'E';
```

If you use `File.Create`, it'll start a new file—if there's one there already, it'll blow it away and start a brand-new one. The `File.OpenWrite` method opens the existing one and starts overwriting it from the beginning instead.

- To use a BinaryWriter, first you need to open a new stream with `File.Create`:

```
2 using (var output = File.Create("binarydata.dat"))
  using (var writer = new BinaryWriter(output))
  {
```

- Now just call its `Write` method. Each time you do this, it adds new bytes onto the end of the file that contain an encoded version of whatever data you passed it as a parameter:

```
writer.Write(intValue);
writer.Write(stringValue);
writer.Write(byteArray);
writer.Write(floatValue);
writer.Write(charValue);
}
```

Each `Write` statement encodes one value into bytes, and then sends those bytes to the `FileStream` object. You can pass it any value type, and it'll encode it automatically.

The `FileStream` writes the bytes to the end of the file.

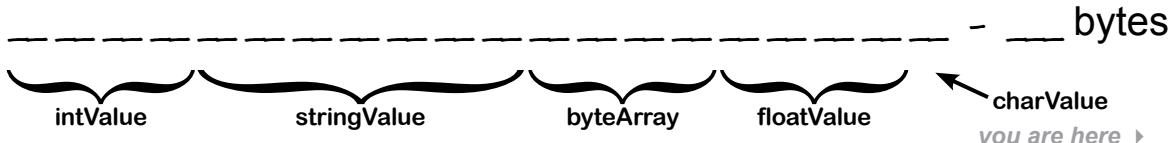
- Now use the same code you used before to read in the file you just wrote:

```
byte[] dataWritten = File.ReadAllBytes("binarydata.dat");
foreach (byte b in dataWritten)
    Console.WriteLine("{0:x2} ", b);
Console.WriteLine(" - {0} bytes", dataWritten.Length);
```

Write down the output in the blanks below. Can you **figure out what bytes correspond** to each of the five `writer.Write(...)` statements? We put a bracket under the groups of bytes that correspond with each statement to help you figure out which bytes in the file correspond with data written by the app.



Here's a hint: strings can be different lengths, so the string has to start with a number to tell .NET how long it is. `BinaryWriter` uses UTF-8 to encode strings, and in UTF-8 all of the characters in "Hello!" have UTF code points that consist of a single byte. Download `UnicodeData.txt` from unicode.org (we gave you the URL earlier) and use it to look up the code points for each character.



Use BinaryReader to read the data back in

The BinaryReader class works just like BinaryWriter. You create a stream, attach the BinaryReader object to it, and then call its methods...but the reader **doesn't know what data's in the file!** It has no way of knowing. Your float value of 491.695F was encoded as d8 f5 43 45. Those same bytes are a perfectly valid int—1,140,185,334—so you'll need to tell the BinaryReader exactly what types to read from the file. Add the following code to your program, and have it read the data you just wrote.

Don't take our word for it.
Replace the line that reads the float with a call to ReadInt32.
(You'll need to change the type of floatRead to int.) Then you can see for yourself what it reads from the file.

- Start out by setting up the FileStream and BinaryReader objects:

```
using (var input = File.OpenRead("binarydata.dat"))
using (var reader = new BinaryReader(input))
{
```

- You tell BinaryReader what type of data to read by calling its different methods:

```
int intRead = reader.ReadInt32();
string stringRead = reader.ReadString();
byte[] byteArrayRead = reader.ReadBytes(4);
float floatRead = reader.ReadSingle();
char charRead = reader.ReadChar();
```

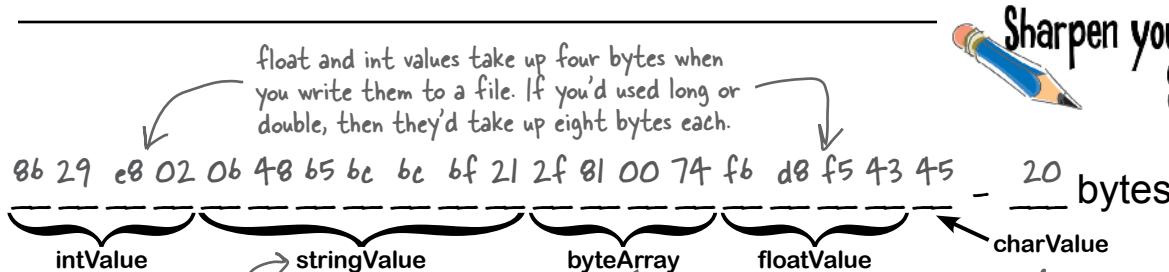
Each value type has its own method in BinaryReader that returns the data in the correct type. Most don't need any parameters, but ReadBytes takes one parameter that tells BinaryReader how many bytes to read.

- Now write the data that you read from the file to the console:

```
Console.WriteLine("int: {0} string: {1} bytes: ", intRead, stringRead);
foreach (byte b in byteArrayRead)
    Console.WriteLine("{0} ", b);
Console.WriteLine(" float: {0} char: {1} ", floatRead, charRead);
}
```

Here's the output that gets printed to the console:

```
int: 48769414 string: Hello! bytes: 47 129 0 116 float: 491.695 char: E
```



The first byte in the string is b—that's the length of the string. You can use Character Map to look up each of the characters in "Hello!"—it starts with U+0048 and ends with U+0021.



Both the Windows and Mac calculator apps have programmer modes that can convert these bytes from hex to decimal, which will let you convert them back to the values in the array.

char holds a Unicode character, and 'E' only takes one byte—it's encoded as U+0045.

there are no Dumb Questions

Q: Earlier in the chapter when I wrote “Eureka!” to a file and then read the bytes back, it took one byte per character. So why did each of the Hebrew letters in שלוֹן take two bytes? And why did it write the bytes “FF FE” at the beginning of the file?

A: What you’re seeing is the difference between two closely related Unicode encodings. Latin characters (including plain English letters), numbers, normal punctuation marks, and some standard characters (like curly brackets, ampersands, and other things you see on your keyboard) all have very low Unicode numbers—between 0 and 127. They correspond to a very old encoding called ASCII that dates back to the 1960s, and UTF-8 was designed to be backward compatible with ASCII. A file with only those Unicode characters contains just their bytes and nothing else.

Things get a little more complicated when you add Unicode characters with higher-numbered code points into the mix. One byte can only hold a number between 0 and 255. Two bytes in a row can store numbers between 0 and 65,536—which, in hex, is FFFF. The file needs to be able to tell whatever program opens it up that it’s going to contain these higher-numbered characters, so it puts a special reserved byte sequence at the beginning of the file: FF FE. That’s called the *byte order mark*. As soon as a program sees that, it knows that all of the characters are encoded with two bytes each (so an E is encoded as 00 45, with a leading zero).

Q: Why is it called a byte order mark?

A: Go back to the code that wrote שלוֹן to a file, then printed the bytes it wrote. You’ll see that the bytes in the file were reversed. For example, the ש code point U+05E9 was written to the file as E9 05. That’s called *little-endian*—it means the least significant byte is written first. Go back to the code that calls WriteAllText and modify it to **change the third argument from Encoding.Unicode to Encoding.BigEndianUnicode**. That tells it to write the data out in *big-endian*, which doesn’t flip the bytes around—when you run it again, you’ll see the bytes come out as 05 E9 instead. You’ll also see a different byte order mark: FE FF. This tells Notepad orTextEdit how to interpret the bytes in the file.

Q: Why didn’t I use a using block or call Close after I used File.ReadAllText and File.WriteAllText?

A: The File class has several very useful static methods that automatically open up a file, read or write data, and then **close it automatically**. In addition to the ReadAllText and WriteAllText methods, there are ReadAllBytes and WriteAllBytes, which work with byte arrays, and ReadAllLines and WriteAllLines, which read and write string arrays, where each string in the array is a separate line in the file. All of these methods automatically open and close the streams, so you can do your whole file operation in a single statement.

Q: If FileStream has methods for reading and writing, why do I ever need to use StreamReader and StreamWriter?

A: The FileStream class is really useful for reading and writing bytes in binary files. Its methods for reading and writing operate with bytes and byte arrays. A lot of programs work exclusively with text files, and that’s where StreamReader and StreamWriter come in really handy. They have methods that are built specifically for reading and writing lines of text. Without them, if you wanted to read a line of text from a file, you’d have to first read a byte array and then write a loop to search through that array for a linebreak—so it’s not hard to see how they make your life easier.

If you’re writing a string that only has Unicode characters with low numbers (such as Latin letters), it writes one byte per character. If it’s got high-numbered characters (like emoji characters), they’ll be written using two or more bytes each.

A hex dump lets you see the bytes in your files

A **hex dump** is a *hexadecimal* view of the contents of a file, and it's a really common way for programmers to take a deep look at a file's internal structure.

It turns out that hex is a convenient way to display bytes in a file. A byte takes two characters to display in hex: bytes range from 0 to 255, or 00 to ff in hex. That lets you see a lot of data in a really small space, and in a format that makes it easier to spot patterns. It's useful to display binary data in rows that are 8, 16, or 32 bytes long because most binary data tends to break down in chunks of 4, 8, 16, or 32...like all the types in C#. (For example, an int takes up 4 bytes.) A hex dump lets you see exactly what those values are made of.

How to make a hex dump of some plain text

Start with some familiar text using Latin characters:

When you have eliminated the impossible, whatever remains, however improbable, must be the truth. - Sherlock Holmes

First, break up the text into 16-character segments, starting with the first 16: When you have el

Next, convert each character in the text to its UTF-8 code point. Since the Latin characters all have **1-byte** UTF-8 code points, each will be represented by a two-digit hex number from 00 to 7F. Here's what each line of our dump will look like:

This is the segment's offset (or position in the file) written as a hex number.	These are the first 8 bytes of the 16-byte segment.	This divider makes the line more readable.	These are the last 8 bytes of the 16-byte segment.	These are the text characters that were dumped.
0000:	57 68 65 6e 20 79 6f 75 -- 20 68 61 76 65 20 65 6c			When you have el

Repeat until you've dumped every 16-character segment in the file:

0000: 57 68 65 6e 20 79 6f 75 -- 20 68 61 76 65 20 65 6c
0010: 69 6d 69 6e 61 74 65 64 -- 20 74 68 65 20 69 6d 70
0020: 6f 73 73 69 62 6c 65 2c -- 20 77 68 61 74 65 76 65
0030: 72 20 72 65 6d 61 69 6e -- 73 2c 20 68 6f 77 65 76
0040: 65 72 20 69 6d 70 72 6f -- 62 61 62 6c 65 2c 20 6d
0050: 75 73 74 20 62 65 20 74 -- 68 65 20 74 72 75 74 68
0060: 2e 20 2d 20 53 68 65 72 -- 6c 6f 63 6b 20 48 6f 6c
0070: 6d 65 73 0a --

When you have el
iminated the imp
ossible, whatev
er remains, howev
er improbable, m
ust be the truth
. - Sherlock Hol
mes.

And that's our dump. There are many hex dump programs for various operating systems, and each of them has a slightly different output. Each line in our particular hex dump format represents 16 characters in the input that was used to generate it, with the offset at the start of each line and the text for each character at the end. Other hex dump apps might display things differently (for example, rendering escape sequences or showing values in decimal).

A hex dump is a hexadecimal view of data in a file or memory, and can be a really useful tool to help you debug binary data.

Use StreamReader to build a hex dumper

Let's build a hex dump app that reads data from a file with StreamReader and writes its dump to the console. We'll take advantage of the **ReadBlock method** in StreamReader, which reads a block of characters into a char array: you specify the number of characters you want to read, and it'll either read that many characters or, if there are fewer than that many left in the file, it'll read the rest of the file. Since we're displaying 16 characters per line, we'll read blocks of 16 characters.

Create a new console app called HexDump. Before you add code, **run the app** to create the folder with the binary. Use Notepad orTextEdit to **create a text file called textdata.txt**, add some text to it, and put it in the same folder as the binary.

Here's the code from inside the Main method—it reads the *textdata.txt* file and writes a hex dump to the console. Make sure you add **using System.IO;** to the top.

```
static void Main(string[] args) {
    var position = 0; ↙ A StreamReader's EndOfStream property returns false if there are characters still left to read in the file.
    using (var reader = new StreamReader("textdata.txt")) {
        while (!reader.EndOfStream)
        {
            // Read up to the next 16 bytes from the file into a byte array
            var buffer = new char[16];
            var bytesRead = reader.ReadBlock(buffer, 0, 16);

            // Write the position (or offset) in hex, followed by a colon and space
            Console.WriteLine("{0:x4}: ", position);
            position += bytesRead;

            // Write the hex value of each character in the byte array
            for (var i = 0; i < 16; i++)
            {
                if (i < bytesRead)
                    Console.Write("{0:x2} ", (byte)buffer[i]);
                else
                    Console.Write("  ");
                if (i == 7) Console.Write("-- ");
            }

            // Write the actual characters in the byte array
            var bufferContents = new string(buffer);
            Console.WriteLine("  {0}", bufferContents.Substring(0, bytesRead));
        }
    }
}
```

This loop goes through the characters and prints each of them to a line in the output.

The **ReadBlock method** reads the next characters from its input into a byte array (sometimes referred to as a buffer). It **blocks**, which means it keeps executing and doesn't return until it's either read all of the characters you asked for or run out of data to read.

The **{0:x4}** formatter converts a numeric value to a four-digit hex number, so 1984 is converted to the string "07c0".

The **String.Substring method** returns a part of a string. The first parameter is the starting position (in this case, the beginning of the string), and the second is the number of characters to include in the substring. The String class has an overloaded constructor that takes a char array as a parameter and converts it to a string.

Now run your app. It will print a hex dump to the console:

0000: 45 6c 65 6d 65 6e 74 61 -- 72 79 2c 20 6d 79 20 64
0010: 65 61 72 20 57 61 74 73 -- 6f 6e 21

Elementary, my dear Watson!

pass a filename on the command line

Use Stream.Read to read bytes from a stream

The hex dumper works just fine for text files—but there's a problem. **Copy the binarydata.dat file** you wrote with BinaryWriter into the same folder as your app, then change the app to read it:

```
using (var reader = new StreamReader("binarydata.dat"))
```

Now run your app. This time it prints something else—but it's not quite right:

```
0000: fd 29 fd 02 06 48 65 6c -- 6c 6f 21 2f fd 00 74 fd ?)?Hello!/? t?  
0010: fd fd 43 45 -- ??CE
```

These bytes were 81 and f6 in the "Sharpen your pencil" solution, but StreamReader changed them to fd.

The text characters ("Hello!") seem OK. But compare the output with the "Sharpen your pencil" solution—the bytes aren't quite right. It looks like it replaced some bytes (86, e8, 81, f6, d8, and f5) with a different byte, fd. That's because **StreamReader is built to read text files**, so it only reads **7-bit values**, or byte values up to 127 (7F in hex, or 1111111 in binary—which are 7 bits).

So let's do this right—by **reading the bytes directly from the stream**. Modify the using block so it uses **File.OpenRead**, which opens the file and **returns a FileStream**. You'll use the Stream's Length property to keep reading until you've read all of the bytes in the file, and its Read method to read the next 16 bytes into the byte array buffer:

```
using (Stream input = File.OpenRead("binarydata.dat")) {  
    var buffer = new byte[16];  
    while (position < input.Length)  
    {  
        // Read up to the next 16 bytes from the file into a byte array  
        var bytesRead = input.Read(buffer, 0, buffer.Length);
```

We used an explicit type instead of var to make it clear that you're using a stream—specifically a FileStream (which extends Stream).

The Stream.Read method takes three arguments: the byte array to read into (buffer), the starting index in the array (0), and the number of bytes to read (buffer.Length).

The rest of the code is the same, except for the line that sets **bufferContents**:

```
// Write the actual characters in the byte array  
var bufferContents = Encoding.UTF8.GetString(buffer);
```

You used the Encoding class earlier in the chapter to convert a byte array to a string. This byte array contains a single byte per character—that means it's a valid UTF-8 string. That means you can use Encoding.UTF8.GetString to convert it. Since the Encoding class is in the System.Text namespace, you'll need to add **using System.Text**; to the top of the file.

Now run your app again. This time it prints the correct bytes instead of changing them to fd:

```
0000: 86 29 e8 02 06 48 65 6c -- 6c 6f 21 2f 81 00 74 f6 ?)?Hello!/? t?  
0010: d8 f5 43 45 -- ??CE
```

There's just one more thing we can do to clean up the output. Many hex dump programs replace non-text characters with dots in the output. **Add this line to the end of the for loop**:

```
if (buffer[i] < 0x20 || buffer[i] > 0x7F) buffer[i] = (byte)'.';
```

Now run your app again—this time the question marks are replaced with dots:

```
0000: 86 29 e8 02 06 48 65 6c -- 6c 6f 21 2f 81 00 74 f6 .)...Hello!..t.  
0010: d8 f5 43 45 -- ..CE
```

} Now your app reads all of the bytes from the file, not just the text characters.

Modify your hex dumper to use command-line arguments

Most hex dump programs are utilities that you run from the command line. You can dump a file by passing its name to the hex dumper as a **command-line argument**, like this: C:\> HexDump myfile.txt

Let's modify the hex dumper to use command-line arguments. When you create a console app, C# makes the command-line arguments available as the **args string array** that gets passed to the Main method:

```
static void Main(string[] args)
```

We'll modify the Main method to open a file and read its contents from a stream. The **File.OpenRead** **method** takes a filename as a parameter, opens it for reading, and returns a stream with the file contents.

Change these lines in your Main method:

```
static void Main(string[] args)
{
    var position = 0;
    using (Stream input = File.OpenRead(args[0]))
    {
        var buffer = new byte[16];
        int bytesRead;

        // Read up to the next 16 bytes from the file into a byte array
        while ((bytesRead = input.Read(buffer, 0, buffer.Length)) > 0) {
```

Now let's use the command-line argument in the IDE by **changing the debug properties** to pass a command line to the program. **Right-click on the project** in the solution, then:

- ★ **On Windows**, choose Properties, then click Debug, and enter the filename to dump in the Application arguments box (either the full path or the name of a file in the binary folder).
- ★ **On macOS**, choose Options, expand Run >> Configurations, click Default, and enter the filename in the Arguments box.

Now when you debug your app, its `args` array will contain the arguments you set in the project settings.

Make sure you specify a valid filename when you set up the command-line arguments.

Run your app from the command line

You can also run the app from the command line, replacing `[filename]` with the name of a file (either the full path or the name of a file in the current directory):

- ★ **On Windows**, Visual Studio builds an executable under the bin\Debug folder (in the same place you put your files to read), so you can run the executable directly from that folder. Open a command window, `cd` to the bin\Debug folder, and run `HexDump [filename]`.
- ★ **On a Mac**, you'll need to **build a self-contained application**. Open a Terminal window, go to the project folder, and run this command: `dotnet publish -r osx-x64`.

The output will include a line like this: `HexDump -> /path-to-binary/osx-x64/publish/` Open a Terminal window, `cd` to the full path that it printed, and run `./HexDump [filename]`.



Downloadable exercise: Hide and Seek

In this next exercise you'll build an app where you explore a house and play a game of Hide and Seek against a computer player. You'll put your collection and interface skills to the test when you lay out the locations. Then you'll turn it into a game, serializing the state of the game to a file so you can save and load it.

First you'll explore a virtual house, navigating from room to room and examining the items in each location.



Then you'll add a computer player who finds a hiding place. See how few moves it takes to find them!

Go to the GitHub page for the book and download the project PDF:
<https://github.com/head-first-csharp/fourth-edition>

BULLET POINTS

- Unicode is an industry standard for *encoding* characters, or converting them into bytes. Every one of the over one million Unicode characters has a *code point*, or a unique number assigned to it.
- Most files and web pages are encoded using **UTF-8**, a variable-length Unicode encoding that encodes some characters with either one, two, three, or four bytes.
- C# and .NET use **UTF-16** when storing characters and text in memory, treating a string as a **read-only collection of chars**.
- The **Encoding.UTF8.GetString** method converts a UTF-8 byte array to a string. **Encoding.Unicode** converts a byte array encoded with UTF-16 to a string, and **Encoding.UTF32** converts a UTF-32 byte array.
- Use **\u escape sequences** to include Unicode in C# strings. The \u escape sequence encodes UTF-16, while \U encodes **UTF-32**, a 4-byte fixed-length encoding.
- StreamWriter and StreamReader work will with text, but will not handle many characters outside of the Latin character sets. Use **BinaryWriter** and **BinaryReader** to read and write binary data.
- The **StreamReader.ReadBlock** method reads characters into a byte array buffer. It **blocks**, or keeps executing and doesn't return, until it's either read all of the characters you asked for or run out of data to read.
- File.OpenRead returns a FileStream, and the **FileStream.Read** method reads bytes from a stream.
- The **String.Substring** method returns a part of a string. The String class has an **overloaded constructor** that takes a char array as a parameter and converts it to a string.
- C# makes the command-line arguments for a console app available as the args string array that gets passed to the Main method.

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, most things in the game aren't directly controlled by the player. So how do these objects find their way around a scene?

The goal of labs 5 and 6 is to get you familiar with Unity's **pathfinding and navigation system**, a sophisticated AI system that lets you create characters that can find their way around the worlds that you create. In this lab, you'll build a scene out of GameObjects and use navigation to move a character around it.

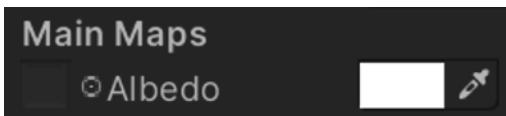
You'll use **raycasting** to write code that's responsive to the geometry of the scene, **capture input**, and use it to move a GameObject to the point where the player clicked. Just as importantly, you'll **get practice writing C# code** with classes, fields, references, and other topics we've discussed.

Create a new Unity project and start to set up the scene

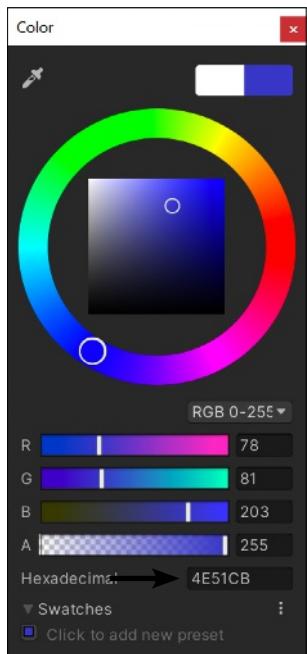
Before you begin, close any Unity project that you have open. Also close Visual Studio—we'll let Unity open it for us. Create a new Unity project using the 3D template, set your layout to Wide so it matches our screenshots, and give it a name like **Unity Labs 5 and 6** so you can come back to it later.

Start by creating a play area that the player will navigate around. Right-click inside the Hierarchy window and **create a Plane** (GameObject >> 3D Object >> Plane). Name your new Plane GameObject *Floor*.

Right-click on the Assets folder in the Project window and **create a folder inside it called Materials**. Then right-click on the new Materials folder you created and choose **Create >> Material**. Call the new material *FloorMaterial*. Let's keep this material simple for now—we'll just make it a color. Select Floor in the Project window, then click on the white box to the right of the word Albedo in the Inspector.

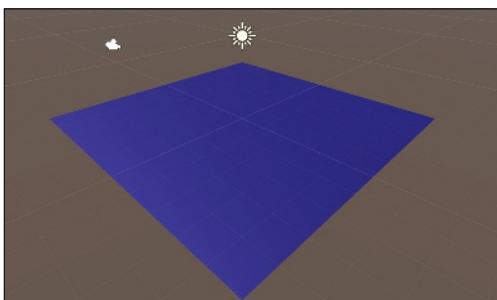


You can use this dropper to grab a color from anywhere on your screen.



In the Color window, use the outer ring to choose a color for the floor. We used a color with number 4E51CB in the screenshot—you can type that into the Hexadecimal box.

Drag the material from the **Project window onto the Plane GameObject in the Hierarchy window**. Your floor plane should now be the color that you selected.



A Plane has no Y dimension. What happens if you give it a large Y scale value? What if the Y scale value is negative? What if it's zero?

Think about it and take a guess. Then use the Inspector window to try various Y scale values and see if the plane acts the way you expected. (Don't forget to set them back!)

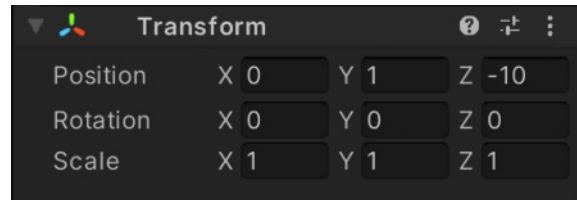
A **Plane** is a flat square object that's 10 units long by 10 units wide (in the X-Z plane), and 0 units tall (in the Y plane). Unity creates it so that the center of the plane is at point (0,0,0). This center point of the plane determines its position in the scene. Just like our other objects, you can move a plane around the scene by using the Inspector or the tools to change its position and rotation. You can also change its scale, but since it has no height, you can only change the X and Z scale—any positive number you put into the Y scale will be ignored.

The objects that you can create using the 3D Object menu (planes, spheres, cubes, cylinders, and a few other basic shapes) are called **primitive objects**. You can learn more about them by opening the Unity Manual from the Help menu and searching for the “**Primitive and placeholder objects**” help page. Take a minute and open up that help page right now. Read what it says about planes, spheres, cubes, and cylinders.

Set up the camera

In the last two Unity Labs you learned that a GameObject is essentially a “container” for components, and that the Main Camera has just three components: a Transform, a Camera, and an Audio Listener. That makes sense, because all a camera really needs to do is be at a location and record what it sees and hears. Have a look at the camera’s Transform component in the Inspector window.

Notice how the position is $(0, 1, -10)$. Click on the Z label in the Position line and drag up and down. You’ll see the camera fly back and forth in the scene window. Take a close look at the box and four lines in front of the camera. They represent the camera’s **viewport**, or the visible area on the player’s screen.



Move the camera around the scene and rotate it using

the Move tool (W) and Rotate tool (E), just like you did with other GameObjects in your scene. The Camera Preview window will update in real time, showing you what the camera sees. Keep an eye on the Camera Preview while you move the camera around. The floor will appear to move as it flies in and out of the camera’s view.

Use the context menu in the Inspector window to reset the Main Camera’s Transform component. Notice how it **doesn’t reset the camera to its original position**—it resets both the camera’s position and its rotation to $(0, 0, 0)$. You’ll see the camera intersecting the plane in the Scene window.

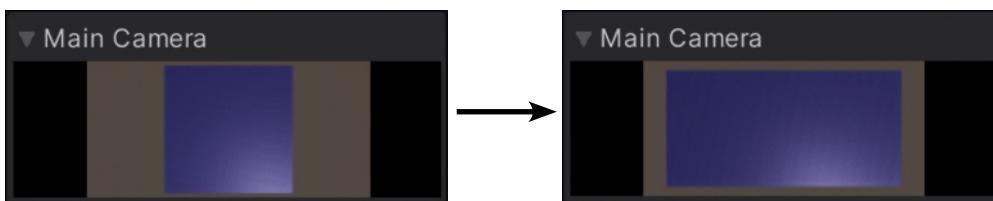


Now let’s point the camera straight down. Start by clicking on the X label next to Rotation and dragging up and down. You’ll see the viewport in the camera preview move. Now **set the camera’s X rotation to 90** in the Inspector window to point it straight down.

You’ll notice that there’s nothing in the Camera Preview anymore, which makes sense because the camera is looking straight down below the infinitely thin plane. **Click on the Y position label in the Transform component and drag up** until you see the entire plane in the Camera Preview.

Now **select Floor in the Hierarchy window**. Notice that the Camera Preview disappears—it only appears when the camera is selected. You can also switch between the Scene and Game windows to see what the camera sees.

Use the Plane’s Transform component in the Inspector window to **set the Floor GameObject’s scale to (4, 1, 2)** so that it’s twice as long as it is wide. Since a Plane is 10 units wide and 10 units long, this scale will make it 40 units long and 20 units wide. The plane will completely fill up the viewport again, so move the Camera further up along the Y axis until the entire plane is in view.



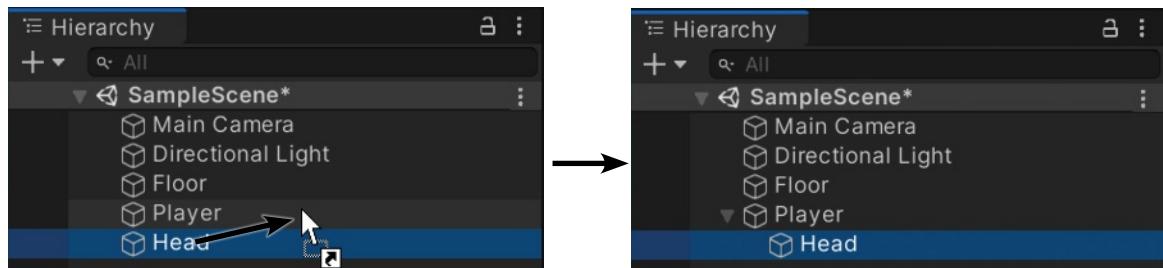
You can
switch
between
the Scene
and Game
windows
to see
what the
camera
sees.

Create a GameObject for the player

Your game will need a player to control. We'll create a simple humanoid-ish player that has a cylinder for a body and a sphere for a head. Make sure you don't have any objects selected by clicking the scene (or the empty space) in the Hierarchy window.

Create a Cylinder GameObject (3D Object >> Cylinder)—you'll see a cylinder appear in the middle of the scene. Change its name to *Player*, then **choose Reset from the context menu** for the Transform component to make sure it has all of its default values. Next, **create a Sphere GameObject** (3D Object >> Sphere). Change its name to *Head*, and reset its Transform component as well. They'll each have a separate line in the Hierarchy window.

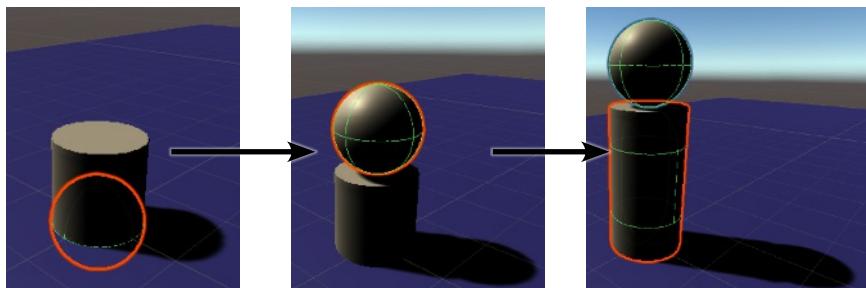
But we don't want separate GameObjects—we want a single GameObject that's controlled by a single C# script. This is why Unity has the concept of **parenting**. Click on Head in the Hierarchy window and **drag it onto Player**. This makes Player the parent of Head. Now the Head GameObject is **nested** under Player.



Select Head in the Hierarchy window. It was created at (0, 0, 0) like all of the other spheres you created. You can see the outline of the sphere, but you can't see the sphere itself because it's hidden by the plane and the cylinder. Use the Transform component in the Inspector window to **change the Y position of the sphere to 1.5**. Now the sphere appears above the cylinder, just the right place for the player's head.

Now select Player in the Hierarchy window. Since its Y position is 0, half of the cylinder is hidden by the plane. **Set its Y position to 1**. The cylinder pops up above the plane. Notice how it took the Head sphere along with it. Moving Player causes Head to move along with it because moving a parent GameObject moves its children too—in fact, *any* change that you make to its Transform component will automatically get applied to the children. If you scale it down, its children will scale, too.

Switch to the Game window—your player is in the middle of the play area.



When you modify the Transform component for a GameObject that has nested children, the children will move, rotate, and scale along with it.

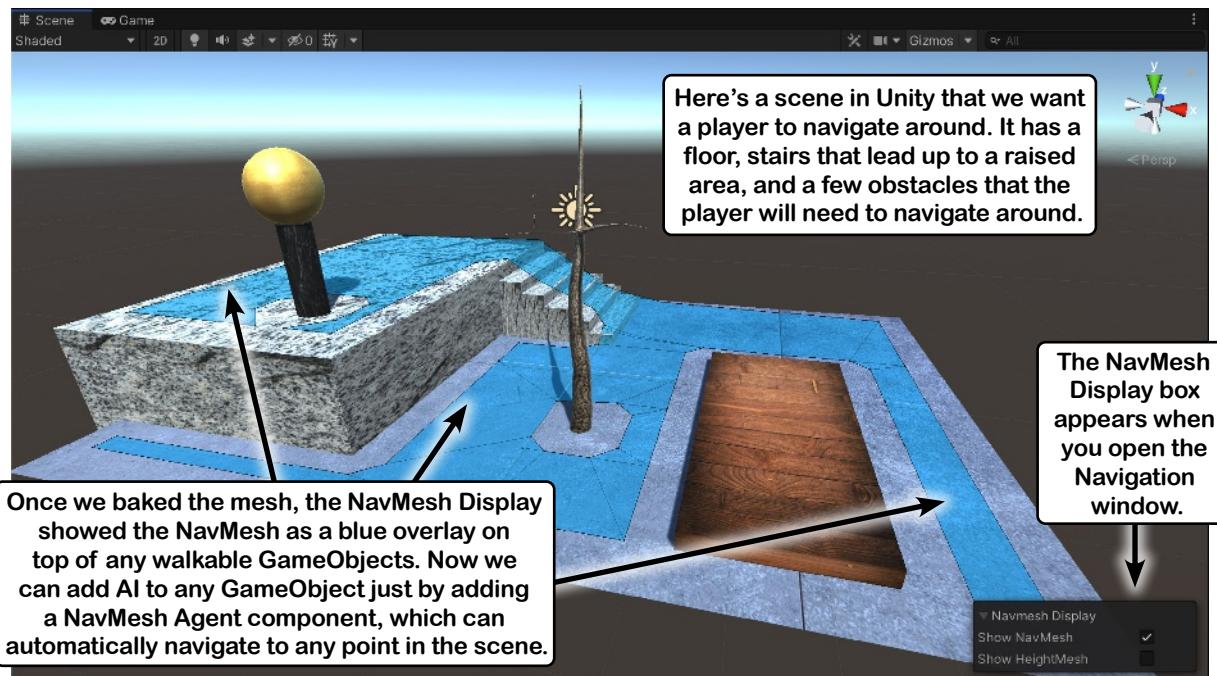
Introducing Unity's navigation system

One of the most basic things that video games do is move things around. Players, enemies, characters, items, obstacles...all of these things can move. That's why Unity is equipped with a sophisticated artificial intelligence-based navigation and pathfinding system to help GameObjects move around your scenes. We'll take advantage of the navigation system to make our player move toward a target.

Unity's navigation and pathfinding system lets your characters intelligently find their way around a game world. To use it, you need to set up basic pieces to tell Unity where the player can go:

- ★ First, you need to tell Unity exactly where your characters are allowed to go. You do this by **setting up a NavMesh**, which contains all of the information about the walkable areas in the scene: slopes, stairs, obstacles, and even points called off-mesh links that let you set up specific player actions like opening a door.
- ★ Second, you **add a NavMesh Agent component** to any GameObject that needs to navigate. This component automatically moves the GameObject around the scene, using its AI to find the most efficient path to a target and avoiding obstacles and, optionally, other NavMesh Agents.
- ★ It can sometimes take a lot of computation for Unity to navigate complex NavMeshes. That's why Unity has a Bake feature, which lets you set up a NavMesh in advance **and precompute (or bake)** the geometric details to make the agents work more efficiently.

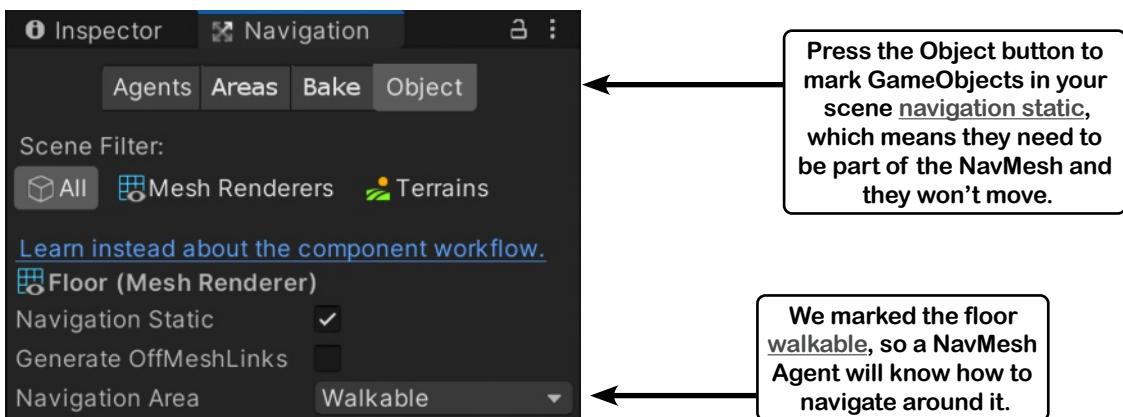
Unity provides a sophisticated AI navigation and pathfinding system that can move your GameObjects around a scene in real time by finding an efficient path that avoids obstacles.



Set up the NavMesh

Let's set up a NavMesh that just consists of the Floor plane. We'll do this using the Navigation window. **Choose AI >> Navigation from the Window menu** to add the Navigation window to your Unity workspace. It should show up as a tab in the same panel as the Inspector window. Then use the Navigation window to **mark the Floor GameObject navigation static and walkable**:

- ★ Press the **Object button** at the top of the Navigation window.
- ★ **Select the Floor plane** in the Hierarchy window.
- ★ Check the **Navigation Static box**. This tells Unity to include the Floor when baking the NavMesh.
- ★ **Select Walkable** from the Navigation Area dropdown. This tells Unity that the Floor plane is a surface that any GameObject with a NavMesh Agent can navigate.



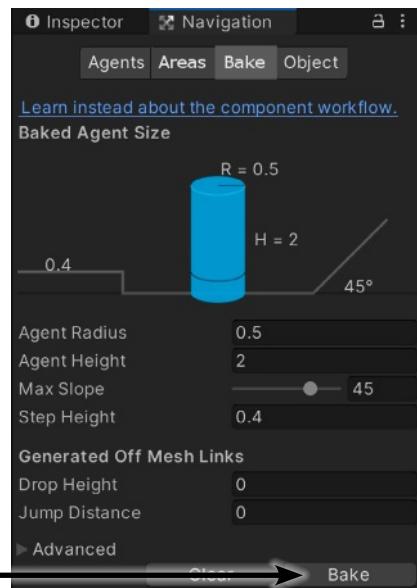
Since the only walkable area in this game will be the floor, we're done in the Object section. For a more complex scene with many walkable surfaces or nonwalkable obstacles, each individual GameObject needs to be marked appropriately.

Click the Bake button at the top of the Navigation window to see the bake options.

Now **click the other Bake button** at the bottom of the Navigation window. It will briefly change to Cancel and then switch back to Bake. Did you notice that something changed in the Scene window? Switch back and forth between the Inspector and Navigation windows. When the Navigation window is active, the Scene window shows the NavMesh Display and highlights the NavMesh as a blue overlay on top of the GameObjects that are part of the baked NavMesh. In this case, it highlights the plane that you marked as navigation static and walkable.

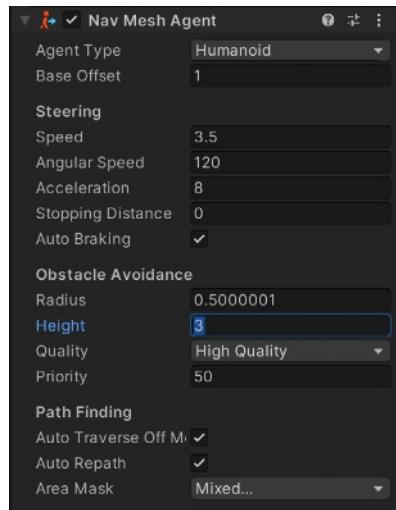
Your NavMesh is now set up.

Click the Bake button to bake the NavMesh.



Make your player automatically navigate the play area

Let's add a NavMesh Agent to your Player GameObject. Select **Player** in the Hierarchy window, then go back to the Inspector window, click the **Add Component** button, and choose **Navigation > NavMesh Agent** to add the NavMesh Agent component. The cylinder body is 2 units tall and the sphere head is 1 unit tall, so you want your agent to be 3 units tall—so set the Height to 3. Now the NavMesh Agent is ready to move the Player GameObject around the NavMesh.



Create a Scripts folder and add a script called *MoveToClick.cs*. This script will let you click on the play area and tells the NavMesh Agent to move the GameObject to that spot. You learned about private fields in Chapter 5. This script will use one to store a reference to the NavMeshAgent. Your GameObject's code will need a reference to its agent so it can tell the agent where to go, so you'll call the GetComponent method to get that reference and save it in a **private NavMeshAgent field** called **agent**:

```
agent = GetComponent<NavMeshAgent>();
```

The navigation system uses classes in the `UnityEngine.AI` namespace, so you'll need to **add this using line** to the top of your *MoveToClick.cs* file:

```
using UnityEngine.AI;
```

Here's the **code for your MoveToClick script**:

```
public class MoveToClick : MonoBehaviour
{
    private NavMeshAgent agent;

    void Awake()
    {
        agent = GetComponent<NavMeshAgent>();
    }

    void Update()
    {
        if (Input.GetMouseButtonUp(0))
        {
            Camera cameraComponent = GameObject.Find("Main Camera").GetComponent<Camera>();
            Ray ray = cameraComponent.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit, 100))
            {
                agent.SetDestination(hit.point);
            }
        }
    }
}
```

In the last Unity Lab, you used the `Start` method to set a GameObject's position when it first appears. There's actually a method that gets called before your script's `Start` method. The `Awake` method is called when the object is created, while `Start` is called when the script is enabled. The *MoveToClick* script uses the `Awake` method to initialize the field, not the `Start` method.

Here's where the script handles mouse clicks. The `Input.GetMouseButtonUp` method checks if the user is currently pressing a mouse button, and the `0` argument tells it to check for the left button. Since `Update` is called every frame, it's always checking to see if the mouse button is clicked.

Experiment with the Speed, Angular Speed, Acceleration, and Stopping Distance fields in the NavMesh agent. You can change them while the game is running (but remember it won't save any values that you change while running the game). What happens when you make some of them really big?

Drag the script onto Player and run the game. While the game is running, **click anywhere on the floor**. When you click on the plane, the NavMesh Agent will move your player to the point that you clicked.

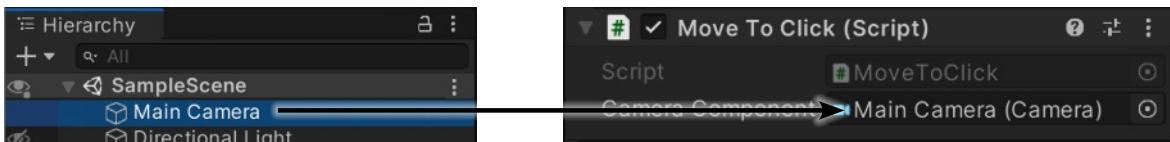


We've talked a lot about object references and reference variables over the last few chapters. Let's do a little pencil-and-paper work to get some of the ideas and concepts behind object references into your brain.

Add this **public** field to the MoveToClick class:

```
public Camera cameraComponent;
```

Go back to the Hierarchy window, click on Player, and find the new Main Camera field in the Move To Click (Script) component. Then **drag the Main Camera** out of the Hierarchy window and **onto the Camera Component field** in the Player GameObject's Move To Click (Script) component in the inspector:



Now **comment out** this line:

```
Camera cameraComponent = GameObject.Find("Main Camera").GetComponent<Camera>();
```

Run your game again. It still works! Why? Think about it, and see if you can figure it out. Write down the answer:

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



MY SCRIPT CALLED METHODS THAT
HAD THE WORD **RAY** IN THE NAME. I USED RAYS
BACK IN THE FIRST UNITY LAB. ARE WE USING RAYS TO HELP
MOVE THE PLAYER?

Yes! We're using a really useful tool called raycasting.

In the second Unity Lab, you used Debug.DrawRay to explore how 3D vectors work by drawing a ray that starts at (0, 0, 0). Your MoveToClick script's Update method actually does something similar to that. It uses the **Physics.Raycast** method to “cast” a ray—just like the one you used to explore vectors—that starts at the camera and goes through the point where the user clicked, and **checks if the ray hit the floor**. If it did, then the Physics.Raycast method provides the location on the floor where it hit. Then the script sets the **NavMesh Agent's destination** field, which causes the NavMesh Agent to **automatically move the player** toward that location.



Your MoveToClick script calls the **Physics.Raycast** method, a really useful tool Unity provides to help your game respond to changes in the scene. It shoots (or “casts”) a virtual ray across your scene and tells you if it hit anything. The Physics.Raycast method’s parameters tell it where to shoot the ray and the maximum distance to shoot it:

Physics.Raycast(where to shoot the ray, out hit, maximum distance)

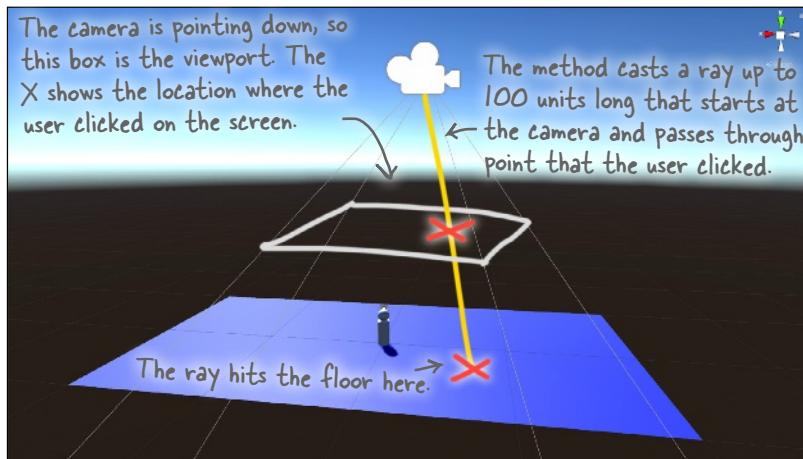
This method returns true if the ray hit something, or false if it didn’t. It uses the **out** keyword to save the results in a variable, exactly like you saw with `int.TryParse` in the last few chapters. Let’s take a closer look at how this works.

We need to tell `Physics.Raycast` where to shoot the ray. So the first thing we need to do is find the camera—specifically, the **Camera** component of the **Main Camera GameObject**. Your code gets it just like you got the **GameController** in the last Unity Lab:

```
GameObject.Find("Main Camera").GetComponent<Camera>();
```

The **Camera** class has a method called **ScreenPointToRay** that creates a ray that shoots from the camera’s position through an (X, Y) position on the screen. The **Input.mousePosition** method provides the (X, Y) position on the screen where the user clicked. This ray provides you with the location to feed into `Physics.Raycast`:

```
Ray ray = cameraComponent.ScreenPointToRay(Input.mousePosition);
```



Now that the method has a ray to cast, it can call the `Physics.Raycast` method to see where it hits:

```
RaycastHit hit;
if (Physics.Raycast(ray, out hit, 100))
{
    agent.SetDestination(hit.point);
}
```

It returns a bool and uses the **out** keyword—in fact, it works exactly like `int.TryParse`. If it returns true, then the **hit** variable contains the location on the floor that the ray hit. Setting **agent.destination** tells the NavMesh Agent to start moving the player toward the point where the ray hit.

Sharpen your pencil Solution

Run your game again. It still works! Why? Think about it, and see if you can figure it out. Write down the answer:

When my code called `mainCamera.GetComponent<Camera>` it returned a reference to a `GameObject`. I replaced

it with a field and dragged the Main Camera `GameObject` from the Hierarchy window into the Inspector window,

which caused the field to be set to a reference to the same `GameObject`. Those were two different ways to set

the `cameraComponent` variable to reference the same object, which is why it behaved the same way.

You'll be reusing the `MoveToClick` script in later Unity Labs, so after you're done writing down the answers, change the script back to the way it was by removing the `MainCamera` field and restoring the line that sets the `cameraComponent` variable.

BULLET POINTS

- A **Plane** is a flat square object that's 10 units by 10 units wide (in the X-Z plane), and 0 units tall (in the Y plane).
- You can **move the Main Camera** to change the part of the scene that it captures by modifying its Transform component, just like you move any other `GameObject`.
- When you modify the Transform component of a `GameObject` that has **nested children**, the children will move, rotate, and scale along with it.
- Unity's **AI navigation and pathfinding system** can move your `GameObjects` around a scene in real time by finding an efficient path that avoids obstacles.
- A **NavMesh** contains all of the information about the walkable areas in the scene. You can set up a NavMesh in advance and pre-compute—or bake—the geometric details to make the agents work more efficiently.
- A **NavMesh Agent** component automatically moves a `GameObject` around the scene, using its AI to find the most efficient path to a target.
- The **NavMeshAgent.SetDestination** method triggers the agent to calculate a path to a new position and start moving toward the new destination.
- Unity calls your script's **Awake** method when it first loads the `GameObject`, well before it calls the script's Start method but after it instantiates other `GameObjects`. It's a great place to initialize references to other `GameObjects`.
- The **Input.GetMouseButtonUp** method returns true if a mouse button is currently being clicked.
- The **Physics.Raycast** method does *raycasting* by shooting (or “casting”) a virtual ray across the scene and returns true if it hit anything. It uses the `out` keyword to return information about what it hit.
- The camera's **ScreenPointToRay** method creates a ray that goes through a point on the screen. Combine it with `Physics.Raycast` to determine where to move the player.

CAPTAIN AMAZING

THE DEATH OF THE OBJECT

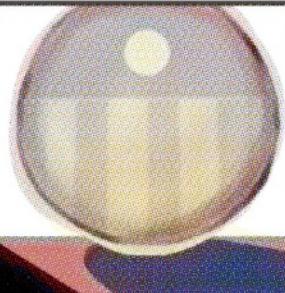
Head First C#

Four
bucks

Chapter
11



CAPTAIN AMAZING, OBJECTVILLE'S MOST AMAZING OBJECT, PURSUES HIS ARCH-NEMESIS...



I'VE GOT YOU NOW,
SWINDLER.

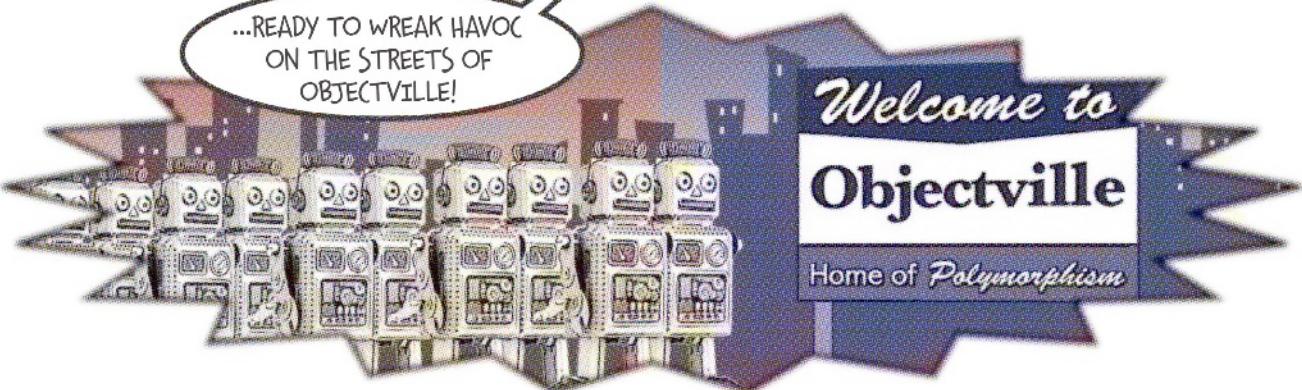
YOU'RE TOO LATE! AS WE SPEAK
MY CLONE ARMY IS GATHERING IN
THE FACTORY BENEATH US...



...READY TO WREAK HAVOC
ON THE STREETS OF
OBJECTVILLE!

Welcome to Objectville

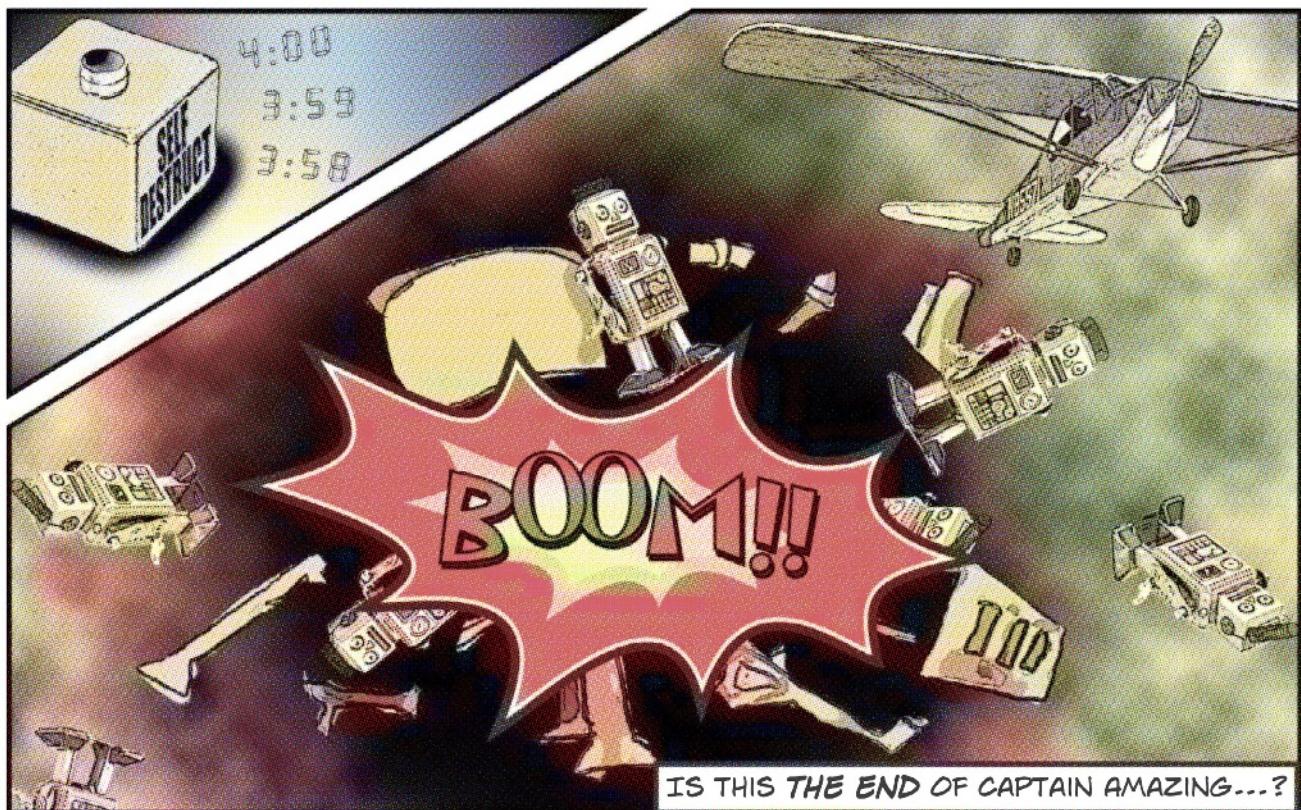
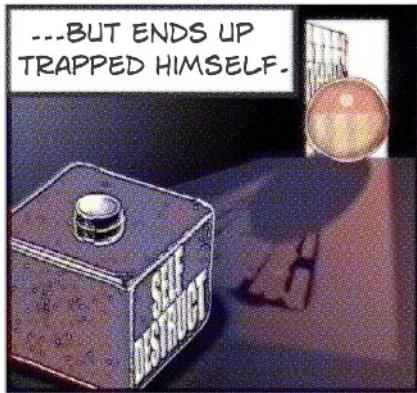
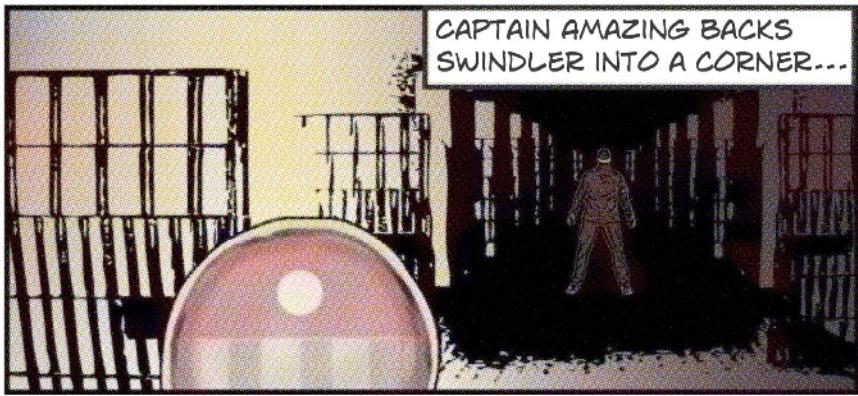
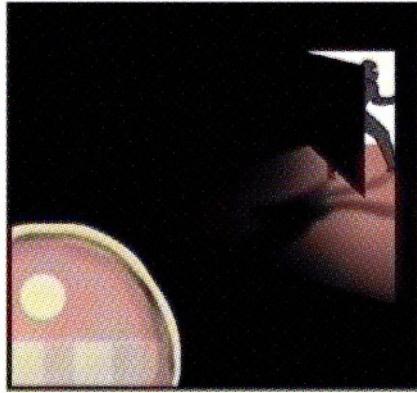
Home of *Polymorphism*



POW!!!

I'LL TAKE DOWN EACH
CLONE'S REFERENCES, ONE
BY ONE.





IS THIS THE END OF CAPTAIN AMAZING...?

The life and death of an object

Here's a quick review of what we know about how objects live and die:

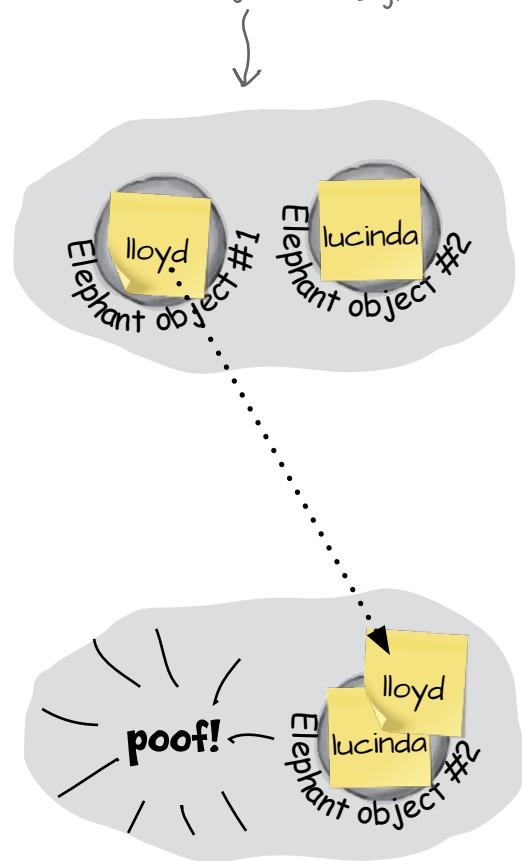
- ★ When you create an object, the CLR—which runs your .NET applications and manages memory—allocates enough memory for it on the heap, a special portion of your computer's memory reserved for objects and their data.
- ★ It's kept "alive" by a reference, which can be stored in a variable, a collection, or a property or field of another object.
- ★ There can be lots of references to the same object—like you saw in Chapter 4, when you pointed the `lloyd` and `lucinda` reference variables to the same instance of Elephant.
- ★ When you took away the last reference to the Elephant object, that caused the CLR to mark it for garbage collection.
- ★ And eventually the CLR removed the Elephant object and the memory was reclaimed so it could be used for new instances of objects that your program would go on to create later.

Now we'll explore all of these points in more detail, writing small programs that help show how garbage collection works.

But before we can start experimenting with garbage collection, we need to take a step back. You learned earlier that objects are "marked" for garbage collection—but that the actual removal of the object can happen at any time (or never!). We'll need a way to know when an object has been garbage-collected, and a way to force that garbage collection to happen. So that's where we'll start.



Here's a picture from back in Chapter 4. You created two Elephant objects on the heap, then you removed the reference to one of them to mark it for garbage collection. But what does that actually mean? What's doing the collecting?



Use the **GC class** (with caution) to force garbage collection

.NET gives you a **GC class** that controls the garbage collector. We'll use its static methods—like `GetTotalMemory`, which returns a long with an *approximate* count of the number of bytes currently *thought* to be allocated on the heap:

```
Console.WriteLine(GC.GetTotalMemory(false));
```

 **class System.GC**

Controls the system garbage collector, a service that automatically reclaims unused memory.

You may be thinking, “Why *approximate*? What does *thought* to be allocated mean? How can the garbage collector not know exactly how much memory is allocated?” That reflects one of the basic rules of garbage collection: you can absolutely, 100% rely on garbage collection, but **there are a lot of unknowns and approximations**.

In this chapter we're going to use a few GC functions:

- ★ `GC.GetTotalMemory` returns the approximate number of bytes currently thought to be allocated on the heap.
- ★ `GC.GetTotalAllocatedBytes` returns the approximate number of bytes that have been allocated since the program was started.
- ★ `GC.Collect` forces the garbage collector to reclaim all unreferenced objects immediately.

There's just one thing about these methods: we're using them for learning and exploration, but unless you **really** know what you're doing, **do not call `GC.Collect` in code for a real project**. The .NET garbage collector is a finely tuned, carefully calibrated piece of engineering. In general, when it comes to determining when to collect objects, it's smarter than we are, and we should trust it to do its job.

there are no
Dumb Questions

Q: I have...questions. What...how do I put this...what exactly *is* Captain Amazing?

A: Captain Amazing is the world's most amazing object, the superhero protector of the innocent citizens of Objectville, and friend to all small animals everywhere.

More specifically, Captain Amazing is an anthropomorphized object, inspired by one of the most important comic book events of the early 21st century that deals with the death of a superhero—specifically, a comic that came out in 2007, when we were working on the first draft of *Head First C#* and looking for a good way to talk about the life and death of an object. We noticed a striking similarity between the shape of the objects in our memory heap diagrams and the shield of a certain famous comic book Captain...and thus, Captain Amazing was born. (If you're not a comic book fan, don't worry—you don't need to know anything about the comic we're making a reference to in order to understand the material in this chapter.)

A normal,
unassuming object.

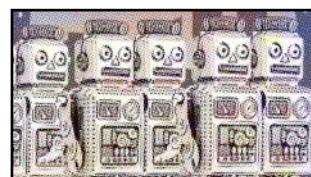


Captain Amazing,
the world's most
amazing object.



Q: Why do your “clones” look like robots? Shouldn’t they be people?

A: Yes, in our comic, we made the clones look like this:



because we didn't want to show graphic depictions of people getting destroyed.

Also, *relax*. The point of the comic panels in this chapter is to help get important C# and .NET concepts into your brain. The story is just a tool to draw some helpful analogies.

Your last chance to DO something... your object's finalizer

Sometimes you need to be sure something happens **before** your object gets garbage-collected, like **releasing unmanaged resources**. ←

A special method in your object called the **finalizer** allows you to write code that will always execute when your object is destroyed. It gets executed last, no matter what.

Let's do some experimentation with finalizers. **Create a new console app** and add this class with a finalizer:

```
class EvilClone
{
    public static int CloneCount = 0;
    public int CloneID { get; } = ++CloneCount;

    public EvilClone() => Console.WriteLine("Clone #{0} is wreaking havoc", CloneID);

    ~EvilClone()
    {
        Console.WriteLine("Clone #{0} destroyed", CloneID);
    }
}
```

When you put the `++` operator in front of a variable, it gets incremented before the statement is run. Why do you think we did that?

This is the **finalizer** (sometimes called a "destructor"). It's declared as a method that starts with a tilde `~` and has no return value or parameters. An object's finalizer is run just before the object is garbage-collected.

The Main method instantiates EvilClone objects, **dereferences** (or removes references to) them, and collects them:

```
class Program
{
    static void Main(string[] args)
    {
        var stopwatch = System.Diagnostics.Stopwatch.StartNew();
        var clones = new List<EvilClone>();
        while (true)
        {
            switch (Console.ReadKey(true).KeyChar)
            {
                case 'a':
                    clones.Add(new EvilClone());
                    break;
                case 'c':
                    Console.WriteLine("Clearing list at time {0}", stopwatch.ElapsedMilliseconds);
                    clones.Clear();
                    break;
                case 'g':
                    Console.WriteLine("Collecting at time {0}", stopwatch.ElapsedMilliseconds);
                    GC.Collect();
                    break;
            }
        }
    }
}
```

When you press the 'a' key your app creates a new instance of EvilClone and adds it to the clones List.

We'll use Stopwatch to get an idea of how fast garbage collection runs. The Stopwatch class lets you accurately measure elapsed time by starting a new stopwatch and getting the number of milliseconds elapsed since you started it.

Pressing 'c' tells the app to clear the List, removing all references to—or dereferencing—all of the clones that you instantiated and added.

Pressing 'g' tells the CLR to collect all objects that have been marked for garbage collection.

Run your app and press **a** a few times to create some EvilClone objects and add them to the List. Then press **c** to clear the List and remove all references to those EvilClone objects. Press **c** a few times—there's a *small* chance the CLR will collect some of the objects that were dereferenced, but you *probably* won't see them collected until you press **g** to call `GC.Collect`.

In general, you'll never write a finalizer for an object that only owns managed resources. Everything you've encountered so far in this book has been managed by the CLR. But occasionally programmers need to access a Windows resource that isn't in a .NET namespace.

For example, if you find code on the internet that has `[DllImport]` above a declaration, you might be using an unmanaged resource. And some of those non-.NET resources might leave your system unstable if they're not "cleaned up." That's what finalizers are for.

When EXACTLY does a finalizer run?

The finalizer for your object runs **after** all references are gone, but **before** that object gets garbage-collected. Garbage collection only happens after **all** references to your object go away, but it doesn't always happen *right after* the last reference disappears.

Suppose you have an object with a reference to it. The CLR sends the garbage collector to work, and it checks out your object. But since there are references to your object, the garbage collector ignores it and moves along. Your object keeps living on in memory.

Then, something happens. That last object holding a reference to *your* object removes that reference. Now your object is sitting in memory, with no references. It can't be accessed. It's basically a **dead object**.

But here's the thing: ***garbage collection is something that the CLR controls***, not your objects. So if the garbage collector isn't sent out again for, say, a few seconds, or maybe even a few minutes, your object still lives on in memory. It's unusable, but it hasn't been garbage-collected. **And the object's finalizer cannot (yet) run.**

Finally, the CLR sends the garbage collector out again. It checks your object, finds there are no references, and runs the finalizer...possibly several minutes after the last reference to the object was removed or changed. Now that it's been finalized, your object is dead, and the collector tosses it away.

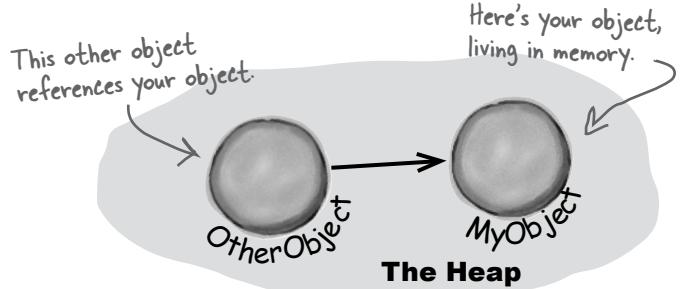
You can SUGGEST to .NET that it's time to collect the garbage

.NET does let you **suggest** that garbage collection would be a good idea. **Most times, you'll never use this method, because garbage collection is tuned to respond to a lot of conditions in the CLR, and calling it isn't really a good idea.** But just to see how a finalizer works, you could call for garbage collection on your own, using GC.Collect.

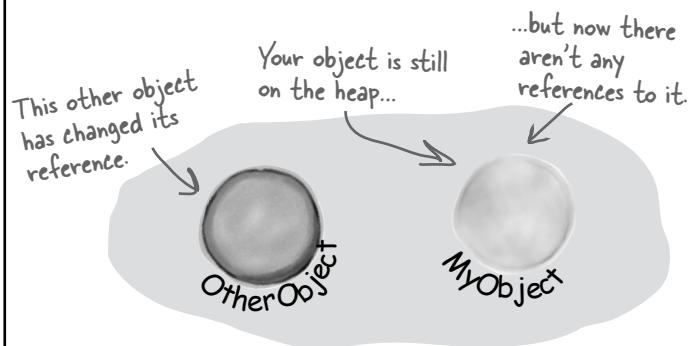
Be careful, though. That method doesn't **force** the CLR to garbage-collect things immediately. It just says, "Do garbage collection as soon as possible."

The life and death of an object...a timeline

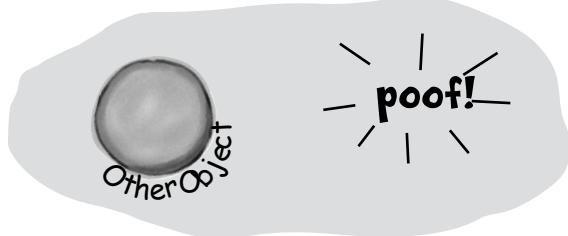
1. Your object is living its best life on the heap. Another object has a reference to it, keeping it alive.



2. The other object changes its reference, so now there are no other objects referencing your object.

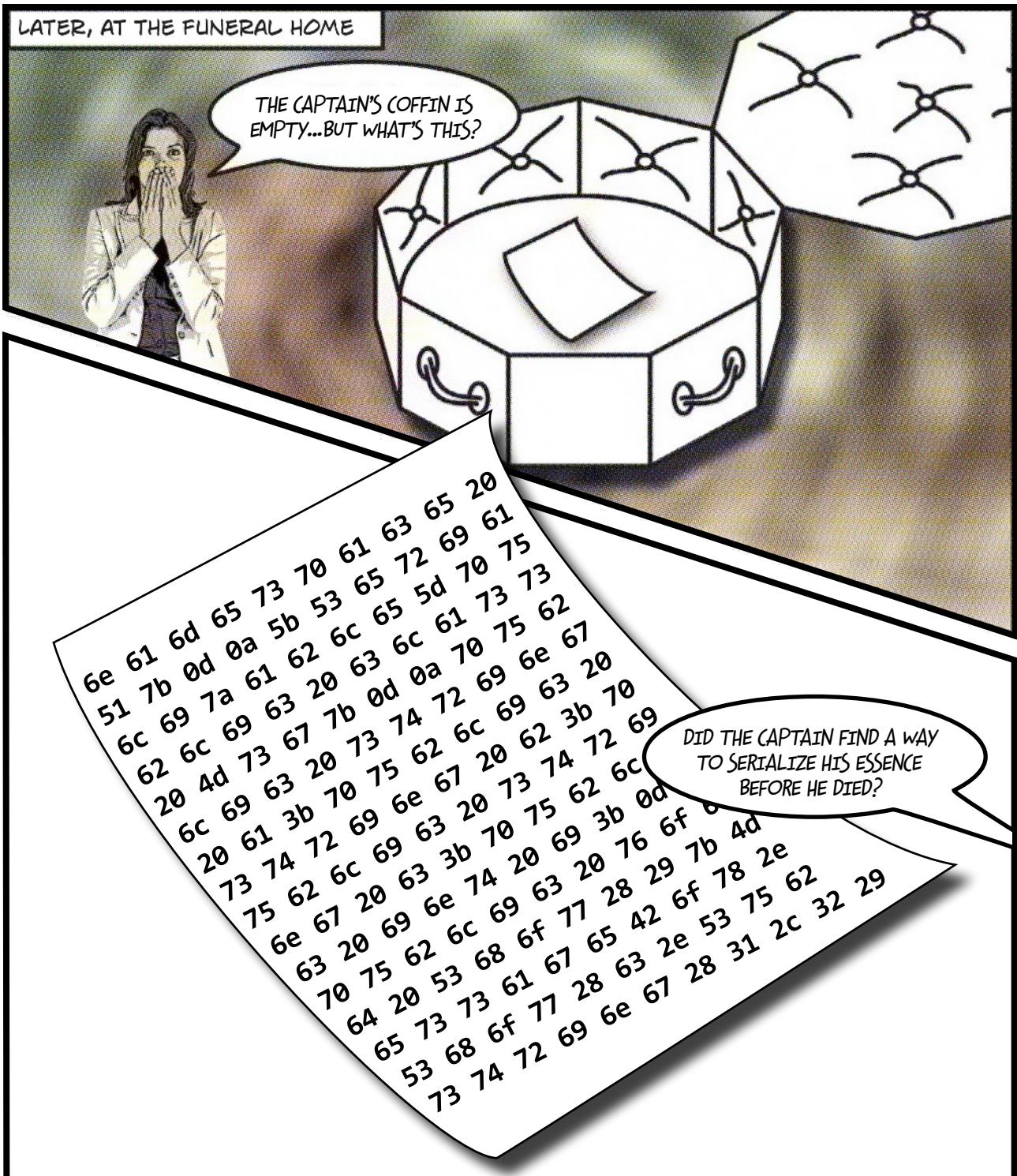


3. The CLR marks your object for garbage collection.



4. Eventually the garbage collector runs the object's finalizer and removes the object from the heap.

We're using GC.Collect as a learning tool to help you understand how **garbage collection** works. You definitely should not use it outside of toy programs (unless you really understand how garbage collection in .NET works on a deeper level than we'll go into in this book).



Finalizers can't depend on other objects

When you write a finalizer, you can't depend on it running at any one time. Even if you call `GC.Collect`, you're only **suggesting** that the garbage collector is run. It's not a guarantee that it'll happen right away. And when it does, you have no way of knowing what order the objects will be collected in.

So what does that mean, in practical terms? Well, think about what happens if you've got two objects that have references to each other. If object #1 is collected first, then object #2's reference to it is pointing to an object that's no longer there. But if object #2 is collected first, then object #1's reference is invalid. That means ***you can't depend on references in your object's finalizer***. Which means that it's a really bad idea to try to do something inside a finalizer that depends on references being valid.

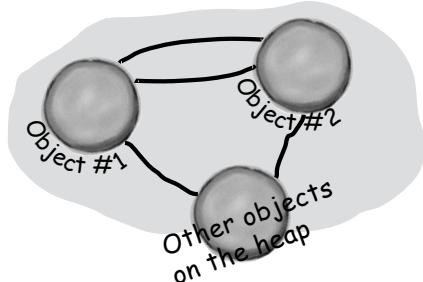
Don't use finalizers for serialization

Serialization is a really good example of something that you **shouldn't do inside a finalizer**. If your object's got a bunch of references to other objects, serialization depends on ***all*** of those objects still being in memory... and all of the objects they reference, and the ones those objects reference, and so on. So if you try to serialize when garbage collection is happening, you could end up **missing** vital parts of your program because some objects might've been collected **before** the finalizer ran.

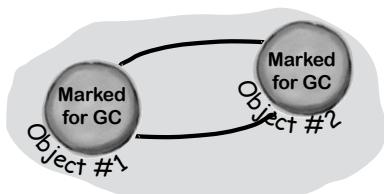
Luckily, C# gives us a really good solution to this: `IDisposable`. Anything that could modify your core data or that depends on other objects being in memory needs to happen as part of a `Dispose` method, not a finalizer.

Some people like to think of a finalizer as a kind of fail-safe for the `Dispose` method. And that makes sense—you saw with your `Clone` object that just because you implement `IDisposable`, that doesn't mean the object's `Dispose` method will get called. But you need to be careful—if your `Dispose` method depends on other objects that are on the heap, then calling `Dispose` from your finalizer can cause trouble. The best way around this is to make sure you **always use a `using` statement** any time you're creating an `IDisposable` object.

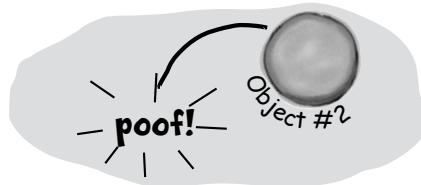
Start with two objects with references to each other.



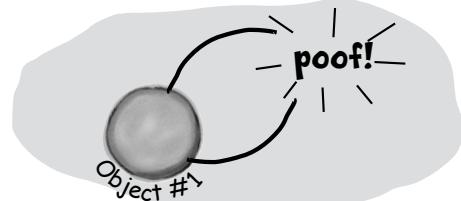
If all other objects on the heap remove their references to objects #1 and #2, they'll both be marked for collection.



If object #1 is collected first, then its data won't be available when the CLR runs object #2's finalizer.



On the other hand, object #2 could disappear before object #1. You've got no way of knowing the order.



And that's why one object's finalizer can't rely on any other object still being on the heap.

Fireside Chats



Tonight's debate: **the Dispose method and a finalizer spar over who's more valuable** to you, a C# developer.

Dispose:

To be honest, I'm a little surprised I was invited here. I thought the programming world had come to a consensus. I mean, I'm simply far more valuable as a C# tool than you are. Really, you're pretty feeble. You can't even depend on other objects still being around by the time that you're called. Pretty unstable, aren't you?

There's an interface specifically **because** I'm so important. In fact, I'm the only method in it!

OK, you're right, programmers need to know they're going to need me and either call me directly or use a **using** statement to call me. But they always know when I'll run, and they can use me to do whatever they need to do to clean up after their objects. I'm powerful, reliable, and easy to use. I'm a triple threat. And you? Nobody knows exactly when you'll run or what the state of the app will be when you finally do decide to show up.

You think you're a big shot because you always run with GC, but at least I can depend on other objects.

Finalizer:

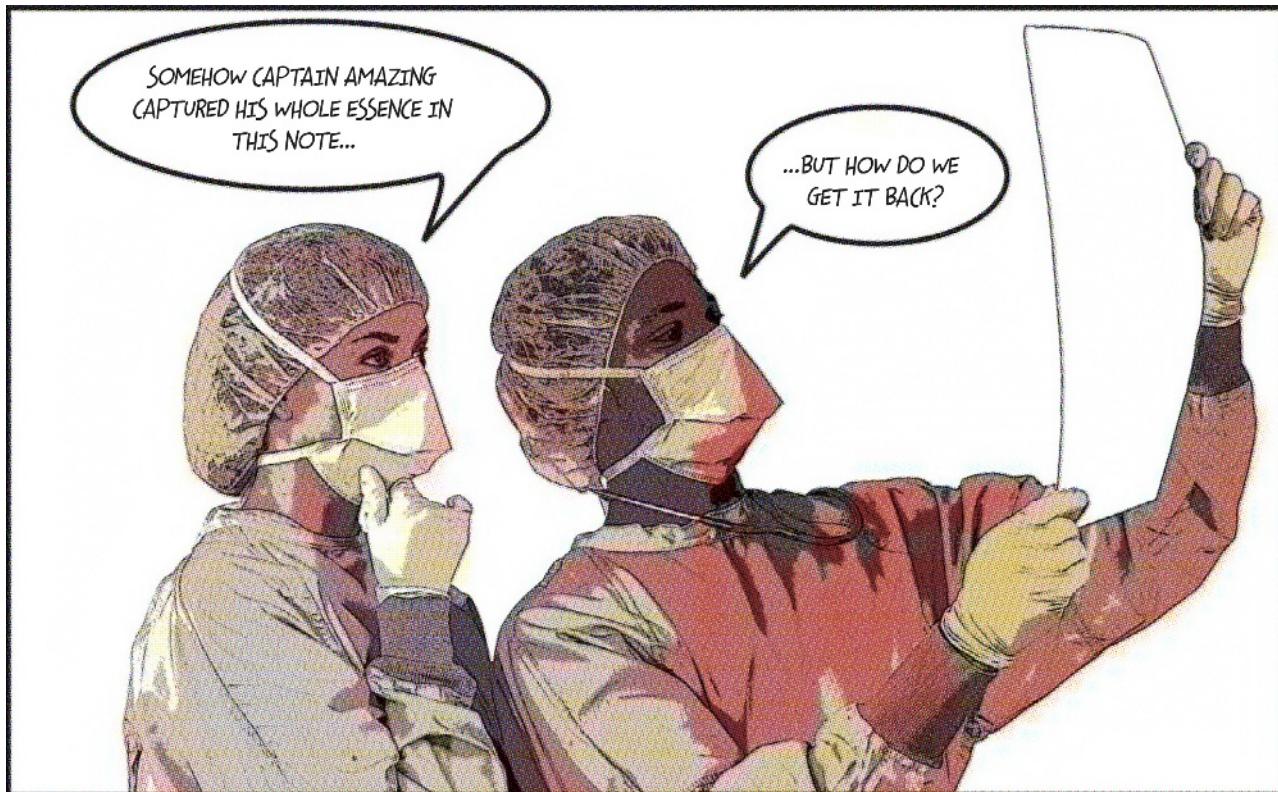
Excuse me? That's rich. I'm "feeble"? OK. Well, I didn't want to get into this, but since we're already stooping this low...at least I don't need an interface to get started. Without the `IDisposable` interface, well, let's face it...you're just another useless method.

Right, right...keep telling yourself that. And what happens when someone forgets to use a **using** statement when they instantiate their object? Then you're nowhere to be found.

Handles are what your programs use when they go around .NET and the CLR and interact directly with Windows. Since .NET doesn't know about them, it can't clean them up for you.

But if you need to do something at the very last moment just before an object is garbage-collected, there's no way to do it without me. I can free up network resources and Windows handles and anything else that might cause a problem for the rest of the app if you don't clean it up. I can make sure that your objects deal with being trashed more gracefully, and that's nothing to sneeze at.

That's right, pal—but I always run. You need someone else to run you. I don't need anyone or anything!



there are no Dumb Questions

Q: Can a finalizer use all of an object's fields and methods?

A: Yes. While you can't pass parameters to a finalizer method, you can use any of the fields in an object, either directly or using `this`—but be careful, because if those fields reference other objects, then the other objects may have already been garbage-collected. So your finalizer can call other methods and properties in the object...as long as those methods and properties *don't depend on other objects*.

Q: What happens to exceptions that get thrown in a finalizer?

A: Good question. Exceptions in finalizers work exactly like they do anywhere else in your code. Try replacing your `EvilClone` finalizer with this one that throws an exception:

```
~EvilClone() => throw new Exception();
```

Then run your app again, create some `EvilClone` instances, clear the list, and run the garbage collector. Your app will halt in the finalizer, just like it would if it hit any other exceptions. (Spoiler alert: in the next chapter you'll learn about how to *catch* exceptions, detecting when they happen and running code to handle them.)

Q: How often does the garbage collector run automatically?

A: The short answer is: we're not sure. Garbage collection doesn't run on an easily predictable cycle, and you don't have firm control over it. You can be sure it will run when your program exits normally. Even when you call `GC.Collect` (which you generally should avoid), you're only suggesting that the CLR should start garbage collection.

Q: So how soon after I call `GC.Collect` will collection start?

A: When you run `GC.Collect`, you're telling .NET to garbage-collect as soon as possible. That's *usually* as soon as .NET finishes whatever it's doing. That means it'll happen pretty soon, but you can't precisely control when it starts.

Q: If something absolutely must run, do I put it in a finalizer?

A: No. It's possible that your finalizer won't run. And it's possible to suppress finalizers when garbage collection happens. Or the process could end entirely. If you aren't freeing unmanaged resources, you're almost always better off using `IDisposable` and `using` statements.

MEANWHILE, ON THE STREETS OF OBJECTVILLE...

CAPTAIN AMAZING...
HE'S BACK!

BUT SOMETHING'S WRONG. HE
DOESN'T SEEM THE SAME...AND
HIS POWERS ARE WEIRD.

CAPTAIN AMAZING TOOK SO LONG
TO GET HERE THAT MR FLUFFY
RESCUED HIMSELF FROM THE TREE...

LATER...

MEOW!

EVEN LATER...

Captain Amazing's Hideout Collection **TOP SECRET**

PUFF...PANT...UGH!
I'M EXHAUSTED.

WHAT'S WRONG? WHY
ARE THE CAPTAIN'S
POWERS BEHAVING
DIFFERENTLY? IS
THIS THE END?

A struct looks like an object...

We've been talking about the heap, because that's where your objects live. But that's not the only part of memory where objects live. One of the types in .NET we haven't talked about much is the *struct*, and we'll use it to explore a different aspect of life and death in C#. Struct is short for **structure**, and structs look a lot like objects. They have fields and properties, just like objects. And you can even pass them into a method that takes an object type parameter:

```
public struct AlmostSuperhero : IDisposable {
    private bool superStrength;
    public int SuperSpeed { get; private set; }

    public void RemoveVillain(Villain villain)
    {
        Console.WriteLine("OK, {0}, surrender now!", villain.Name);
        if (villain.Surrendered)
            villain.GoToJail();
        else
            villain.StartEpicBattle();
    }

    public void Dispose() => Console.WriteLine("Noooooooo!");
}
```

Structs can implement interfaces but can't subclass other classes. And structs are sealed, so they can't be extended.

← A struct can have properties and fields...

...and define methods.

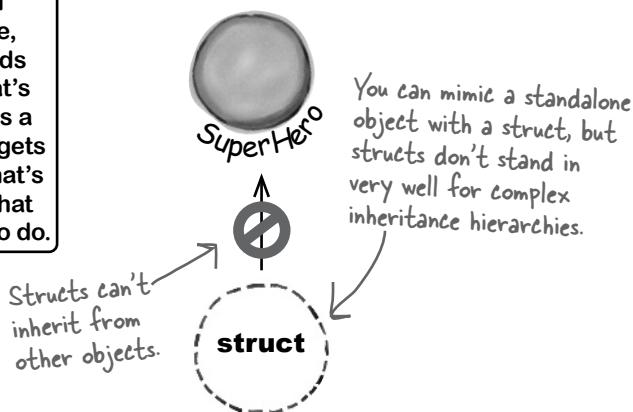
Structs can even implement interfaces like `IDisposable`.

...but isn't an object

But structs **aren't** objects. They *can* have methods and fields, but they *can't* have finalizers. They also can't inherit from classes or other structs, or have classes or structs inherit from them—you're allowed to use the `:` colon operator in a struct's declaration, but only if it's followed by one or more interfaces.

The power of objects lies in their ability to mimic real-world behavior, through inheritance and polymorphism.

All structs extend `System.ValueType`, which in turn extends `System.Object`. That's why every struct has a `ToString` method—it gets it from `Object`. But that's the only inheriting that structs are allowed to do.



Structs are best used for storing data, but the lack of inheritance and references can be a serious limitation.

Values get copied; references get assigned

We've seen how important references are to garbage collection—reassign the last reference to an object, and it gets marked for collection. But we also know that those rules don't quite make sense for values. If we want to get a better sense of how objects and values live and die in the CLR's memory, we'll need to take a closer look at values and references: how they're similar, and more importantly, how they're different.

You already have a sense of how some types are different from others. On the one hand you've got **value types** like int, bool, and decimal. On the other hand, you've got **objects** like List, Stream, and Exception. And they don't quite work exactly the same way, do they?

When you use the equals sign to set one value type variable to another, it **makes a copy of the value**, and afterward, the two variables aren't connected to each other. On the other hand, when you use the equals sign with references, what you're doing is **pointing both references at the same object**.



Variable declaration and assignment work the same with value types and object types:

```
int howMany = 25;
bool Scary = true;
List<double> temps = new List<double>();
throw new NotImplementedException();
```

int and bool are value types, List and Exception are object types.



But once you start assigning values, you can see how they're different. Value types all are handled with copying. Here's an example—this should be familiar stuff:

Changing the fifteenMore variable has no effect on howMany, and vice versa.

```
int fifteenMore = howMany;
fifteenMore += 15;
Console.WriteLine("howMany has {0}, fifteenMore has {1}",
    howMany, fifteenMore);
```

This line copies the value that's stored in the fifteenMore variable into the howMany variable and adds 15 to it.

The output here shows that fifteenMore and howMany are **not** connected:

howMany has 25, fifteenMore has 40



But as we know, when it comes to objects you're assigning references, not values:

This line sets the copy reference to point to the same object as the temps reference.

```
temps.Add(56.5D);
temps.Add(27.4D);
List<double> copy = temps;
copy.Add(62.9D);
```

Both references point at the same actual object.



So changing the List means both references see the update, since they both point to a single List object. Check this by writing a line of output:

```
Console.WriteLine("temps has {0}, copy has {1}", temps.Count(), copy.Count());
```

The output here demonstrates that copy and temps are actually pointing to the **same** object:

temps has 3, copy has 3

When you called copy.Add, it added a new temperature to the object that both copy and temps point to.

Structs are value types; objects are reference types

Let's take a closer look at how structs work, so you can start to understand when you might want to use a struct versus an object. When you create a struct, you're creating a **value type**. What that means is when you use equals to set one struct variable equal to another, you're creating a fresh *copy* of the struct in the new variable. So even though a struct *looks* like an object, it doesn't act like one.

← Do this!

1 Create a struct called Dog.

Here's a simple struct to keep track of a dog. It looks just like an object, but it's not. Add it to a **new console application**:

```
public struct Dog {

    public string Name { get; set; }
    public string Breed { get; set; }

    public Dog(string name, string breed) {
        this.Name = name;
        this.Breed = breed;
    }

    public void Speak() {
        Console.WriteLine("My name is {0} and I'm a {1}.", Name, Breed);
    }
}
```

2 Create a class called Canine.

Make an exact copy of the Dog struct, except **replace struct with class** and then **replace Dog with Canine**. Don't forget to rename Dog's constructor. Now you'll have a Canine *class* that you can play with, which is almost exactly equivalent to the Dog *struct*.

3 Add a Main method that makes some copies of Dog and Canine data.

Here's the code for the Main method:

```
Canine spot = new Canine("Spot", "pug");
Canine bob = spot;
bob.Name = "Spike";
bob.Breed = "beagle";
spot.Speak();
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
betty.Name = "Betty";
betty.Breed = "pit bull";
jake.Speak();
```



4 Before you run the program...

Write down what you think will be written to the console when you run this code:

.....
.....



What did you think would get written to the console?

My name is Spike and I'm a beagle.
My name is Jake and I'm a poodle.

Here's what happened...

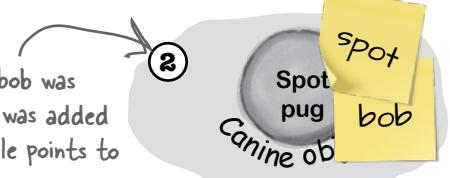
The `bob` and `spot` references both point to the same object, so both changed the same fields and accessed the same `Speak` method. But structs don't work that way. When you created `betty`, you made a fresh copy of the data in `jake`. The two structs are completely independent of each other.

```
Canine spot = new Canine("Spot", "pug"); ①
Canine bob = spot; ②
bob.Name = "Spike";
bob.Breed = "beagle"; ③
spot.Speak();
```

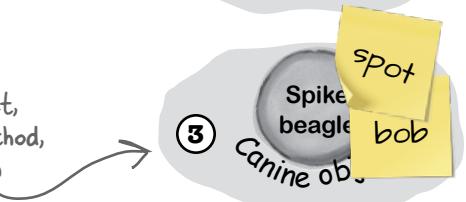
A new `Canine` object was created and the `spot` reference points to it.



The new reference variable `bob` was created, but no new object was added to the heap—the `bob` variable points to the same object as `spot`.

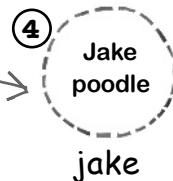


Since `spot` and `bob` both point to the same object, `spot.Speak` and `bob.Speak` both call the same method, and both of them produce the same output with "Spike" and "beagle".



```
Dog jake = new Dog("Jake", "poodle"); ④
Dog betty = jake; ⑤
betty.Name = "Betty";
betty.Breed = "pit bull";
jake.Speak();
```

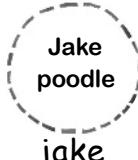
When you create a new struct, it looks really similar to creating an object—you've got a variable that you can use to access its fields and methods.



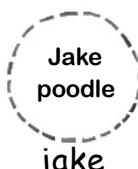
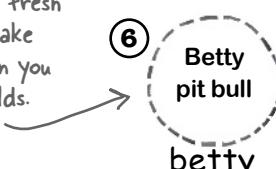
When you set one struct equal to another, you're creating a fresh COPY of the data inside the struct.

That's because struct is a VALUE TYPE (not an object or reference type).

Here's the big difference. When you added the `betty` variable, you created a whole new value.



Since you created a fresh copy of the data, `jake` was unaffected when you changed `betty`'s fields.



The stack vs. the heap: more on memory

Let's quickly recap how a struct differs from an object. You've seen that you can make a fresh copy of a struct just using equals, which you can't do with an object. But what's really going on behind the scenes?

The CLR divides your data between two places in memory: the heap and the stack. You already know that objects live on the **heap**. The CLR also reserves another part of memory called the **stack**, where it stores the local variables you declare in your methods and the parameters that you pass into those methods. You can think of the stack as a bunch of slots that you can stick values in. When a method gets called, the CLR adds more slots to the top of the stack. When it returns, its slots are removed.

The Code

Here's some code you might see in a program.

```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
```

Here's what the stack looks like after these two lines of code run.

```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
```

When you create a new struct—or any other value-type variable—a new “slot” gets added onto the stack. That slot is a copy of the value in your type.

```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
SpeakThreeTimes(jake);
```

```
public SpeakThreeTimes(Dog dog) {
    int i;
    for (i = 0; i < 5; i++)
        dog.Speak();
}
```

When you call a method, the CLR puts its local variables on the top of the stack. In this case, the code calls the `SpeakThreeTimes` method. It has one parameter (`dog`) and one variable (`i`), and the CLR stores them on the stack.

When the method returns, the CLR pops `i` and `dog` off the stack—and that's how values live and die in the CLR.

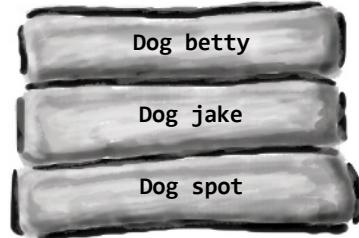
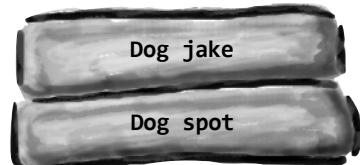
Behind the Scenes



Remember, when your program's running, the CLR is actively managing memory, dealing with the heap, and collecting garbage.

The Stack

This is where structs and local variables hang out.





THIS ALL SOUNDS THEORETICAL, BUT I BET THERE'S A GOOD PRACTICAL REASON TO LEARN ABOUT THE STACK, THE HEAP, VALUES, AND REFERENCES.

It's important to understand how a struct you copy by value is different from an object you copy by reference.

There are times when you need to be able to write a method that can take either a value type *or* a reference type—perhaps a method that can work with either a Dog struct or a Canine object. If you find yourself in that situation, you can use the `object` keyword:

```
public void WalkDogOrCanine(object getsWalked) { ... }
```

You can also use the “`is`” keyword to see if an object is a struct, or any other value type, that’s been boxed and put on the heap.

①

Here’s what the stack and heap look like after you create an object variable and set it equal to a Dog struct.

```
Dog sid = new Dog("Sid", "husky");
WalkDogOrCanine(sid);
```



After a struct is boxed, there are two copies of the data: the copy on the stack, and the copy boxed on the heap.

The `WalkDogOrCanine` method takes an object reference, so the Dog struct was boxed before it was passed in. Casting it back to a Dog unboxes it.

②

If you want to **unbox the object**, all you need to do is cast it to the right type, and it gets unboxed automatically. The `is` keyword works just fine with structs, but be careful, because the `as` keyword doesn’t work with value types.

```
if (getsWalked is Dog doggo) doggo.Speak();
```



After this line runs, you’ve got a third copy of the data in a new struct called `doggo`, which gets its own slot on the stack.

Behind the Scenes



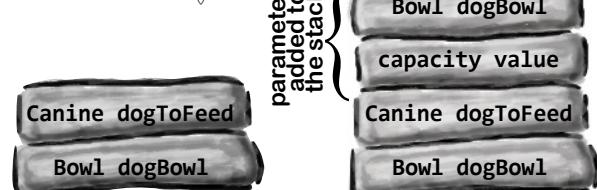
When a method is called, it looks for its arguments on the stack.

The stack plays an important part in how the CLR manages your app's data. One thing we take for granted is the fact that we can write a method that calls another method, which in turn calls another method. In fact, a method can call itself (which is called *recursion*). The stack is what gives our programs the ability to do that.

Here are three methods from a dog simulator program. The FeedDog method calls the Eat method, which calls the CheckBowl method.

Remember the terminology here: a parameter specifies the values a method needs; an argument is the actual value or reference that you pass into a method when you call it.

Here's what the stack looks like as FeedDog calls Eat, which calls CheckBowl, which in turn calls Console.WriteLine().

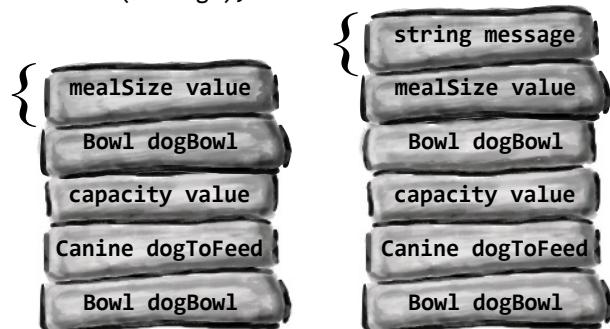


The FeedDog method takes two parameters, a Canine reference and a Bowl reference. So when it's called, the two arguments passed to it are on the stack.

```
public double FeedDog(Canine dogToFeed, Bowl dogBowl) {
    double eaten = Eat(dogToFeed.MealSize, dogBowl);
    return eaten + .05D; // A little is always spilled
}

public void Eat(double mealSize, Bowl dogBowl) {
    dogBowl.Capacity -= mealSize;
    CheckBowl(dogBowl.Capacity);
}

public void CheckBowl(double capacity) {
    if (capacity < 12.5D) {
        string message = "My bowl's almost empty!";
        Console.WriteLine(message);
    }
}
```



FeedDog needs to pass two arguments to the Eat method, so they're pushed onto the stack as well.

As the method calls pile up and the program goes deeper into methods that call methods that call other methods, the stack gets bigger and bigger.

When WriteLine exits, its arguments will be popped off of the stack. That way, Eat can keep going as if nothing had happened.

`var v = Vector3.zero;`

struct UnityEngine.Vector3

Representation of 3D vectors and points.

Since Unity's Vector3 is a struct, creating a bunch of vectors won't cause extra GCs.

Go to a Unity project and hover over Vector3—it's a struct. Garbage collection (or GC) can seriously slow down an app's performance, and a lot of object instances in your game could trigger extra GCs and slow down the frame rate. Games often use a LOT of vectors. Making them structs means their data is kept on the stack, so even creating millions of vectors won't cause extra GCs that will slow down your game.

Use out parameters to make a method return more than one value

Speaking of parameters and arguments, there are a few more ways that you can get values into and out of your programs. They all involve adding **modifiers** to your method declarations. One of the most common ways of doing this is by using the **out modifier** to specify an output parameter. You've seen the **out** modifier many times—you use it every time you call the `int.TryParse` method. You can also use the **out** modifier in your own methods. Create a new console app and add this empty method declaration to the form. Note the **out** modifiers on both parameters:

```
public static int ReturnThreeValues(int value, out double half, out int twice)
{
    return value + 1;
}
```

The out parameter 'half' must be assigned to before control leaves the current method
The out parameter 'twice' must be assigned to before control leaves the current method
Show potential fixes (Alt+Enter or Ctrl+.)

Do this!

A method can return more than one value by using **out** parameters.

Take a closer look at those two errors:

- ★ The out parameter 'half' must be assigned to before control leaves the current method
- ★ The out parameter 'twice' must be assigned to before control leaves the current method

Any time you use an **out** parameter, you **always** need to set it before the method returns—just like you always need to use a `return` statement if your method is declared with a return value.

Here's all of the code for the app::

```
public static int ReturnThreeValues(int value, out double half, out int twice)
{
    half = value / 2f; } All out parameters
    twice = value * 2; } must be assigned
    return value + 1; before the method
} exits.

static void Main(string[] args)
{
    Console.Write("Enter a number: ");
    if (int.TryParse(Console.ReadLine(), out int input))
    {
        var output1 = ReturnThreeValues(input, out double output2, out int output3);

        Console.WriteLine("Outputs: plus one = {0}, half = {1:F}, twice = {2}",
            output1, output2, output3);
    }
}
```

Here's the familiar code you've used throughout the book that uses the **out** modifier with `int.TryParse` to convert a string to an int.

You'll also use the **out** modifier when you call your new method.

Here's what it looks like when you run the app:

```
Enter a number: 17
Outputs: plus one = 18, half = 8.50, twice = 34
```

Pass by reference using the ref modifier

One thing you've seen over and over again is that every time you pass an int, double, struct, or any other value type into a method, you're passing a copy of that value to that method. There's a name for that: **pass by value**, which means that the entire value of the argument is copied.

But there's another way to pass arguments into methods, and it's called **pass by reference**. You can use the **ref** keyword to allow a method to work directly with the argument that's passed to it. Just like the **out** modifier, you need to use **ref** when you declare the method and also when you call it. It doesn't matter if it's a value type or a reference type, either—any variable that you pass to a method's **ref** parameter will be directly altered by that method.

To see how it works, create a new console app with this Guy class and these methods:

```
class Guy
{
    public string Name { get; set; }
    public int Age { get; set; }
    public override string ToString() => $"a {Age}-year-old named {Name}";
}

class Program
{
    static void ModifyAnIntAndGuy(ref int valueRef, ref Guy guyRef)
    {
        valueRef += 10;
        guyRef.Name = "Bob";
        guyRef.Age = 37;
    }

    static void Main(string[] args)
    {
        var i = 1;
        var guy = new Guy() { Name = "Joe", Age = 26 };
        Console.WriteLine("i is {0} and guy is {1}", i, guy);
        ModifyAnIntAndGuy(ref i, ref guy);
        Console.WriteLine("Now i is {0} and guy is {1}", i, guy);
    }
}
```

Run your app—it writes this output the console:

```
i is 1 and guy is My name is Joe
Now i is 11 and guy is My name is Bob
```

When this method sets `valueRef` and `guyRef`, what it's really doing is changing the values of the variables in the method that called it.

When the Main method calls `ModifyAnIntAndGuy`, it passes its `i` and `guy` variables by reference. The method uses them like any other variable. But because they were passed by reference, the method was actually updating the original variables all along, and not just copies of them on the stack. When the method exits, the `i` and `guy` variables in the Main method are updated directly.

The second line is different from the first because the `ModifyAnIntAndGuy` modified references to the variables in the Main method.

Value types have a TryParse method that uses out parameters

You've been using `int.TryParse` to convert strings to int values ("parsing" means analyzing text and extracting values). Other value types have similar functions: `double.TryParse` will attempt to convert strings to double values, `bool.TryParse` will do the same for Boolean values, and so on for `decimal.TryParse`, `float.TryParse`, `long.TryParse`, `byte.TryParse`, and more. And remember back in Chapter 10 when we used a switch statement to convert the string "Spades" into a `Suits.Spades` enum value? Well, the static `Enum.TryParse` method does the same thing, except for enums.

Use optional parameters to set default values

A lot of times, your methods will be called with the same arguments over and over again, but the method still needs the parameter because occasionally it changes. It would be useful if you could set a *default value*, so you only needed to specify the argument when calling the method if it was different.

That's exactly what **optional parameters** do. You can specify an optional parameter in a method declaration by using an equals sign followed by the default value for that parameter. You can have as many optional parameters as you want, but all of the optional parameters have to come after the required parameters.

Here's an example of a method that uses optional parameters to check if someone has a fever:

```
static void CheckTemperature(double temp, double tooHigh = 99.5, double tooLow = 96.5)
{
    if (temp < tooHigh && temp > tooLow)
        Console.WriteLine("{0} degrees F - feeling good!", temp);
    else
        Console.WriteLine("Uh-oh {0} degrees F -- better see a doctor!", temp);
}
```

Optional parameters have default values specified in the declaration.

This method has two optional parameters: `tooHigh` has a default value of 99.5, and `tooLow` has a default value of 96.5. Calling `CheckTemperature` with one argument uses the default values for both `tooHigh` and `tooLow`. If you call it with two arguments, it will use the second argument for the value of `tooHigh`, but still use the default value for `tooLow`. You can specify all three arguments to pass values for all three parameters.

If you want to use some (but not all) of the default values, you can use **named arguments** to pass values for just those parameters that you want to pass. All you need to do is give the name of each parameter followed by a colon and its values. If you use more than one named argument, make sure you separate them with commas, just like any other arguments.

Add the `CheckTemperature` method to a console app, then add this Main method:

```
static void Main(string[] args)
{
    // Those values are fine for your average person
    CheckTemperature(101.3);

    // A dog's temperature should be between 100.5 and 102.5 Fahrenheit
    CheckTemperature(101.3, 102.5, 100.5);

    // Bob's temperature is always a little low, so set tooLow to 95.5
    CheckTemperature(96.2, tooLow: 95.5);
}
```

It prints this output, working differently based on different values for the optional parameters:

```
Uh-oh 101.3 degrees F -- better see a doctor!
101.3 degrees F - feeling good!
96.2 degrees F - feeling good!
```

Use optional parameters and named arguments when you want your methods to have default values.

A null reference doesn't refer to any object

Do this!

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 anything. Let's experiment with null references.

- Create a new console app and add the Guy class you used to experiment with the `ref` keyword.

- Then **add this code** that creates a new Guy object but *doesn't set its Name property*:

```
static void Main(string[] args)
{
    Guy guy;
    guy = new Guy() { Age = 25 };
    Console.WriteLine("guy.Name is {0} letters long", guy.Name.Length);
}
```

- Place a breakpoint** on the last line of the Main method, then debug your app. When it hits the breakpoint, **hover over guy** to inspect its property values:

```
7 static void Main(string[] args)
8 {
9     Guy guy;
10    guy = new Guy() { Age = 25 };
11    Console.WriteLine("guy.Name is {0} letters long", guy.Name.Length);
12 }
13
14
```

String is a reference type. Since you didn't set its value in the Guy object, it still has its default value: null.

- Continue running the code.** `Console.WriteLine` tries to access the `Length` property of the String object referenced by the `guy.Name` property, and throws an exception:

```
Console.WriteLine("guy.Name is {0} letters long", guy.Name.Length);
```

When the CLR throws a **NullReferenceException** (which developers often refer to as an **NRE**) it's telling you that it tried to access a member of an object, but the reference that it used to access that member was **null**. Developers try to prevent null reference exceptions.

Exception Thrown

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

NullTEst.Program.Guy.Name.get returned null.

[View Details](#) | [Copy Details](#) | [Start Live Share session...](#)



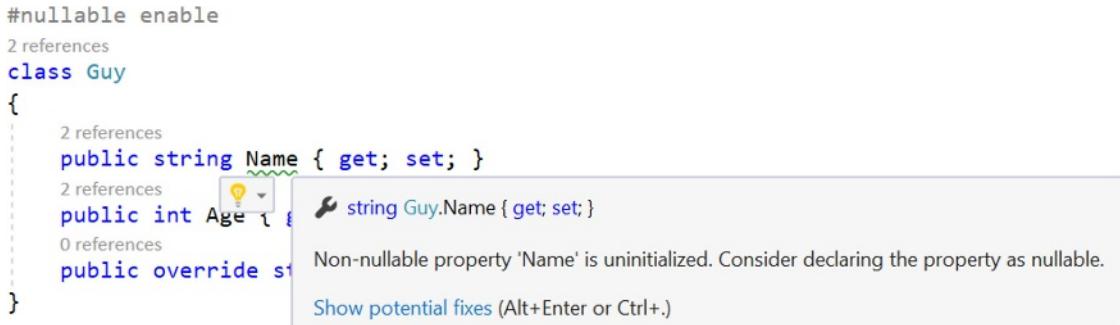
Can you think of some ways to prevent null reference exceptions?

Non-nullable reference types help you avoid NREs

The easiest way to avoid null reference exceptions (or NREs) is to **design your code so references can't be null**. Luckily, the C# compiler gives you a really useful tool to help deal with nulls. Add the following code to the top of your Guy class—it can be either inside or outside the namespace declaration:

```
#nullable enable
```

A line that starts with # is a **directive**, or a way to tell the compiler to set specific options. In this case, it's telling the compiler to treat any reference as a **non-nullable reference type**. As soon as you add the directive, Visual Studio draws a warning squiggle under the Name property. Hover over the property to see the warning:



The C# compiler did something really interesting: it used *flow analysis* (or a way of analyzing the various paths through the code) to determine that **it's possible for the Name property to be assigned a null value**. That means your code could potentially throw an NRE.

You can get rid of the warning by forcing the Name property to be a **nullable reference type**. You can do this by adding a ? character after the type:

`public string? Name { get; set; } ← You can get rid of the warning by making the Name property nullable, but that doesn't really solve the problem.`

But while that gets rid of the error message, it doesn't actually prevent any exceptions.

Use encapsulation to prevent your property from ever being null

Back in Chapter 5 you learned all about how to use encapsulation to keep your class members from having invalid values. So go ahead and make the Name property private, then add a constructor to set its value:

```
class Guy
{
    public string Name { get; private set; }
    public int Age { get; private set; }
    public override string ToString() => $"a {Age}-year-old named {Name}";

    public Guy(int age, string name)
    {
        Age = age;
        Name = name;
    }
}
```

Making the Name property's setter private and adding a constructor forces it to always be assigned. That means it will never be null, which causes the "non-nullable property" compiler warning to disappear.

Once you encapsulate the Name property, you prevent it from ever being set to null, and the warning disappears.

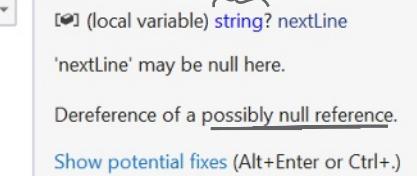
The null-coalescing operator ?? helps with nulls

Sometimes you can't avoid working with nulls. For example, you learned about reading data from strings using `StringReader` in Chapter 10. Create a new console app and add this code:

```
#nullable enable
class Program
{
    static void Main(string[] args)
    {
        using (var stringReader = new StringReader(""))
        {
            var nextLine = stringReader.ReadLine();
            Console.WriteLine("Line length is: {0}", nextLine.Length);
        }
    }
}
```

We've enabled non-nullable types, and now the C# compiler is telling us that `nextLine` is a nullable string and could be null when its `Length` property is accessed.

Run the code—you'll get an NRE. What can we do about it?



?? checks for null and returns an alternative

One way to prevent a null reference from being accessed (or **dereferenced**) is to use the **null-coalescing operator ??** to evaluate the potentially null expression—in this case, calling `stringReader.ReadLine`—and returning an alternative value if it's null. Modify the first line of the `using` block to add `?? String.Empty` to the end of the line:

```
var nextLine = stringReader.ReadLine() ?? String.Empty;
```

`String.Empty` is a static field on the `String` class that returns the empty string "".

And as soon as you add this, the warning goes away. That's because the null coalescing operator tells the C# compiler to execute `stringReader.ReadLine`; and use the value it returns if it's non-null, but substitute the value you provided (in this case, an empty string) if it is.

??= assigns a value to a variable only if it's null

When you're working with null values, it's really common to write code that checks if a value is null and assigns it a non-null value to avoid an NRE. For example, if you wanted to modify your program to print the first line of code, you might write this:

```
if (nextLine == null)
    nextLine = "(the first line is null)";

// Code that works with nextLine and needs it to be non-null
```

You can rewrite that conditional statement using the **null assignment ??=** operator:

```
nextLine ??= "(the first line was empty);
```

The `??=` operator checks the variable, property, or field on the left side of the expression (in this case, `nextLine`) to see if it's null. If it is, the operator assigns the value on the right side of the expression to it. If not, it leaves the value intact.

Nullable value types can be null...and handled safely

When you declare an int, bool, or another value type, if you don't specify a value the CLR assigns it a default value like 0 or true. But let's say you're writing code to store data from a survey where there's a yes/no question that's optional. What if you need to represent a Boolean value that could be true or false, or not have a value at all?

That's where **nullable value types** can be very useful. A nullable value type can either have a value or be set to null. It takes advantage of a generic struct Nullable<T> that can be used to **wrap** a value (or contain the value and provide members to access and work with it). If you set a nullable value type to null, it doesn't have a value—and Nullable<T> gives you handy members to let you work safely with it *even in this case*.

You can declare a nullable Boolean value like this:

```
Nullable<bool> optionalYesNoAnswer = null;
```

C# also has a shortcut—for a value type T, you can declare Nullable<T> like this: **T?**.

```
bool? anotherYesNoAnswer = false;
```

The Nullable<T> struct has a property called Value that gets or sets the value. A bool? will have a value of type bool, an int? will have one of type int, etc. They'll also have a property called HasValue that returns true if it's not null.

You can always convert a value type to a nullable type:

```
int? myNullableInt = 9321;
```

And you can get the value back using its handy Value property:

```
int = myNullableInt.Value;
```

Nullable<bool>
Value: DateTime
HasValue: bool
...
GetValueOrDefault(): DateTime
...

Nullable<T> is a struct that lets you store a value type OR a null value. Here are some of the methods and properties of Nullable<bool>.

But the Value call eventually just casts the value with (int)myNullableInt—and it will throw an InvalidOperationException if the value is null. That's why Nullable<T> also has a HasValue property, which returns true if the value is not null, and false if it is. You can also use the convenient GetValueOrDefault method, which safely returns a default value if the Nullable has no value. You can optionally pass it a default value to use, or use the type's normal default value.

T? is an alias for Nullable<T>

When you add a question mark to any value type (like int? or decimal?), the compiler translates that to the Nullable<T> struct (Nullable<int> or Nullable<decimal>). You can see this for yourself: add a Nullable<bool> variable to a program, put a breakpoint on it, and add a watch for it in the debugger. You'll see bool? displayed in the Watch window in the IDE. This is an example of an alias, and it's not the first one you've encountered. Hover your cursor over any int. You'll see that it translates to a struct called System.Int32:

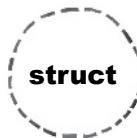
int.Parse() and int.TryParse() are members of this struct. →

```
int value = 3;  
readonly struct System.Int32  
Represents a 32-bit signed integer.
```

Take a minute and do that for each of the types at the beginning of Chapter 4. Notice how all of them are aliases for structs—except for string, which is a class called System.String, not a value type.

"Captain" Amazing...not so much

You should have a pretty good idea by now of what was going on with the less-powerful, more-tired Captain Amazing. In fact, it wasn't Captain Amazing at all, but a boxed struct:



vs.



★ **Structs can't inherit from classes.**

No wonder the Captain's superpowers seemed a little weak! He didn't get any inherited behavior.

★ **Structs are copied by value.**

This is one of the most useful things about them. It's especially useful for encapsulation.

One important point: you can use the "is" keyword to check if a struct implements an interface, which is one aspect of polymorphism that structs do support.

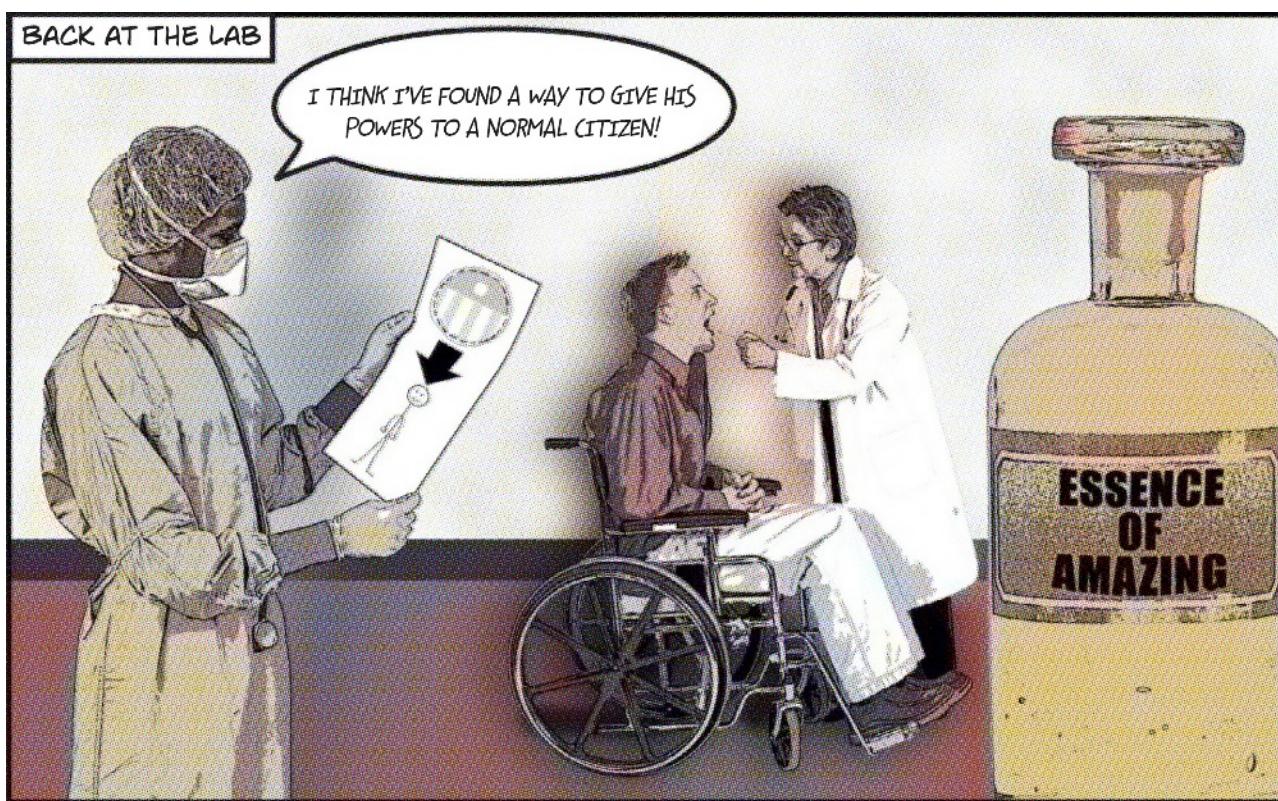
★ **You can use the as keyword with an object.**

Objects allow for polymorphism by allowing an object to function as any of the objects it inherits from.

★ **You can't create a fresh copy of an object.**

When you set one object variable equal to another, you're copying a **reference** to the **same** variable.

That's one advantage of structs (and other value types)—you can easily make copies of them.



there are no Dumb Questions

Q: OK, back up a minute. Why do I care about the stack?

A: Because understanding the difference between the stack and the heap helps you keep your reference types and value types straight. It's easy to forget that structs and objects work very differently—when you use the equals sign with both of them, they look really similar. Having some idea of how .NET and the CLR handle things under the hood helps you understand *why* reference and value types are different.

Q: And boxing? Why is that important to me?

A: Because you need to know when things end up on the stack, and you need to know when data's being copied back and forth. Boxing takes extra memory and more time. When you're only doing it a few times (or a few hundred times) in your program, then you won't notice the difference. But let's say you're writing a program that does the same thing over and over again, millions of times a second. That's not too far-fetched: your Unity games might do exactly that. If you find that your program's taking up more and more memory, or going slower and slower, then it's possible that you can make it more efficient by avoiding boxing in the part of the program that repeats.

Q: I get how you get a fresh copy of a struct when you set one struct variable equal to another one. But why is that useful to me?

A: One place that's really helpful is with **encapsulation**. Take a look at this code:

```
private Point location;  
public Point Location {  
    get { return location; }  
}
```

If Point were a class, then this would be terrible encapsulation. It wouldn't matter that location is private, because you made a public read-only property that returns a reference to it—so any other object would be able to access it.

Lucky for us, Point happens to be a struct. And that means that the public Location property returns a fresh copy of the point. The object that uses it can do whatever it wants to that copy—none of those changes will make it to the private location field.

Q: How do I know whether to use a struct or a class?

A: Most of the time, programmers use classes. Structs have a lot of limitations that can really make it hard to work with them for large jobs. They don't support inheritance or abstraction, and only limited polymorphism, and you already know how important those things are for writing code.

Where structs come in really handy is if you have a small, limited type of data that you need to work with repeatedly. Unity vectors are good examples—some games will use them over and over again, possibly millions of times. Reusing a vector by assigning it to the same variable reuses that same memory on the stack. If Vector3 were a class, then the CLR would have to allocate new memory on the heap for each new Vector3, and it would constantly be garbage-collecting. So by making Vector3 a struct and not a class, the team that develops Unity gave you the gift of higher frame rates—without you having to do a thing.

A struct can be valuable for encapsulation, because a read-only property that returns a struct always makes a fresh copy of it.



This method is supposed to kill an EvilClone object by marking it for GC, but it doesn't work. Why not?

```
void SetCloneToNull(EvilClone clone) => clone = null;
```

Pool Puzzle



Your **job** is to take snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make the code write the output shown below to the console when this app is executed.

```
class Program {
    static void Main(string[] args) =>
        new Faucet();
}

public class Faucet {
    public Faucet() {
        Table wine = new Table();
        Hinge book = new Hinge();
        wine.Set(book);
        book.Set(wine);
        wine.Lamp(10);
        book.garden.Lamp("back in");
        book.bulb *= 2;
        wine.Lamp("minutes");
        wine.Lamp(book);
    }
}
```

```
public _____ Table {
    public string stairs;
    public Hinge floor;

    public void Set(Hinge b) => floor = b;

    public void Lamp(object oil) {
        if (oil _____ int oilInt)
            _____ .bulb = oilInt;
        else if (oil _____ string oilString)
            stairs = oilString;
        else if (oil _____ Hinge _____ )
            Console.WriteLine(
                $"{vine.Table()} {_____ .bulb} {stairs}");
    }
}

public _____ Hinge {
    public int bulb;
    public Table garden;

    public void Set(Table a) => garden = a;

    public string Table() {
        return _____ .stairs;
    }
}
```

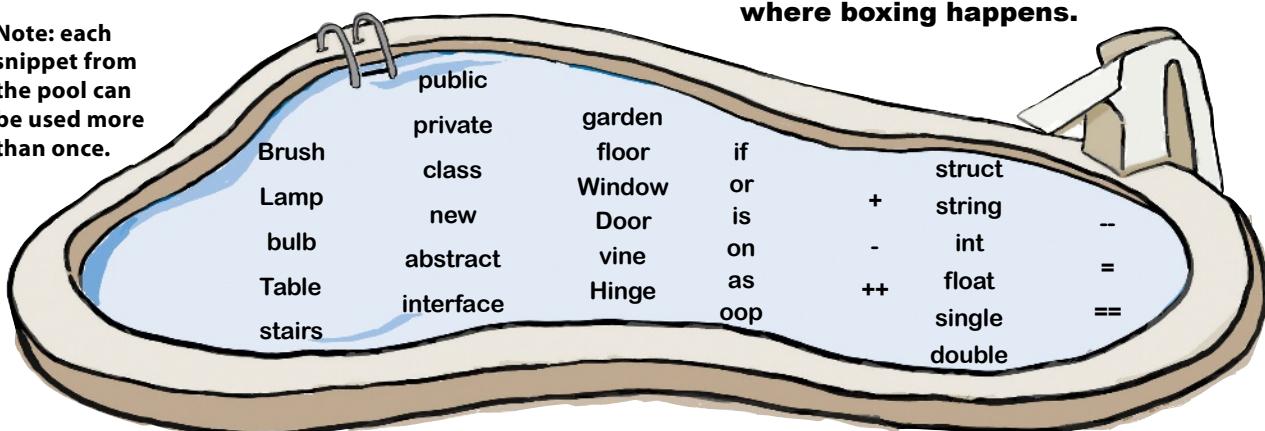
output

Here's the output from the app.

→ back in 20 minutes

Bonus puzzle: Circle the lines where boxing happens.

Note: each snippet from the pool can be used more than once.



Pool Puzzle Solution



The Lamp method sets the various strings and ints. If you call it with an int, then it sets the Bulb field in whatever object Hinge points to.

```
class Program {
    static void Main(string[] args) =>
        new Faucet();
}

public class Faucet {
    public Faucet() {
        Table wine = new Table();
        Hinge book = new Hinge();
        wine.Set(book);
        book.Set(wine);
        wine.Lamp(10);
        book.garden.Lamp("back in");
        book.bulb *= 2;
        wine.Lamp("minutes");
        wine.Lamp(book);
    }
}
```

Here's why Table has to be a struct. If it were a class, then wine would point to the same object as book.Garden, which would cause this to overwrite the "back in" string.

Bonus puzzle: Circle the lines where boxing happens.



The clone parameter is just on the stack, so setting it to null doesn't do anything to the heap.

All this method does is set its own parameter to null, but that parameter's just a reference to an EvilClone. It's like sticking a label on an object and peeling it off again.

```
public struct Table {
    public string stairs;
    public Hinge floor;

    public void Set(Hinge b) => floor = b;

    public void Lamp(object oil) {
        if (oil is int oilInt)
            floor.bulb = oilInt;
        else if (oil is string oilString)
            stairs = oilString;
        else if (oil is Hinge vine)
            Console.WriteLine(
                $"{vine.Table()} {floor.bulb} {stairs}");
    }
}

public class Hinge {
    public int bulb;
    public Table garden;

    public void Set(Table a) => garden = a;

    public string Table() {
        return garden.stairs;
    }
}
```

If you pass a string to Lamp, it sets the Stairs field to whatever is in that string.

output
back in 20 minutes

Both Hinge and Table have expression-bodied methods called Set. Hinge.Set sets its Table field called Garden, while Table.Set sets its Hinge field called Floor.

Since the Lamp method takes an object parameter, boxing automatically happens when it's passed an int or a string. But book isn't boxed, because it's already an object—an instance of class Hinge.

Extension methods add new behavior to EXISTING classes

Remember the sealed modifier from Chapter 7? It's how you set up a class that can't be extended.

Sometimes you need to extend a class that you can't inherit from, like a sealed class (a lot of the .NET classes are sealed, so you can't inherit from them). And C# gives you a flexible tool for that: **extension methods**. When you add a class with extension methods to your project, it **adds new methods that appear on classes** that already exist. All you have to do is create a static class, and add a static method that accepts an instance of the class as its first parameter using the `this` keyword.

So let's say you've got a sealed OrdinaryHuman class:

```
sealed class OrdinaryHuman {
    private int age;
    int weight;

    public OrdinaryHuman(int weight) {
        this.weight = weight;
    }

    public void GoToWork() { /* code to go to work */ }
    public void PayBills() { /* code to pay bills */ }
}
```

The OrdinaryHuman class is sealed, so it can't be subclassed. But what if we want to add a method to it?

You use an extension method by specifying the first parameter using the "this" keyword.

Since we want to extend the OrdinaryHuman class, we make the first parameter "this OrdinaryHuman".

The AmazeballsSerum method adds an extension method to OrdinaryHuman:

```
static class AmazeballsSerum {
    public static string BreakWalls(this OrdinaryHuman h, double wallDensity) {
        return $"I broke through a wall of {wallDensity} density.";
    }
}
```

Extension methods are always static methods, and they have to live in static classes.

As soon as the AmazeballsSerum class is added to the project, OrdinaryHuman gets a BreakWalls method. So now your Main method can use it:

```
static void Main(string[] args){
    OrdinaryHuman steve = new OrdinaryHuman(185);
    Console.WriteLine(steve.BreakWalls(89.2));
}
```

And that's it! All you need to do is add the AmazeballsSerum class to your project, and suddenly every OrdinaryHuman class gets a brand-new BreakWalls method.

When the program creates an instance of the OrdinaryHuman class, it can access the BreakWalls method directly—as long as the AmazeballsSerum class is in the project. Go ahead, try it out! [Create a new console application](#) and add the two classes and the Main method to it. Debug into the BreakWalls method and see what's going on.

Hmm...earlier in the book we "magically" added methods to classes just by adding a using directive to the top of our code. Do you remember where that was?

there are no
Dumb Questions

Q: Tell me again why I wouldn't add the new methods I need directly to my class code, instead of using extensions?

A: You could do that, and you probably should if you're just talking about adding a method to one class. Extension methods should be used pretty sparingly, and only in cases where you absolutely can't change the class you're working with for some reason (like it's part of the .NET Framework or another third party). Where extension methods really become powerful is when you need to extend the behavior of something you **wouldn't normally have access to**, like a type or an object that comes for free with the .NET Framework or another library.

Q: Why use extension methods at all? Why not just extend the class with inheritance?

A: If you can extend the class, then you'll usually end up doing that—extension methods aren't meant to be a replacement for inheritance. But they come in really handy when you've got classes that you can't extend. With extension methods, you can change the behavior of whole groups of objects, and even add functionality to some of the most basic classes in the .NET Framework.

Extending a class gives you new behavior, but requires that you use the new subclass if you want to use that new behavior.

Q: Does my extension method affect all instances of a class, or just a certain instance of the class?

A: It will affect all instances of a class that you extend. In fact, once you've created an extension method, the new method will show up in your IDE alongside the extended class's normal methods.



One more point to remember about extension methods: you don't gain access to any of the class's internals by creating an extension method, so it's still acting as an outsider.

WHEN I ADDED "USING SYSTEM.LINQ;" TO THE TOP OF MY CODE, ALL OF MY COLLECTIONS AND SEQUENCES SUDDENLY GOT LINQ METHODS ADDED TO THEM. WAS I USING EXTENSION METHODS?



Yes! LINQ is based on extension methods.

In addition to extending classes, you can also extend **interfaces**. All you have to do is use an interface name in place of the class, after the `this` keyword in the extension method's first parameter. The extension method will be added to **every class that implements that interface**. And that's exactly what the .NET team did when they created LINQ—all of the LINQ methods are static extension methods for the `IEnumerable<T>` interface.

Here's how it works. When you add `using System.Linq;` to the top of your code, it causes your code to “see” a static class called `System.Linq.Enumerable`. You've used some of its methods, like `Enumerable.Range`, but it also has extension methods. Go to the IDE and type `Enumerable.First`, then look at the declaration. It starts with `(extension)` to tell you that it's an extension method, and its first parameter uses the `this` keyword just like the extension method you wrote. You'll see the same pattern for every LINQ method.

`Enumerable.First`

`Enumerable.First` is an extension method that uses the “`this`” keyword in its declaration.



`(extension) TSource Enumerable.First<TSource>(this IEnumerable<TSource> source) (+ 1 generic overload)`

Returns the first element of a sequence.



This button in the IntelliSense window causes it to show only extension methods.

Extending a fundamental type: string

Let's explore how extension methods work by extending the String class.

Create a new Console App project, and add a file called *HumanExtensions.cs*.

Do this!

1

Put all of your extension methods in a separate namespace.

Make sure your class is public so it's visible when you add the using declaration. It's a good idea to keep all of your extensions in a different namespace than the rest of your code. That way, you won't have trouble finding them for use in other programs. Set up a static class for your method to live in, too:

```
namespace AmazingExtensions {
    public static class ExtendAHuman {
```

Using a separate namespace is a good organizational tool. And the class your extension method is defined in must be static.

2

Create the static extension method, and define its first parameter as this and then the type you're extending.

The two main things you need to know when you declare an extension method are that the method needs to be static and it takes the class it's extending as its first parameter:

The extension method must be static, too.

```
public static bool IsDistressCall(this string s) {
```

"this string s" says we're extending the String class and uses the parameter s to access the string being used to call the method.

3

Finish the extension method.

This method checks the string to see if it contains the word "Help!"—if it does, then that string is a distress call, which every superhero is sworn to answer:

```
if (s.Contains("Help!"))
    return true;
else
    return false;
}
```

This uses the String.Contains method to see if the string contains the word "Help!"—and that's definitely not something an ordinary string normally does.

4

Use your new IsDistressCall extension method.

Add `using AmazingExtensions;` to the top of the file with your Program class. Then add code to the class that creates a string and calls its `IsDistressCall` method. You'll see your extension in the IntelliSense window:

```
0 references
static void Main(string[] args)
{
    string message = "Evil clones are wreaking havoc. Help!";
    message.IsDistressCall
}
```

As soon as you add the using directive to add the namespace with your static class, the IntelliSense window contains your new extension method. This is exactly how LINQ works.



Extension Magnets

Arrange the magnets to produce this output:

a buck begets more bucks

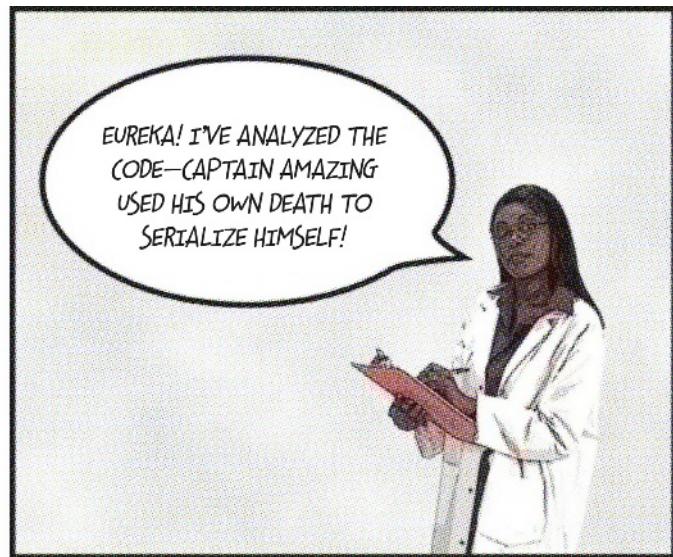
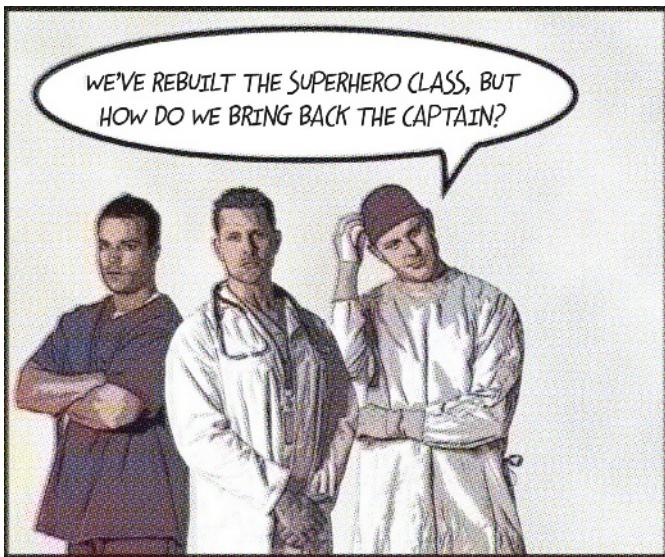
```
namespace Upside {  
    public static class Margin {  
        public static void SendIt  
    }  
    public static string ToPrice
```

```
namespace Sideways  
{  
    using Upside;  
    class Program {
```

```
    }  
}  
  
static void Main(string[] args) {  
    s.SendIt();  
    if (n == 1)  
        return "a buck ";  
    else  
        return " more bucks";  
    int i = 1;  
    string s = i.ToPrice();  
    Console.WriteLine(s);  
    bool b = true;
```

```
    (this int n) {  
        b.Green().SendIt();  
        i = 3;  
        if (b == true)  
            return "be";  
        else  
            return "gets";  
    }  
    (this bool b) {  
        b.Green().SendIt();  
    }
```

```
    if (b == true)  
        return "be";  
    b.Green().SendIt();  
    public static string Green  
    {  
        i.ToPrice()  
        .SendIt();  
        (this string s) {  
    }
```



TheUNIVERSE



CAPTAIN AMAZING REBORN DEATH WAS NOT THE END

by Lucky Burns
UNIVERSE STAFF WRITER

OBJECTVILLE

Captain Amazing deserializes himself, making a stunning comeback.

In a stunning turn of events, Captain Amazing has returned to Objectville. Last month, Captain Amazing's coffin was found empty, and only a strange note left where his body should have been. Analysis of the note revealed Captain Amazing's object DNA—all his last fields and values, captured faithfully in binary format.

Today, that data has sprung to life. The Captain is back, deserialized from his own brilliant note. When asked how he conceived of such a plan, the Captain merely shrugged and mumbled, "Chapter 10." Sources close to the Captain refused to comment on the meaning of his cryptic reply, but did admit that prior to his failed assault on the Swindler, the Captain had spent a lot of time reading books, studying Dispose methods and persistence. We expect Captain Amazing...



Captain Amazing is back!

...see AMAZING on A-5



Extension Magnets Solution

Your job was to arrange the magnets to produce this output:

a buck begets more bucks

The Upside namespace has the extensions. The Sideways namespace has the entry point.

```
namespace Upside {
```

```
    public static class Margin {
```

```
        public static void SendIt (this string s) {
            Console.WriteLine(s);
        }
```

```
        public static string ToPrice (this int n) {
            if (n == 1)
                return "a buck ";
            else
                return " more bucks";
        }
```

```
        public static string Green (this bool b) {

```

```
            if (b == true)
                return "be";
            else
                return "gets";
        }
```

The Green method extends a bool—it returns the string “be” if the bool is true, and “gets” if it’s false.

The Margin class extends a string by adding a method called SendIt that just writes the string to the console, and it extends int by adding a method called ToPrice that returns “a buck” if the int’s equal to 1, or “more bucks” if it’s not.

```
namespace Sideways
{
```

```
    using Upside;
```

```
    class Program {
```

```
        static void Main(string[] args) {
```

```
            int i = 1;
```

```
            string s = i.ToPrice();
```

```
            s.SendIt();
```

```
            bool b = true;
```

```
            b.Green().SendIt();
```

```
            b = false;
```

```
            b.Green().SendIt();
```

```
            i = 3;
```

```
            i.ToPrice() .SendIt();
```

The Main method uses the extensions that you added in the Margin class.

12 exception handling



Putting out fires gets old

I KNOW THAT CREEPY CLOWN IS AROUND HERE SOMEWHERE. GOOD THING I WROTE CODE TO HANDLE MY TOTALLYFREAKEDOUTEXCEPTION.



Programmers aren't meant to be firefighters.

You've worked your tail off, wading 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.

your hex dumper threw an exception

Your hex dumper reads a filename from the command line

At the end of Chapter 10 you built a hex dumper that uses command-line arguments to dump any file. You used the project properties in the IDE to set the arguments for the debugger, and you saw how to call it from a Windows command prompt or macOS Terminal window.

```
C:\WINDOWS\system32\cmd.exe
C:\Users\Public\source\repos\HexDump\HexDump\bin\Debug\netcoreapp3.1>HexDump binarydata.dat
0000: 86 29 e8 02 06 48 65 6c -- 6c 6f 21 2f 81 00 74 f6      .)...Hello!..t.
0010: d8 f5 43 45          --                                ..CE
MacBook-Pro publish % ./HexDump binarydata.dat
0000: 86 29 e8 02 06 48 65 6c -- 6c 6f 21 2f 81 00 74 f6      .)...Hello!..t.
0010: d8 f5 43 45          --                                ..CE
MacBook-Pro publish %
```

But what happens if you give HexDump an invalid filename?

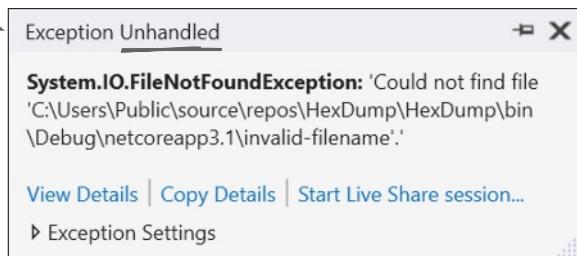
When you modified your HexDump app to use command-line arguments, we asked you to be careful to specify a valid filename. What happens when you give it an invalid filename? Try running your app again from the command line, but this time give it the argument `invalid-filename`. Now it *throws an exception*.

```
C:\WINDOWS\system32\cmd.exe    The exception has a class name and a message...
C:\Users\Public\source\repos\HexDump\HexDump\bin\Debug\netcoreapp3.1>HexDump invalid-filename
Unhandled exception System.IO.FileNotFoundException: Could not find file 'C:\Users\Public\source\repos\HexDump\HexDump\bin\Debug\netcoreapp3.1\invalid-filename'.
File name: 'C:\Users\Public\source\repos\HexDump\HexDump\bin\Debug\netcoreapp3.1\invalid-filename'
   at System.IO.FileStream.ValidateFileHandle(SafeFileHandle fileHandle)
   at System.IO.FileStream.CreateFileOpenHandle(FileMode mode, FileShare share, FileOptions options)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare share, Int32 bufferSize, FileOptions options)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare share)
   at System.IO.File.OpenRead(String path)
   at HexDump.Program.Main(String[] args) in C:\Users\Public\source\repos\HexDump\HexDump\Program.cs:line 12
...and a stack trace.
```

Use the project settings to set the program's argument to an invalid filename and run the app in the IDE's debugger. Now you'll see it throw an exception with the same class name (`System.IO.FileNotFoundException`) and a similar "Could not find file" message.

An unhandled exception means our app encountered a problem we didn't account for.

The IDE halts the debugger on the line that threw the exception, and shows you information in its Exception Unhandled window. You can even see the call stack in its Call Stack window.





```

class HoneyBee
{
    public double Capacity { get; set; }
    public string Name { get; set; }

    public HoneyBee(double capacity, string name)
    {
        Capacity = capacity;
        Name = name;
    }

    public static void Main(string[] args)
    {
        object myBee = new HoneyBee(36.5, "Zippo");
        float howMuchHoney = (float)myBee;

        HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
        double beeName = double.Parse(anotherBee.Name); ←

        double totalHoney = 36.5 + 12.5;
        string beesWeCanFeed = "";
        for (int i = 1; i < (int)totalHoney; i++)
        {
            beesWeCanFeed += i.ToString();
        }
        int numberOfBees = int.Parse(beesWeCanFeed);

        int drones = 4;
        int queens = 0;
        int dronesPerQueen = drones / queens;

        anotherBee = null;
        if (dronesPerQueen < 10)
        {
            anotherBee.Capacity = 12.6;
        }
    }
}

```

This code is broken. The code throws five different exceptions, and the error messages you'll see in the IDE or the console are on the right. It's your job to **match the line of code** that has a problem with the **exception that line generates**. Read the exception messages for hints. Keep in mind that

this code does not work. If you add this class to an app and run its Main method, it will halt after the first exception is thrown. Just read through the code and match each exception to the line that would throw it if it ran.

The double.Parse method converts a string to a double, so if you pass it a string (such as "32.7") it will return the equivalent double value (32.7). What do you think happens if you pass it a string that can't be converted to a double?

System.OverflowException: 'Value was either too large or too small for an Int32.'

1

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

2

System.InvalidCastException: 'Unable to cast object of type 'ExceptionTests.HoneyBee' to type 'System.Single'.'

3

System.DivideByZeroException: 'Attempted to divide by zero.'

4

System.FormatException: 'Input string was not in a correct format.'

5

Sharpen your pencil Solution

Your job was to match each line of code that has a problem with the exception that line generates.

```
object myBee = new HoneyBee(36.5, "Zippo");  
float howMuchHoney = (float)myBee;
```

The code to cast `myBee` to a float compiles, but there's no way to convert a `HoneyBee` object to a float value. When your code runs, the CLR has no idea how to actually do that cast, so it throws an `InvalidOperationException`.

System.InvalidCastException: 'Unable to cast object of type 'ExceptionTests.HoneyBee' to type 'System.Single'.'

3

IDE Tip: Set Next Statement

You can reproduce these exceptions in the IDE by pasting your code in and running it. Place a breakpoint on the first line of code, then right-click on the line of code you want to run and choose **Set Next Statement**—when you continue running, your app will jump straight to that statement.

```
HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");  
double beeName = double.Parse(anotherBee.Name);
```

System.FormatException: 'Input string was not in a correct format.'

The `Parse` method wants you to give it a string in a certain format. "Buzzy" isn't a string—it knows how to convert to a number. That's why it throws a `FormatException`.

```
double totalHoney = 36.5 + 12.5;  
string beesWeCanFeed = "";  
for (int i = 1; i < (int)totalHoney; i++)  
{  
    beesWeCanFeed += i.ToString();  
}
```

```
int numberOfBees = int.Parse(beesWeCanFeed);
```

The for loop will create a string called `beesWeCanFeed` that contains a number with over 60 digits in it. There's no way an int can hold a number that big, and trying to cram it into an int will throw an `OverflowException`.

System.OverflowException: 'Value was either too large or too small for an Int32.'

1

You'd never actually get all these exceptions in a row—the program would throw the first exception and then halt. You'd only get to the second exception if you fixed the first.



```
int drones = 4;
int queens = 0;
int dronesPerQueen = drones / queens;
```

It's really easy to throw a DivideByZeroException. Just divide any number by zero.

System.DivideByZeroException: 'Attempted to divide by zero.'

4

Dividing any integer by zero always throws this kind of exception. Even if you don't know the value of queens, you can prevent it just by checking the value to make sure it's not zero before you divide it into drones.



I CAN SEE JUST FROM LOOKING AT THAT CODE THAT IT'S TRYING TO DIVIDE BY ZERO. I BET THAT EXCEPTION DIDN'T HAVE TO HAPPEN.

That DivideByZero error didn't have to happen.

You can see just by looking at the code that there's something wrong. If you think about it, the same goes for the other exceptions—this whole “Sharpen your pencil” exercise was about spotting those exceptions without running the code. Every one of those exceptions was **preventable**. The more you know about exceptions, the better you'll be at keeping your apps from crashing.

```
anotherBee = null;
if (dronesPerQueen < 10)
{
    anotherBee.Capacity = 12.6;
```

Setting the anotherBee reference variable equal to null tells C# that it doesn't point to anything. So instead of pointing to an object, it points to nothing. Throwing a NullReferenceException is C#'s way of telling you that there's no object whose DoMyJob method can be called.

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

2

When your program throws an exception, the CLR generates an Exception object

You've been looking at the CLR's way of telling you something went wrong in your program: an **exception**. When an exception occurs in your code, an object is created to represent the problem. It's called—no surprise here—Exception.

For example, suppose you have an array with four items, and you try to access the 16th item (index 15, since we're zero-based here):

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

This code is
obviously going to
cause problems.

As soon as your program runs into an exception, it generates an object with all the data it has about the problem.

The Exception object has a message that tells you what's wrong, and a stack trace, or a list of all of the calls that were made leading up to the statement that caused the exception.

When the IDE halts because the code threw an exception, you can see the details of the exception by **expanding \$exception in the Locals window**. The Locals window shows you all of the variables currently **in scope** (which means the current statement has access to them).

Locals	
Name	Value
\$exception	{"Index was outside the bounds of the array."}
Data	{System.Collections.ListDictionaryInternal}
HRESULT	-2146233080
HelpLink	null
InnerException	null
Message	"Index was outside the bounds of the array."
Source	"ConsoleApp1"
StackTrace	" at ConsoleApp1.Program.Main(String[] arg...")
TargetSite	{Void Main(System.String[])}
Static members	
Non-Public members	

The CLR goes to the trouble of creating an object because it wants to give you all the information it has about what caused the exception. You may have code to fix, or you may just need to make some changes to how you handle a particular situation in your program.

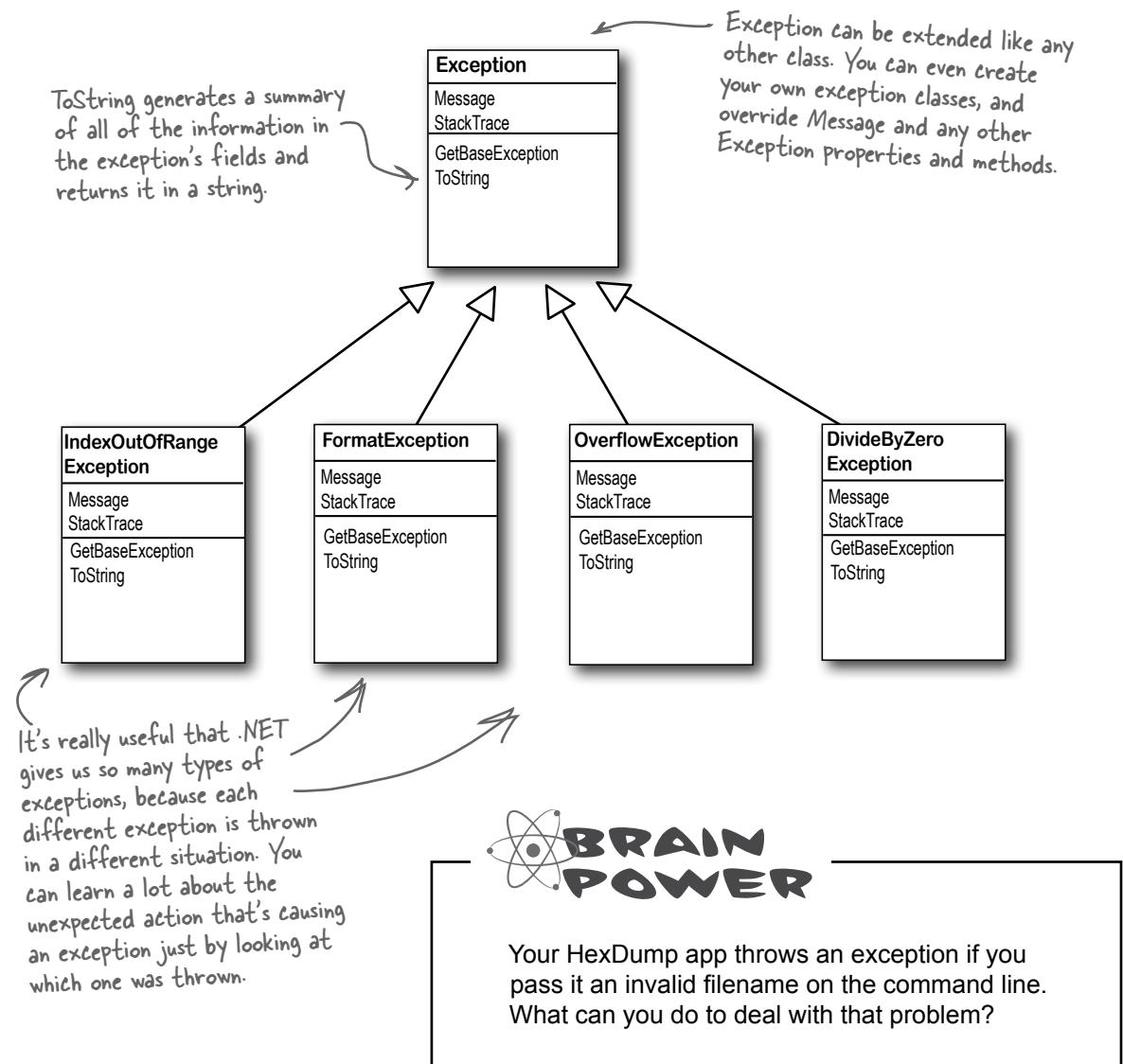
This particular exception is an **IndexOutOfRangeException**, which gives you an idea of what the problem is: you're trying to access an index in the array that's out of range. You've also got information about exactly where in the code the problem occurred, making it easier to track down and solve (even if you've got thousands of lines of code).

ex-cep-tion, noun.
a person or thing that is excluded from a general statement or does not follow a rule. *While Jamie usually hates peanut butter, they made an exception for Parker's peanut butter fudge.*

All Exception objects inherit from System.Exception

.NET has lots of different exceptions it may need to report. Since many of these have a lot of similar features, inheritance comes into play. .NET defines a base class, called `Exception`, that all specific exception types inherit from.

The `Exception` class has a couple of useful members. The `Message` property stores an easy-to-read message about what went wrong. `StackTrace` tells you what code was being executed when the exception occurred, and what led up to the exception. (There are others, too, but we'll use those first.)





Sleuth it out

The first step in troubleshooting any exception is taking a really close look at all of the information that it gives you. When you're running a console app, that information is written to the console. Let's take a closer look at the diagnostic information our app printed (we moved our app to the C:\HexDump folder to make the paths in this stack trace shorter):

```
C:\HexDump\bin\Debug\netcoreapp3.1> HexDump invalid-filename
Unhandled exception. System.IO.FileNotFoundException: Could not find file 'C:\HexDump\bin\Debug\netcoreapp3.1\invalid-filename'.
File name: 'C:\HexDump\bin\Debug\netcoreapp3.1\invalid-filename'
   at System.IO.FileStream.ValidateFileHandle(SafeFileHandle fileHandle)
   at System.IO.FileStream.CreateFileOpenHandle(FileMode mode, FileShare share, FileOptions options)
   at System.IO.FileStream..ctor(String path, FileMode mode, ... , FileOptions options)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare share)
   at System.IO.File.OpenRead(String path)
   at HexDump.Program.Main(String[] args) in C:\HexDump\Program.cs:line 12
```

Here's what we noticed:

- The exception class: `System.IO.FileNotFoundException`
- The exception message: Could not find file 'C:\HexDump\bin\Debug\netcoreapp3.1\invalid-filename'.
- Additional diagnostic information: File name: 'C:\HexDump\bin\Debug\netcoreapp3.1\invalid-filename'
- The first five lines in the stack trace come from classes in the `System.IO` namespace.
- The last line of the stack trace is in our namespace, `HexDump`—and it includes a line number. Here's what's on that line: `using (Stream input = File.OpenRead(args[0]))`

Reproduce the exception in the debugger

At the end of Chapter 10 we showed you how to set the application arguments when you run an app in the debugger. Set the arguments to `invalid-filename`, then place a breakpoint on the line in the app that's throwing the exception. Run the app, then when you get to the breakpoint step over that statement. You should see the exception in the IDE.

Add code to prevent the exception

If you're running from the command line on your Mac, don't forget to re-publish it by running "dotnet publish -r osx-x64" from the solution folder.

The app is throwing an exception because it's trying to read a file that doesn't exist. So let's **prevent** this exception from happening by checking if the file exists first. If it doesn't, instead of using `File.OpenRead` to open a stream with the contents of the file, we'll use `Console.OpenStandardInput`, which returns a stream of your app's **standard input** (or `stdin`). Start by **adding this `GetInputStream` method** to your app:

```
static Stream GetInputStream(string[] args) {
    if ((args.Length != 1) || !File.Exists(args[0]))
        return Console.OpenStandardInput();
    else
        return File.OpenRead(args[0]);
}
```

`Console.OpenStandardInput` returns a `Stream` object connected to the app's standard input. If you pipe input into the app or run it in the IDE and type `console` or `terminal`, anything you type or pipe in ends up in the stream.

Then modify the line of the app that throws the exception to call the new method:

```
using (Stream input = GetInputStream(args))
```

Now run your app in the IDE. It doesn't throw an exception. Instead, it's reading from standard input. Test it out:

- Type some data and press Enter—it will show a hex dump of everything you typed, ending with a return (0d 0a on Windows, 0a on Mac). The `stdin` stream only adds data after a return, so the app will do a new dump for each line.
- Run your app from the command line: `HexDump << input.txt` (or `./HexDump << input.txt` on a Mac). The app will pipe data from `input.txt` into the `stdin` stream and dump all of the bytes in the file.

there are no
Dumb Questions

Q: What is an exception, really?

A: It's an object that the CLR creates when there's a problem. You can specifically generate exceptions in your code, too—in fact, you've done that already using the `throw` keyword.

Q: An exception is an *object*?

A: Yes, an exception is an object. The CLR generates it to give you as much information as it can about exactly what was going on when it executed the statement that threw the exception. Its properties—which you saw when you expanded \$exception in the Locals window—tell you information about the exception. For example, its `Message` property will have a useful string like *Attempted to divide by zero or Value was either too large or too small for an Int.32.*

Q: Why are there so many types of Exception objects?

A: Because there are so many ways that your code can act in unexpected ways. There are a lot of situations that will cause your code to simply crash. It would be really hard to troubleshoot the problems if you didn't know why the crash happened. By throwing different kinds of exceptions under different circumstances, the CLR is giving you a lot of really valuable information to help you track down and correct the problem.



IT SOUNDS LIKE EXCEPTIONS AREN'T ALWAYS BAD.
SOMETIMES THEY IDENTIFY BUGS, BUT A LOT OF THE TIME
THEY'RE JUST TELLING ME THAT SOMETHING HAPPENED
THAT WAS DIFFERENT FROM WHAT I EXPECTED.

That's right. Exceptions are a really useful tool that you can use to find places where your code acts in ways you don't expect.

A lot of programmers get frustrated the first time they see an exception. But exceptions are really useful, and you can use them to your advantage. When you see an exception, it's giving you a lot of clues to help you figure out why your code is reacting in a way that you didn't anticipate. That's good for you: it lets you know about a new scenario that your program has to handle, and it gives you an opportunity to **do something about it**.

Q: Does that mean that when my code throws an exception, it's not necessarily because I did something wrong?

A: Exactly. Sometimes your data's different than you expected it to be, like in the example where we had a statement working with an array that was a lot shorter than anticipated. Also, don't forget that real, live, actual human beings are using your program, and they'll often act in unpredictable ways. Exceptions are C#'s way to help you handle those unexpected situations so that your code still runs smoothly and doesn't simply crash or give a cryptic, useless error message.

Q: So exceptions are there to help me, not just cause me grief and force me to track down frustrating bugs at 3:00 AM?

A: Yes! Exceptions are all about helping you expect the unexpected. A lot of people get frustrated when they see code throw an exception. Instead, think about them as C#'s way of helping you track down problems and debug your programs.

**Exceptions
are all about
helping you
find and fix
situations
where
your code
behaves
in ways
you didn't
expect.**

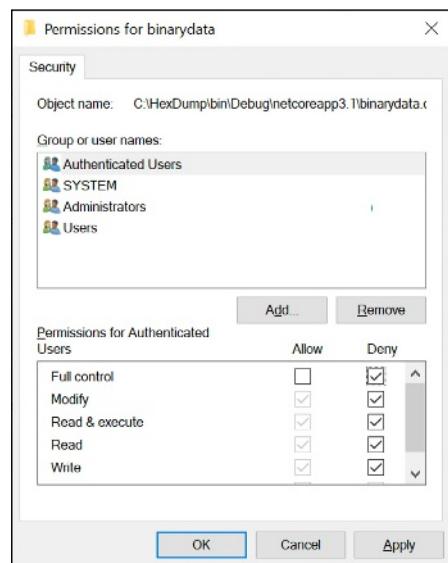
There are some files you just can't dump

In Chapter 9 we talked about making your code **robust** so that it can deal with bad data, malformed input, user errors, and other unexpected situations. Dumping stdin if no file is passed on the command line or if the file doesn't exist is a great start to making the hex dumper robust.

But are there still some cases that we need to handle? For example, what if the file exists but it's not readable? Let's see what happens if we remove read access from a file, then try to read it:

- ★ **On Windows:** Right-click the file in the Windows Explorer, go to the Security tab, and click Edit to change the permissions. Check all of the Deny boxes. 
- ★ **On a Mac:** In a Terminal window, change to the folder with the file you want to dump, and run this command, replacing `binarydata.dat` with the name of your file): `chmod 000 binarydata.dat`.

Now that you've removed read permissions from your file, try running your app again, either in the IDE or from the command line.



You'll see an exception—the stack trace shows that the **using statement called the `GetInputStream` method**, which eventually caused the `FileStream` to throw a `System.UnauthorizedAccessException`:

```
C:\HexDump\bin\Debug\netcoreapp3.1>hexdump binarydata.dat
Unhandled exception. System.UnauthorizedAccessException: Access to the path 'C:\HexDump\bin\Debug\netcoreapp3.1\binarydata.dat' is denied.
   at System.IO.FileStream.ValidateFileHandle(SafeFileHandle fileHandle)
   at System.IO.FileStream.CreateFileOpenHandle(FileMode mode, ..., FileMode options)
   at System.IO.FileStream..ctor(String path, ..., Int32 bufferSize, FileMode options)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare share)
   at System.IO.File.OpenRead(String path)
   at HexDump.Program.GetInputStream(String[] args) in C:\HexDump\Program.cs:line 14
   at HexDump.Program.Main(String[] args) in C:\HexDump\Program.cs:line 20
```



WAIT A SECOND. OF COURSE THE PROGRAM'S GONNA CRASH.
I GAVE IT AN UNREADABLE FILE. USERS SCREW UP ALL THE
TIME. YOU CAN'T EXPECT ME TO DO ANYTHING ABOUT THAT...
RIGHT?

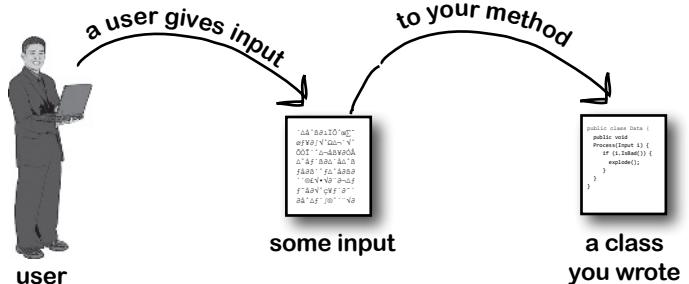
Actually, there *is* something you can do about it.

Yes, users do, indeed, screw up all the time. They'll give you bad data, weird input, click on things you didn't even know existed. That's a fact of life, but that doesn't mean you can't do anything about it. C# gives you really useful **exception handling tools** to help you make your programs more robust. Because while you *can't* control what your users do with your app, you *can* make sure that your app doesn't crash when they do it.

What happens when a method you want to call is risky?

Users are unpredictable. They feed all sorts of weird data into your program, and click on things in ways you never expected. That's just fine, because you can deal with exceptions that your code throws by adding **exception handling**, which lets you write special code that gets executed any time an exception is thrown.

- ① Let's say a method called in your program takes input from a user.



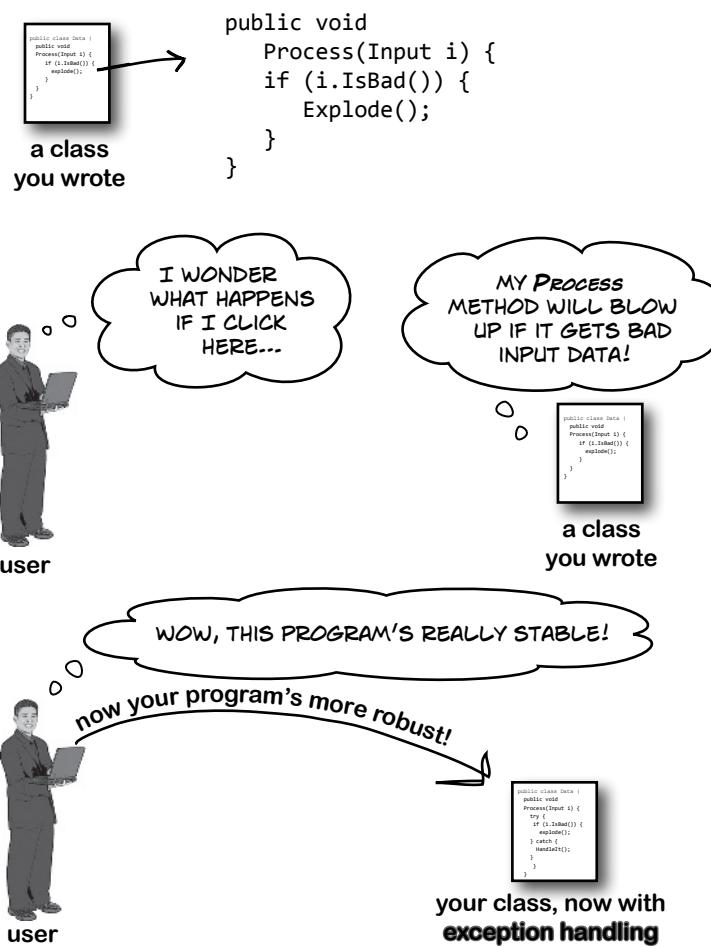
- ② That method could do something risky that might not work at runtime.

"Runtime" just means "while your program is running." Some people refer to exceptions as "runtime errors."

- ③ You need to know that the method you're calling is risky.

If you can come up with a way to do a less risky thing that avoids throwing the exception, that's the best possible outcome! But some risks just can't be avoided, and that's when you want to do this.

- ④ You can then write code that can handle the exception if it does happen. You need to be prepared, just in case.



Handle exceptions with try and catch

When you add exception handling to your code, you're using the **try** and **catch** keywords to create a block of code that gets run if an exception is thrown.

Your *try/catch* code basically tells the C# compiler: “**Try** this code, and if an exception occurs, **catch** it with this *other* bit of code.” The part of the code you’re trying is the **try block**, and the part where you deal with exceptions is called the **catch block**. In the **catch block**, you can do things like print a friendly error message instead of letting your program come to a halt.

Let’s have another look at the last three lines of the stack trace in our HexDump scenario to help us figure out where to put our exception handling code:

```
at System.IO.File.OpenRead(String path)
at HexDump.Program.GetInputStream(String[] args) in Program.cs:line 14
at HexDump.Program.Main(String[] args) in Program.cs:line 20
```

The UnauthorizedAccessException is caused by the line in GetInputStream that calls File.OpenRead. Since we can’t prevent that exception, let’s modify GetInputStream to use a *try/catch* block:

```
static Stream GetInputStream(string[] args)
{
    if ((args.Length != 1) || !File.Exists(args[0]))
        return Console.OpenStandardInput();
    else
    {
        try
        {
            return File.OpenRead(args[0]);
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.Error.WriteLine("Unable to read {0}, dumping from stdin: {1}",
                args[0], ex.Message);
            return Console.OpenStandardInput();
        }
    }
}
```

This is the try block. You start exception handling with “try”. In this case, we’ll put the existing code in it.

Put the code that might throw an exception inside the try block. If no exception happens, it'll get run exactly as usual, and the statements in the catch block will be ignored. If a statement in the try block throws an exception, the rest of the try block won't get executed.

The catch keyword signals that the block immediately following it contains an exception handler.

When an exception is thrown inside the try block, the program immediately jumps to the catch statement and starts executing the catch block.

We kept things simple in our exception handler. First we used Console.Error to write a line to the error output (stderr) letting the user know that an error occurred, then we fell back to reading data from standard input so the program still did something. Notice how **the catch block has a return statement**. The method returns a Stream, so if it handles an exception it still needs to return a Stream; otherwise you’ll get the “not all code paths return a value” compiler error.



If throwing an exception makes your code automatically jump to the catch block, what happens to the objects and data you were working with before the exception happened?

Use the debugger to follow the try/catch flow

An important part of exception handling is that when a statement in your `try` block throws an exception, the rest of the code in the block gets **short-circuited**. The program's execution immediately jumps to the first line in the `catch` block. Let's use the IDE's debugger to explore how this works.

Debug this

- ➊ Replace the `GetInputStream` method in your `HexDump` app with the one that we just showed you to handle an `UnauthorizedAccessException`.
- ➋ Modify your project options to set the argument so that it contains the path to an unreadable file.
- ➌ Place a breakpoint on the first statement in `GetInputStream`, then start debugging your project.
- ➍ When it hits the breakpoint, step over the next few statements until you get to `File.OpenRead`. Step over it—the app jumps to the first line in the `catch` block.

Here's the breakpoint we placed on the first line of `GetInputStream`.

Step through the method. When it hits `File.OpenRead` it will throw an exception, which will cause execution to jump to the `catch` block.

```

HexDump - Program.cs
Program.cs + X
HexDump
HexDump.Program
GetInputStream(string[] args)

1 reference
static Stream GetInputStream(string[] args)
{
    if ((args.Length != 1) || !File.Exists(args[0]))
        return Console.OpenStandardInput();
    else
    {
        try
        {
            return File.OpenRead(args[0]);
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.Error.WriteLine("Unable to read {0}, dumping from stdin: {1}",
                args[0], ex.Message);
            return Console.OpenStandardInput();
        }
    }
}

```

- ➎ Keep stepping through the rest of the `catch` block. It will write the line to the console, then return `Console.OpenStandardInput` and resume the `Main` method.

If you have code that ALWAYS needs to run, use a **finally** block

When your program throws an exception, a couple of things can happen. If the exception **isn't** handled, your program will stop processing and crash. If the exception **is** handled, your code jumps to the **catch** block. What about the rest of the code in your **try** block? What if you were closing a stream, or cleaning up important resources? That code needs to run, even if an exception occurs, or you're going to make a mess of your program's state. That's where you'll use a **finally** block. It's a block of code that comes after the **try** and **catch**. The **finally** block **always runs**, whether or not an exception was thrown. Let's use the debugger to explore how the **finally** block works.

1 Create a new Console App project.

Add using `System.IO`; to the top of the file, then add this Main method:

```
static void Main(string[] args)
{
    var firstLine = "No first line was read";
    try
    {
        var lines = File.ReadAllLines(args[0]);
        firstLine = (lines.Length > 0) ? lines[0] : "The file was empty";
    }
    catch (Exception ex)
    {
        Console.Error.WriteLine("Could not read lines from the file: {0}", ex);
    }
    finally
    {
        Console.WriteLine(firstLine);
    }
}
```

WriteLine calls the `Exception` object's `Tostring` method, which returns the exception name, message, and stack trace.

This finally block will run whether or not the try catches an exception.

You'll see the exception in the Locals window, just like you did earlier.

2 Add a breakpoint to the first line of the Main method.

Debug your app and step through it. The first line in the **try** block tries to access `args[0]`, but since you didn't specify any command-line arguments the `args` array is empty and it throws an exception—specifically, a `System.IndexOutOfRangeException`, with the message “*Index was outside the bounds of the array.*” After it prints the message, it **executes the finally block**, and then the program exits.

3 Set a command-line argument with the path of a valid file.

Use the project properties to pass a command-line argument to the app. Give it the full path of a valid file. Make sure there are no spaces in the filename, otherwise the app will interpret it as two arguments. Debug your app again—after it finishes the **try** block, it **executes the finally block**.

4 Set a command-line argument with an invalid file path.

Go back to the project properties and change the command-line argument to pass the app the name of a file that does not exist. Run your app again. This time it catches a different exception: a `System.IO.FileNotFoundException`. Then it **executes the finally block**.

Catch-all exceptions handle System.Exception

You just made your console app throw two different kinds of exception—an `IndexOutOfRangeException` and a `FileNotFoundException`—and they were both handled. Take a closer look at the `catch` block:

```
catch (Exception ex)
```

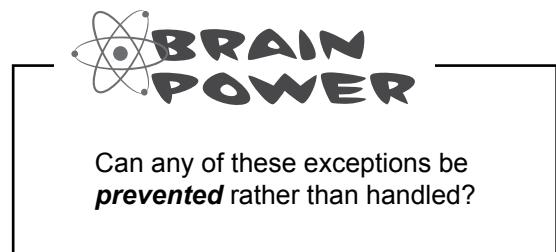
This is a **catch-all exception**: the type after the `catch` block indicates what type of exception to handle, and since all exceptions extend the `System.Exception` class, specifying `Exception` as the type tells the `try/catch` block to catch any exception.

Avoid catch-all exception with multiple catch blocks

It's always better to try to anticipate the specific exceptions that your code will throw and handle them. For example, we know that this code can throw an `IndexOutOfRangeException` if no filename is specified, or a `FileNotFoundException` if an invalid file is found. We also saw earlier in the chapter that trying to read an unreadable file causes the CLR to throw an `UnauthorizedAccessException`. You can handle these different kinds of exceptions by adding **multiple catch blocks** to your code:

```
static void Main(string[] args)
{
    var firstLine = "No first line was read";
    try
    {
        var lines = File.ReadAllLines(args[0]);
        firstLine = (lines.Length > 0) ? lines[0] : "The file was empty";
    }
    catch (IndexOutOfRangeException)
    {
        Console.Error.WriteLine("Please specify a filename.");
    }
    catch (FileNotFoundException)
    {
        Console.Error.WriteLine("Unable to find file: {0}", args[0]);
    }
    catch (UnauthorizedAccessException ex) ← Only this catch block had to specify a
    {                                         variable name for the Exception object.
        Console.Error.WriteLine("File {0} could not be accessed: {1}",
                               args[0], ex.Message);
    }
    finally
    {
        Console.WriteLine(firstLine);
    }
}
```

This exception handler has three catch blocks to handle three different Exception types.



Now your app will write different error messages depending on which exception is handled. Notice that the first two `catch` blocks **did not specify a variable name** (like `ex`). You only need to specify a variable name if you're going to use the `Exception` object.

there are no
Dumb Questions

Q: Back up a second. So every time my program runs into an exception, it's going to stop whatever it's doing unless I specifically write code to catch it? How is that a good thing?

A: One of the best things about exceptions is that they make it obvious when you run into problems. Imagine how easy it could be in a complex application for you to lose track of all of the objects your program was working with. Exceptions call attention to your problems and help you root out their causes so that you always know that your program is doing what it's supposed to do.

Any time an exception occurs in your program, something you expected to happen didn't. Maybe an object reference wasn't pointing where you thought it was, or a user supplied a value you hadn't considered, or a file you thought you'd be working with suddenly isn't available. If something like that happened and you didn't know it, it's likely that the output of your program would be wrong, and the behavior from that point on would be pretty different from what you expected when you wrote the program.

Now imagine that you had no idea the error had occurred and your users started calling you up and telling you that your program was unstable. That's why it's a good thing that exceptions disrupt everything your program is doing. They force you to deal with the problem while it's still easy to find and fix.

Q: Remind me again—what do I use the Exception object for?

A: The Exception object gives you clues about what went wrong. You can use its type to determine what kind of problem happened, and write an exception handler to deal with it in a way that keeps your app running.

Q: What's the difference between a *handled exception* and an *unhandled exception*?

A: Whenever your program throws an exception, the runtime environment will search through your code looking for a catch block that handles it. If you've written one, the catch block will execute and do whatever you specified for that particular exception. Since you wrote a catch block to deal with that error up front, that exception is considered handled.

If the runtime can't find a catch block that matches the exception, it stops everything your program is doing and raises an error. That's an *unhandled exception*.

Q: What happens when you have a catch that doesn't specify a particular exception?

A: That's called a **catch-all exception**. A catch block like that will catch any kind of exception the try block can throw. So if you don't need to declare a variable to use the Exception object, an easy way to write a catch-all exception is like this:

```
catch
{
    // handle the exception
}
```

Q: Isn't it easier to use a catch-all exception? Isn't it safer to write code that always catches every exception?

A: You should **always do your best to avoid catching Exception**, and instead catch specific exceptions. You know that old saying about how an ounce of prevention is better than a pound of cure? That's especially true in exception handling. Depending on catch-all exceptions is usually just a way to make up for bad programming. For example, you're often better off using File.Exists to check for a file before you try to open it than catching a FileNotFoundException. While some exceptions are unavoidable, you'll find that a surprising number of them never have to be thrown in the first place.

Q: If a catch block with no specified exception will catch anything, why would I ever want to specify an exception type?

A: Certain exceptions might require different actions to keep your program moving. Maybe an exception caused by dividing by zero might have a catch block where it goes back and sets properties to save some of the data you've been working with, while a null reference exception in the same block of code might require it to create new instances of an object.

An unhandled exception can cause your program to run unpredictably. That's why the program stops whenever it runs into one.

Pool Puzzle



Your **job** is to take code snippets from the pool and place them into the blank lines in the program. You can use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make the program produce the output below.

Output: → G'day Mate!

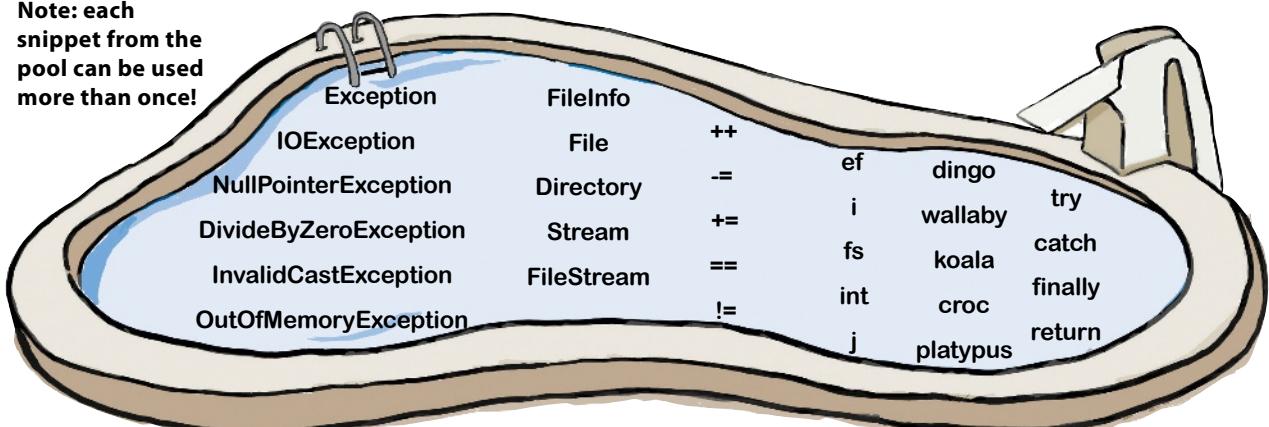
```
using System.IO;

class Program {
    public static void Main() {
        Kangaroo joey = new Kangaroo();
        int koala = joey.Wombat(
            joey.Wombat(joey.Wombat(1)));
        try {
            Console.WriteLine((15 / koala)
                + " eggs per pound");
        }
        catch (_____) {
            Console.WriteLine("G'Day Mate!");
        }
    }
}
```

```
class Kangaroo {
    _____ fs;
    int croc;
    int dingo = 0;

    public int Wombat(int wallaby) {
        _____;
        try {
            if (_____ > 0) {
                fs = File.OpenWrite("wobbiegong");
                croc = 0;
            } else if (_____ < 0) {
                croc = 3;
            } else {
                _____ = _____.OpenRead("wobbiegong");
                croc = 1;
            }
        }
        catch (IOException) {
            croc = -3;
        }
        catch {
            croc = 4;
        }
        finally {
            if (_____ > 2) {
                croc _____ dingo;
            }
        }
    }
}
```

Note: each snippet from the pool can be used more than once!



Poo! Puzzle Solution



```
using System.IO;

class Program {
    public static void Main() {
        Kangaroo joey = new Kangaroo();
        int koala = joey.Wombat(
            joey.Wombat(joey.Wombat(1)));
        try {
            Console.WriteLine((15 / koala)
                + " eggs per pound");
        }
        catch ( DivideByZeroException ) {
            Console.WriteLine("G'Day Mate!");
        }
    }
}
```

The clue that this is a FileStream is that it has an OpenRead method and throws an IOException.

```
class Kangaroo {
    FileStream fs;
    int croc;
    int dingo = 0;
```

This code opens a file called "wobbiegong" and keeps it open the first time it's called. Later on, it opens the file again. But it never closed the file, which causes it to throw an IOException.

```
public int Wombat(int wallaby) {
    dingo ++;
    try {
        if (wallaby > 0) {
            fs = File.OpenWrite("wobbiegong");
            croc = 0;
        } else if (wallaby < 0) {
            croc = 3;
        } else {
            fs = File.OpenRead("wobbiegong");
            croc = 1;
        }
    }
    catch (IOException) {
        croc = -3;
    }
    catch {
        croc = 4;
    }
    finally {
        if (dingo > 2) {
            croc -= dingo;
        }
    }
    return croc;
}
```

Remember, you should avoid catch-all exceptions in your code. You should also avoid other things we do to make puzzles more interesting, like using obfuscated variable names.

joey.Wombat is called three times, and the third time it returns zero. That causes WriteLine to throw a DivideByZeroException.

This catch block only catches exceptions where the code divides by zero.

You already know that you always have to close files when you're done with them. If you don't, the file will be locked open, and if you try to open it again it'll throw an IOException.



YOU KEEP TALKING ABOUT **RISKY CODE**,
BUT ISN'T IT RISKY TO LEAVE AN EXCEPTION
UNHANDLED? WHY WOULD I EVER WRITE AN
EXCEPTION HANDLER THAT DOESN'T HANDLE EVERY
TYPE OF EXCEPTION?

Unhandled exceptions **bubble up**.

Believe it or not, it can be really useful to leave exceptions unhandled. Real-life programs have complex logic, and it's often difficult to recover correctly when something goes wrong, especially when a problem occurs very far down in the program. By only handling specific exceptions and avoiding catch-all exception handlers, you let unexpected exceptions **bubble up**: instead of being handled in the current method, they're caught by the next statement up the call stack. Anticipating and handling the exceptions that you expect and letting unhandled exceptions bubble up is a great way to build more robust apps.

Sometimes it's useful to **rethrow** an exception, which means that you handle an exception in a method but *still bubble it up* to the statement that called it. All you need to do to rethrow an exception is call **throw**; inside a **catch** block, and the exception that it caught will immediately bubble up:

```
try {
    // some code that might throw an exception
} catch (DivideByZeroException d) {
    Console.Error.WriteLine($"Got an error: {d.Message}");
    throw;
}
```

The **throw** command causes the **DivideByZeroException** to bubble up to whatever code called this **try/catch** block.

Here's a career tip: a lot of C# programming job interviews include a question about how you deal with exceptions in a constructor.



Watch it!

Keep risky code out of the constructor!

You've noticed by now that a constructor doesn't have a return value, not even **void**. That's because a constructor doesn't actually return anything. Its only purpose is to initialize an object—which is a problem for exception handling inside the constructor.

When an exception is thrown inside the constructor, then the statement that tried to instantiate the class won't end up with an instance of the object.

Use the right exception for the situation

When you use the IDE to generate a method, it adds code that looks like this:

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

The **NotImplementedException** is used any time you have an operation or method that hasn't been implemented. It's a great way to add a placeholder—as soon as you see one you know there's code that you need to write. That's just one of the many exceptions that .NET provides.

Choosing the right exception can make your code easier to read, and make exception handling cleaner and more robust. For example, code in a method that validates its parameters can throw an `ArgumentException`, which has an overloaded constructor with a parameter to specify which argument caused the problem.

Consider the `Guy` class back in Chapter 3, had a `ReceiveCash` method that checked the `amount` parameter to make sure it was receiving a positive amount. This is a good opportunity to throw an `ArgumentException`:

```
public void ReceiveCash(int amount)
{
    if (amount <= 0)
        throw new ArgumentException($"Must receive a positive value", "amount");
    Cash += amount;
}
```

Pass the name of the invalid argument to
the `ArgumentException` constructor.
↓

Take a minute and look over the list of exceptions that are part of the .NET API—you can throw any of them in your code: <https://docs.microsoft.com/en-us/dotnet/api/system.systemexception>.

Catch custom exceptions that extend `System.Exception`

Sometimes you want your program to throw an exception because of a special condition that could happen when it runs. Let's go back to the `Guy` class from Chapter 3. Suppose you're using it in an app that absolutely depended on a `Guy` always having a positive amount of cash. You could add a custom exception that **extends** `System.Exception`:

```
class OutOfCashException : System.Exception {
    public OutOfCashException(string message) : base(message) { }
}
```

This custom `OutOfCashException`
extends the base `System.Exception`
constructor with a `message` parameter.
↓

Now you can throw that new exception, and catch it exactly like you'd catch any other exception:

```
class Guy
{
    public string Name;
    public int Cash;

    public int GiveCash(int amount)
    {
        if (Cash <= 0) throw new OutOfCashException($"{Name} ran out of cash");
        ...
    }
}
```

Now that your `Guy` throws a custom
exception, whatever method calls
`GiveCash` can handle that exception
in its own try/catch block.
↓

```

class Program {
    public static void Main(string[] args) {
        Console.Write("when it ");
        ExTestDrive.Zero("yes");
        Console.Write(" it ");
        ExTestDrive.Zero("no");
        Console.WriteLine(".");
    }
}

class MyException : Exception { }

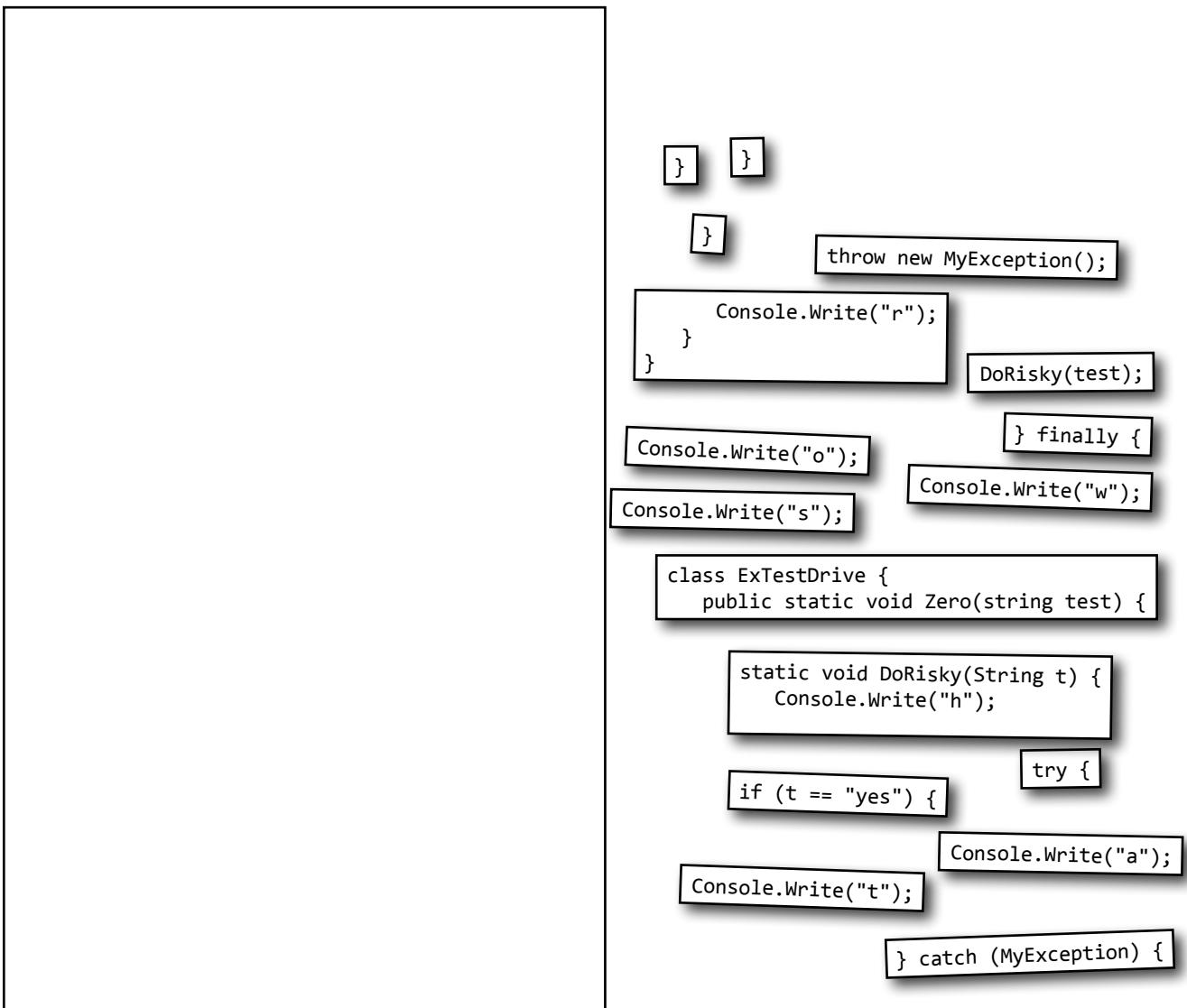
```



Exception Magnets

Arrange the magnets so the application writes the following output to the console:

when it thaws it throws.



```
class Program {  
    public static void Main(string[] args) {  
        Console.Write("when it ");  
        ExTestDrive.Zero("yes");  
        Console.Write(" it ");  
        ExTestDrive.Zero("no");  
        Console.WriteLine(".");
    }  
}
```

```
class MyException : Exception { }
```

This line defines a custom exception called `MyException`, which gets caught in a catch block in the code.

```
class ExTestDrive {  
    public static void Zero(string test) {  
        try {  
            Console.Write("t");  
            DoRisky(test);  
            Console.Write("o");  
        } catch (MyException) {  
            Console.Write("a");  
        } finally {  
            Console.Write("w");  
        }  
        Console.Write("s");  
    }  
  
    static void DoRisky(String t) {  
        Console.Write("h");  
        if (t == "yes") {  
            throw new MyException();  
        }  
        Console.Write("r");  
    }  
}
```

This line only gets executed if `DoRisky` doesn't throw the exception.



Solution Exception Magnets

Arrange the magnets so the application writes the following output to the console:

when it thaws it throws.

The `Zero` method either prints "thaws" or "throws", depending on whether it was passed "yes" or something else as its `test` parameter.

The `finally` block makes sure that the method always prints "w", and the "s" is printed outside the exception handler, so it always prints, too.

The `DoRisky` method only throws an exception if it's passed the string "yes".



IDisposable uses try/finally to ensure the Dispose method is called

Remember when we sleuthed out this code from Chapter 10?

```
using System.IO;
using System.Text;

class Program
{
    static void Main(string[] args)
    {
        using (var ms = new MemoryStream())
        {
            using (var sw = new StreamWriter(ms))
            {
                sw.WriteLine("The value is {0:0.00}", 123.45678);
            }
            Console.WriteLine(Encoding.UTF8.GetString(ms.ToArray()));
        }
    }
}
```

We moved this using statement inside the outer one to make sure the StreamWriter.Close method got called before MemoryStream.Close.

Behind the Scenes

We were exploring how `using` statements work, and in this case we needed to nest one `using` statement inside the other to make sure the StreamWriter was disposed of before the MemoryStream. We did that because both the StreamWriter and MemoryStream classes implement the `IDisposable` interface and put a call to their `Close` method inside their `Dispose` method. Each `using` statement makes sure the `Dispose` method is called at the end of its block, and that ensures the streams are always closed.

Now that you've been working with exception handling, you can see how the `using` statement works. The `using` statement is an example of **syntactic sugar**: a way that C# "sweetens" the language by giving you a convenient shortcut that makes your code easier to read. When you do this, it's actually a kind of shortcut:

```
using (var sw = new StreamWriter(ms))
{
    sw.WriteLine("The value is {0:0.00}", 123.45678);
}
```

The C# compiler actually generates compiled code that looks (approximately) like this:

```
try {
    var sw = new StreamWriter(ms)
    sw.WriteLine("The value is {0:0.00}", 123.45678);
} finally {
    sw.Dispose();
}
```

**IDisposable
is a great
tool for
avoiding
exceptions.**

Putting the `Dispose` statement in a `finally` block makes sure that it's always run—even if an exception occurs.

- AVOID UNNECESSARY EXCEPTIONS...ALWAYS USE A
- USING BLOCK ANY TIME YOU USE A STREAM!

Or anything else
that implements
`IDisposable`.

Exception filters help you create precise handlers

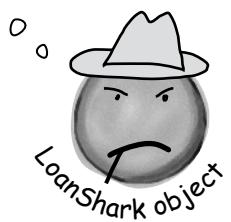
Let's say we're building a game set in classic 1930s mafia gangland, and we've got a LoanShark class that needs to collect cash from instances of Guy using the Guy.GiveCash method, and that handles any OutOfCashException with a good old gangland-style lesson.

The thing is, every loan shark knows the golden rule: don't try to collect money from the big mob boss. That's where **exception filters** can come in really handy. An exception filter uses the `when` keyword to tell your exception handler to catch an exception only under specific conditions.

Here's an example of how an exception filter works:

```
try
{
    loanShark += guy.GiveCash(amount);
    emergencyReserves -= amount;
} catch (OutOfCashException) when (guy.Name == "Al Capone")
{
    Console.WriteLine("Don't mess with the mafia boss");
    loanShark += amount;
} catch (OutOfCashException ex)
{
    Console.Error.WriteLine($"Time to teach {guy.Name} a lesson: {ex.Message}");
}
```

This exception filter only catches OutOfCashException when guy.Name is set to "Al Capone"—otherwise it falls through to the next catch block.



IF TRY/CATCH IS SO GREAT, WHY DOESN'T THE IDE JUST PUT IT AROUND EVERYTHING? THEN WE WOULDN'T HAVE TO WRITE ALL THESE TRY/CATCH BLOCKS ON OUR OWN. RIGHT?



It's always better to build the most precise exception handlers that we can.

There's more to exception handling than just printing out a generic error message. Sometimes you want to handle different exceptions differently—like how the hex dumper handled a FileNotFoundException differently from an UnauthorizedAccessException. Planning for exceptions always involves **unexpected situations**. Sometimes those situations can be prevented, sometimes you want to handle them, and sometimes you want the exceptions to bubble up. A big lesson to learn here is that there's no "one-size-fits-all" approach to dealing with the unexpected, which is why the IDE doesn't just wrap everything in a try/catch block.

This is why there are so many classes that inherit from Exception, and why you may even want to write your own classes to inherit from Exception.

there are no
Dumb Questions

Q: I'm still not clear on when I should be catching exceptions, when I should be preventing them, and when I should let them bubble up.

A: That's because there's no single correct answer, or a rule that you can follow in every situation—it always depends on what you want your code to do.

Actually, that's not 100% true. There is a rule: it's always better to prevent exceptions if you can. You can't always anticipate the unexpected, especially when you're dealing with input from users or decisions they make.

Deciding whether to let exceptions bubble up or handle them in the class often boils down to separation of concerns. Does it make sense for a class to know about a certain exception? It depends on what that class does. Luckily, the IDE's refactoring tools are always there to help you change your code if you decide that certain exceptions are better off bubbled up than caught.

Q: Can you explain what you meant by “syntactic sugar?”

A: When developers use the term *syntactic sugar*, they're typically talking about a programming language giving you a convenient and easy-to-understand shortcut for code that would otherwise be more complex. The word “syntax” refers to the keywords of C# and the rules that govern them. The using statement is an official part of the C# syntax, which has rules that say that it must be followed by a variable declaration that instantiates a type that implements IDisposable, followed by a block of code. The “sugar” part refers to the fact that it's *super sweet* that the C# compiler turns it into something that would be much clunkier if you had to write it out by hand.

Q: Is it possible to use an object with a using statement if it doesn't implement IDisposable?

A: No. You can only create objects that implement IDisposable with using statements, because they're tailor-made for each other. Adding a using statement is just like creating a new instance of a class, except that it always calls its Dispose method at the end of the block. That's why the class **must implement** the IDisposable interface.

Q: Can you put any statement inside a using block?

A: Definitely. The whole idea with using is that it helps you make sure that every object you create with it is disposed of. But what you do with those objects is entirely up to you. In fact, you can create an object with a using statement and never even use it inside the block. That would be pretty useless, though, so we don't recommend doing that.

Q: Can you call the Dispose method outside of a using statement?

A: Yes. You don't ever actually *need* to use a using statement. You can call Dispose yourself when you're done with the object, or manually do whatever cleanup is necessary—like calling a stream's Close method, which you did in Chapter 10. If you use a using statement, it'll make your code easier to understand and prevent problems that happen if you don't dispose of your objects.

Q: Since a using statement basically generates a try/catch block that calls the Dispose method, is it OK to do exception handling inside of the using block?

A: Yes. It works exactly like the nested try/catch blocks that you used earlier in the chapter in your GetInputStream method.

Q: You mentioned a try/finally block. Does that mean it's OK to have a try and finally without a catch?

A: Yes! You can definitely have a try block without a catch, and just a finally. It looks like this:

```
try {
    DoSomethingRisky();
    SomethingElseRisky();
}
finally {
    AlwaysExecuteThis();
}
```

If DoSomethingRisky throws an exception, then the finally block will immediately run.

Q: Does Dispose only work with files and streams?

A: Not at all. There are a lot of classes that implement IDisposable, and when you're using one you should always use a using statement. If you write a class that has to be disposed of in a certain way, then you can implement IDisposable, too.

There's no one-size-fits-all approach to planning for the unexpected.

The worst catch block EVER: catch-all plus comments

A **catch** block will let your program keep running if you want it to. An exception gets thrown, you catch the exception, and instead of shutting down and giving an error message, you keep going. But sometimes, that's not such a good thing.

Take a look at this **Calculator** class, which seems to be acting funny all the time. What's going on?

```
class Calculator
{
    public void Divide(int dividend, int divisor)
    {
        try
        {
            this.quotient = dividend / divisor;
        } catch {

            // TODO: we need to figure out a way to prevent our
            // users from entering a zero in a division problem.

        }
    }
}
```

Here's the problem. If divisor is zero, this will create a `DivideByZeroException`.



The programmer thought that they could bury the exceptions by using an empty catch block, but they just caused a headache for whoever had to track down problems with it later.



You should handle your exceptions, not bury them

Just because you can keep your program running doesn't mean you've handled your exceptions. In the code above, the calculator won't crash...at least, not in the `Divide` method. What if some other code calls that method, and tries to print the results? If the divisor was zero, then the method probably returned an incorrect (and unexpected) value.

Instead of just adding a comment and burying the exception, you need to **handle the exception**. If you're not able to handle the problem, **don't leave empty or commented catch blocks!** That just makes it harder for someone else to track down what's going on. It's better to let the program continue to throw exceptions, because then it's easier to figure out what's going wrong.

Remember, when your code doesn't handle an exception, the exception bubbles up the call stack. Letting an exception bubble up is a perfectly valid way of dealing with an exception, and in some cases it makes more sense to do that than to use an empty catch block to bury the exception.



Temporary solutions are OK (temporarily)

Sometimes you find a problem, and know it's a problem, but aren't sure what to do about it. In these cases, you might want to log the problem and note what's going on. That's not as good as handling the exception, but it's better than doing nothing.

Here's a temporary solution to the calculator problem:

```
class Calculator
{
    public void Divide(int dividend, int divisor)
    {
        try
        {
            this.quotient = dividend / divisor;
        } catch (Exception ex) {

            using (StreamWriter sw = new StreamWriter(@"C:\Logs\errors.txt"));
            {
                sw.WriteLine(ex.getMessage());
            }
        }
    }
}
```

...but in real life,
"temporary" solutions
have a nasty habit of
becoming permanent.

**Take a minute and think
about this catch block.
What happens if the
StreamWriter can't write to
the C:\Logs\ folder? You
can nest another try/catch
block inside it to make it
less risky. Can you think of
a better way to do this?**



This still needs to be fixed, but short-term, this makes it clear where the problem occurred. Still, wouldn't it be better to figure out why your Divide method is being called with a zero divisor in the first place?

I GET IT- IT'S SORT OF
LIKE USING EXCEPTION HANDLING
TO PLACE A MARKER IN THE
PROBLEM AREA.



Handling exceptions doesn't always mean the same thing as FIXING exceptions.

It's never good to have your program bomb out. It's way worse to have no idea why it's crashing or what it's doing to users' data. That's why you need to be sure that you're always dealing with the errors you can predict and logging the ones you can't. While logs can be useful for tracking down problems, preventing those problems in the first place is a better, more permanent solution.

BULLET POINTS

- Any statement can throw an **exception** if something fails at **runtime** (or while the code is running).
- Use a **try/catch** block to handle exceptions. Unhandled exceptions will cause your program to stop execution and pop up an error window.
- Any exception in the block of code after the **try** statement will cause the program's execution to immediately jump to the first statement in the **exception handler**, or the block of code after **catch**.
- The **Exception object** gives you information about the exception that was caught. If you specify an **Exception** variable in your **catch** statement, that variable will contain information about any exception thrown in the **try** block:

```
try {  
    // statements that might  
    // throw exceptions  
} catch (IOException ex) {  
    // if an exception is thrown,  
    // ex has information about it  
}
```

- There are many **different kinds of exceptions** that you can catch. Each has its own object that extends **System.Exception**.
- Try to avoid **catch-all exception handlers** just catching **Exception**. Handle specific exceptions instead.
- A **finally block** after the exception handlers will always run whether or not an exception is thrown.

- Each **try** can have more than one **catch**:

```
try { ... }  
catch (NullReferenceException ex) {  
    // these statements will run if a  
    // NullReferenceException is thrown  
}  
catch (OverflowException ex) { ... }  
catch (FileNotFoundException) { ... }  
catch (ArgumentException) { ... }
```

- Your code can **throw an exception** using **throw**:

```
throw new Exception("Exception message");
```

- Your code can also **rethrow** an exception using **throw**; but this only works inside of a **catch** block. Rethrowing an exception preserves the call stack.

- You can create a **custom exception** by adding a class that extends the **System.Exception** base class:

```
class CustomException : Exception { }
```

- Most of the time, you only need to throw exceptions that are built into .NET, like **ArgumentException**. One reason to use different kinds of exceptions is to **give more information** to people troubleshooting problems.

- An **exception filter** uses the **when** keyword to tell your exception handler to catch an exception only under specific conditions.

- The **using** statement is **syntactic sugar** that causes the C# compiler to generate the equivalent of a **finally** block that calls the **Dispose** method.

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.

Now we'll pick up where the last Unity Lab left off. The goal of these labs is to get you familiar with Unity's **pathfinding and navigation system**, a sophisticated AI system that lets you create characters that can find their way around the worlds you create. In this lab, you'll use Unity's navigation system to make your GameObjects move themselves around a scene.

Along the way you'll learn about useful tools: you'll create a more complex scene and bake a NavMesh that lets an agent navigate it, you'll create static and moving obstacles, and most importantly, you'll **get more practice writing C# code**.

Let's pick up where the last Unity Lab left off

In the last Unity Lab, you created a player out of a sphere head nested under a cylinder body. Then you added a NavMesh Agent component to move the player around the scene, using raycasting to find the point on the floor that the player clicked. In this lab, you'll pick up where the last one left off. You'll add GameObjects to the scene, including stairs and obstacles so you can see how Unity's navigation AI handles them. Then you'll add a moving obstacle to really put that NavMesh Agent through its paces.

So go ahead and **open the Unity project** that you saved at the end of the last Unity Lab. If you've been saving up the Unity Labs to do them back to back, then you're probably ready to jump right in! But if not, take a few minutes and flip through the last Unity Lab again—and also look through the code that you wrote for it.

If you're using our book because you're preparing to be a professional developer, being able to go back and read and refactor the code in your old projects is a really important skill—and not just for game development!

there are no Dumb Questions

Q: There were a lot of moving parts in the last Unity Lab. Can you go through them again, just so I'm sure I have everything?

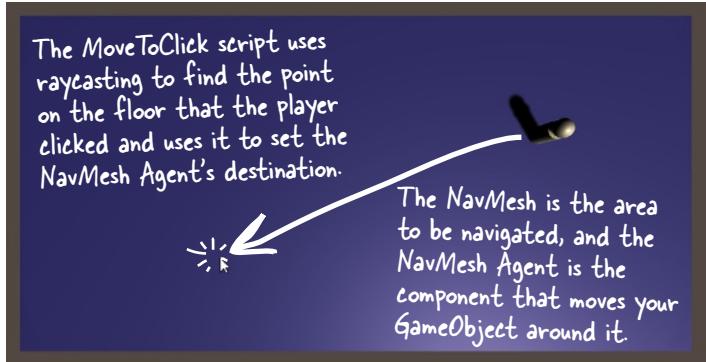
A: Definitely. The Unity scene you created in the last lab has four separate pieces. It's easy to lose track of how they work together, so let's go through them one by one:

1. First there's the **NavMesh**, which defines all of the "walkable" places your player can move around on in the scene. You made this by setting the floor as a walkable surface and then "baking" the mesh.

2. Next there's the **NavMesh Agent**, a component that can "take over" your GameObject and move it around the NavMesh just by calling its SetDestination method. You added this to your *Player* GameObject.

3. The camera's **ScreenPointToRay** method creates a ray that goes through a point on the screen. You added code to the Update method that checks if the player is currently pressing the mouse button. If so, it uses the current mouse position to compute the ray.

4. **Raycasting** is a tool that lets you cast (basically "shoot") a ray. Unity has a useful Physics.Raycast method that takes a ray, casts it up to a certain distance, and if it hits something tells you what it hit.



Q: So how do those parts work together?

A: Whenever you're trying to figure out how the different parts of a system work together, **understanding the overall goal** is a great place to start. In this case, the goal is to let the player click anywhere on the floor and have a GameObject move there automatically. Let's break that down into a set of individual steps. The code needs to:

- Detect that the player clicked the mouse.

Your code uses Input.GetMouseButtonDown to detect a mouse click.

- Figure out what point in the scene that click corresponds to.

It uses Camera.ScreenPointToRay and Physics.Raycast to do raycasting and figure out which point in the scene the player clicked.

- Tell the NavMesh Agent to set that point as a destination.

The NavMeshAgent.SetDestination method triggers the agent to calculate a new path and start moving towards the new destination.

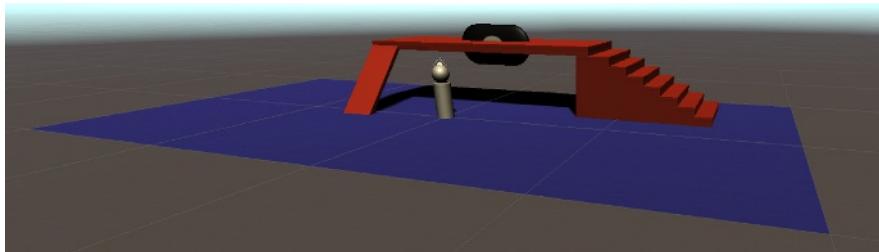
The MoveToClick method was adapted from code on the Unity Manual page for the NavMeshAgent.SetDestination method. Take a minute and read it right now—choose Help >> Scripting Reference from the main menu, then search for NavMeshAgent.SetDestination.

Unity Lab #6

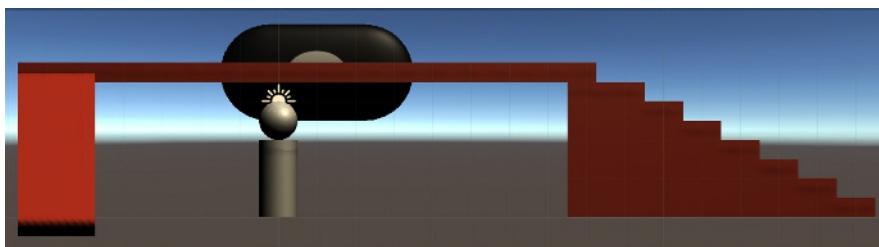
Scene Navigation

Add a platform to your scene

Let's do a little experimentation with Unity's navigation system. To help us do that, we'll add more GameObjects to build a platform with stairs, a ramp, and an obstacle. Here's what it will look like:



It's a little easier to see what's going on if we switch to an **isometric** view, or a view that doesn't show perspective. In a **perspective** view, objects that are further away look small, while closer objects look large. In an isometric view, objects are always the same size no matter how far away they are from the camera.



Add 10 GameObjects to your scene. Create a new material called **Platform** in your Materials folder with albedo color CC472F, and add it to all of the GameObjects except for Obstacle, which uses a **new material called 8 Ball** with the 8 Ball Texture map from the first Unity Lab. This table shows their names, types, and positions:

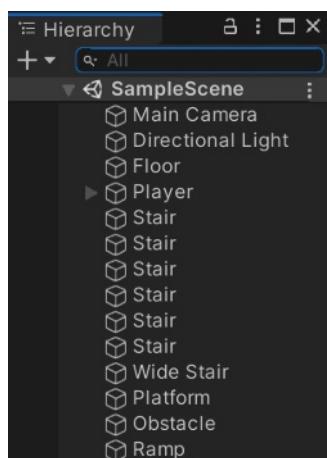
Name	Type	Position	Rotation	Scale
Stair	Cube	(15, 0.25, 5)	(0, 0, 0)	(1, 0.5, 5)
Stair	Cube	(14, 0.5, 5)	(0, 0, 0)	(1, 1, 5)
Stair	Cube	(13, 0.75, 5)	(0, 0, 0)	(1, 1.5, 5)
Stair	Cube	(12, 1, 5)	(0, 0, 0)	(1, 2, 5)
Stair	Cube	(11, 1.25, 5)	(0, 0, 0)	(1, 2.5, 5)
Stair	Cube	(10, 1.5, 5)	(0, 0, 0)	(1, 3, 5)
Wide stair	Cube	(8.5, 1.75, 5)	(0, 0, 0)	(2, 3.5, 5)
Platform	Cube	(0.75, 3.75, 5)	(0, 0, 0)	(15, 0.5, 5)
Obstacle	Capsule	(1, 3.75, 5)	(0, 0, 90)	(2.5, 2.5, 0.75)
Ramp	Cube	(-5.75, 1.75, 0.75)	(-46, 0, 0)	(2, 0.25, 6)

Try creating the first stair, then duplicating it five times and modifying its values.

A Capsule is like a cylinder that has spheres on its ends.

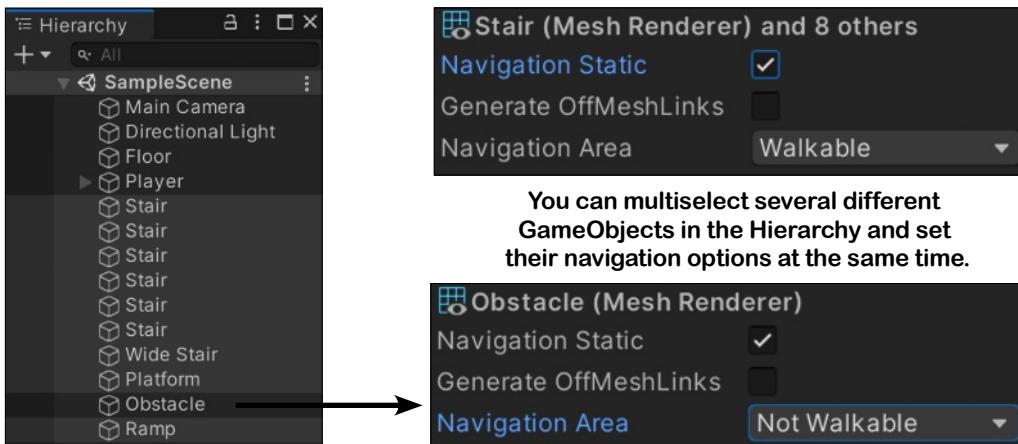
Sometimes it's easier to see what's going on in your scene if you switch to an isometric view. You can always reset the layout if you lose track of the view.

When you start Unity, the label (**<Persp**) under the Scene Gizmo shows the view name. The three lines (**—**) indicate that the Gizmo is in perspective mode. Click the cones to change the view to "Back" (**<Back**). If you click the lines they'll change to three parallel lines (**|||**), which toggles the view to isometric mode.

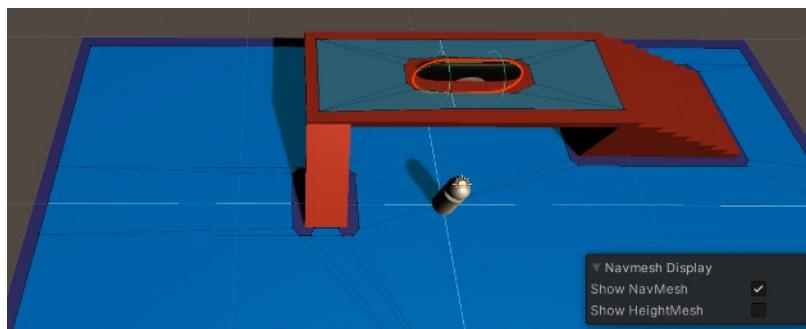


Use bake options to make the platform walkable

Use Shift-click to select all of the new GameObjects that you added to the scene, then use Control-click (or Command-click on a Mac) to deselect Obstacle. Go to the Navigation window and click the Object button, then **make them all walkable** by checking Navigation Static and setting the Navigation Area to Walkable. **Make the Obstacle GameObject not walkable** by selecting it, clicking Navigation Static, and setting Navigation Area to Not Walkable.

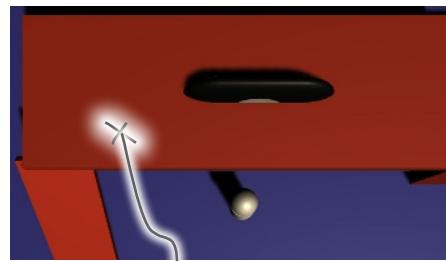


Now follow the same steps that you used before to **bake the NavMesh**: click the Bake button at the top of the Navigation window to switch to the Bake view, then click the Bake button at the bottom.

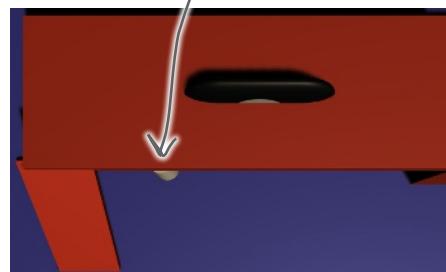


It looks like it worked! The NavMesh now appears on top of the platform, and there's space around the obstacle. Try running the game. Click on top of the platform and see what happens.

Hmm, hold on. Things aren't working the way we expected them to. When you click on top of the platform, the player goes **under** it. If you look closely at the NavMesh that's displayed when you're viewing the Navigation window, you'll see that it also has space around the stairs and ramp, but doesn't actually include either of them in the NavMesh. The player has no way to get to the point you clicked on, so the AI gets it as close as it can.



Clicking on top of the platform causes the player to go underneath it.



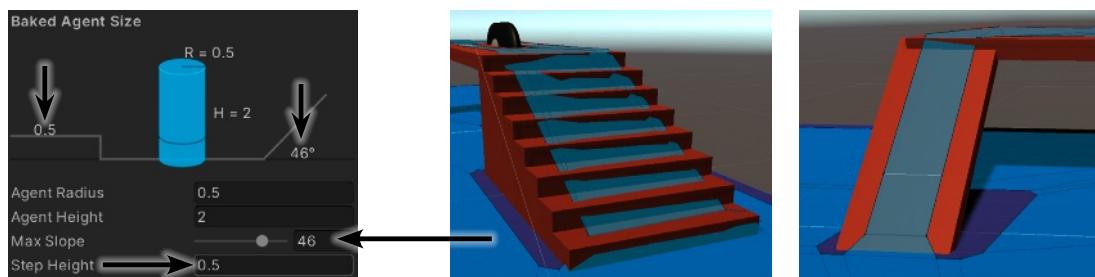
Unity Lab #6

Scene Navigation

Include the stairs and ramp in your NavMesh

An AI that couldn't get your player up or down a ramp or stairs wouldn't be very intelligent. Luckily, Unity's pathfinding system can handle both of those cases. We just need to make a few small adjustments to the options when we bake the NavMesh. Let's start with the stairs. Go back to the Bake window and notice that the default value of Step Height is 0.4. Take a careful look at the measurements for your steps—they're all 0.5 units tall. So to tell the navigation system to include steps that are 0.5 units, **change the Step Height to 0.5**. You'll see the picture of the step in the diagram get taller, and the number above it change from the default 0.4 value to 0.5.

We still need to include the ramp in the NavMesh. When you created the GameObjects for the platform, you gave the ramp an X rotation of -46, which means that it's a 46-degree incline. The Max Slope setting defaults to 45, which means it will only include ramps, hills, or other slopes with at most a 45-degree incline. So **change Max Slope to 46**, then **bake the NavMesh again**. Now it will include the ramp and stairs.



Start your game and test out your new NavMesh changes.



Here's another Unity coding challenge! Earlier, we pointed the camera straight down by setting its X rotation to 90. Let's see if we can get a better look at the player by making the arrow keys and mouse scroll wheel control the camera. You already know almost everything you need to get it to work—we just need to add a little code. *It may seem complicated, but you can do this!*

- **Create a new script called MoveCamera and drag it onto the camera.** It should have a Transform field called Player. Drag the Player GameObject out of the Hierarchy and **onto the Player field in the Inspector**. Since the field's type is Transform, it copies a reference to the Player GameObject's Transform component.
- **Make the arrow keys rotate the camera around the player.** Input.GetKeyDown(KeyCode.LeftArrow) will return true if the player is currently pressing the left arrow key, and you can use RightArrow, UpArrow, and DownArrow to check for the other arrow keys. Use this method just like you used Input.GetMouseButtonDown in your MoveToClick script to check for mouse clicks. When the player presses a key, call transform.RotateAround to rotate around the player's position. The player's position is the first argument; use Vector3.left, Vector3.right, Vector3.up, or Vector3.down as the second argument, and a field called Angle (set to 3F) as the third argument.
- **Make the scroll wheel zoom the camera.** Input.GetAxis("Mouse ScrollWheel") returns a number (usually between -0.4 and 0.4) that represents how much the scroll wheel moved (or 0 if it didn't move). Add a float field called ZoomSpeed set to 0.25F. Check if the scroll wheel moved. If it did, do a little vector arithmetic to zoom the camera by multiplying transform.position by (1F + scrollWheelValue * ZoomSpeed).
- **Point the camera at the player.** The transform.LookAt method makes a GameObject look at a position. Then reset the Main Camera's Transform to position (0, 1, -10) and rotation (0, 0, 0).

It's not cheating to peek at the solution!

Unity Lab #6

Scene Navigation

It's OK if your code looks a little different than ours as long as it works. There are a LOT of ways to solve any coding problem! Just make sure to take some time and understand how this code works.



Exercise Solution

Here's another Unity coding challenge! We pointed the camera straight down by setting its X rotation to 90. Let's see if we can get a better look at the player by making the arrow keys and mouse scroll wheel control the camera. You already know almost everything you need to get it to work—we just need to give you a little code to include. *It seems like a lot, but you can do this!*

```
public class MoveCamera : MonoBehaviour
{
    public Transform Player;
    public float Angle = 3F;
    public float ZoomSpeed = 0.25F;

    void Update()
    {
        var scrollWheelValue = Input.GetAxis("Mouse ScrollWheel");
        if (scrollWheelValue != 0)
        {
            transform.position *= (1F + scrollWheelValue * ZoomSpeed);
        }

        if (Input.GetKey(KeyCode.RightArrow))
        {
            transform.RotateAround(Player.position, Vector3.up, Angle);
        }
        if (Input.GetKey(KeyCode.LeftArrow))
        {
            transform.RotateAround(Player.position, Vector3.down, Angle);
        }

        if (Input.GetKey(KeyCode.UpArrow))
        {
            transform.RotateAround(Player.position, Vector3.right, Angle);
        }
        if (Input.GetKey(KeyCode.DownArrow))
        {
            transform.RotateAround(Player.position, Vector3.left, Angle);
        }

        transform.LookAt(Player.position);
    }
}
```

Did you remember to reset the Main Camera's position and rotation? If not, it may look a little jumpy when the player starts moving (it's because of the way the angle is computed in the Camera.LookAt method).

This is an example of how simple vector arithmetic can simplify a task. A GameObject's position is a vector, so multiplying it by a 1.02 moves it a little further away from the zero point, and multiplying it by .98 moves it a little closer.

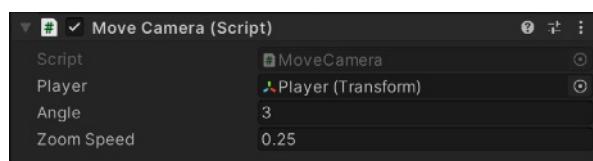
→ This is just like how you used transform.RotateAround in the last two Unity Labs, except instead of rotating around Vector3.zero (0,0,0) you're rotating around the player.

We asked you to create the Player field with type Transform. That gives you a reference to the Player GameObject's Transform component, so Player.position is the player's position.

Use the arrow keys to move the camera so it's looking up at the player. You can see right through the floor plane!

Don't forget to drag Player onto the field in the script component in the Main Camera. →

Try experimenting with different angles and zoom speeds to see what feels best to you.



Unity Lab #6

Scene Navigation

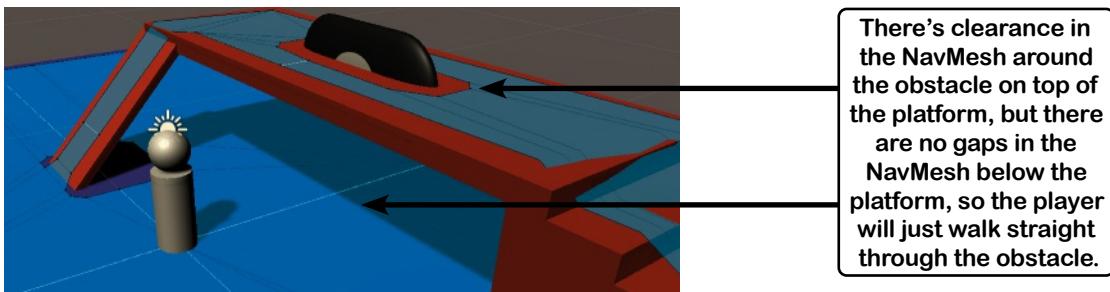
Fix height problems in the NavMesh

Now that we've got control of the camera, we can get a good look at what's going on under the platform—and something doesn't look quite right. Start your game, then rotate the camera and zoom in so you can get a clear view of the obstacle sticking out under the platform. Click the floor on one side of the obstacle, then the other. It looks like the player is going right through the obstacle! And it goes right through the end of the ramp, too.

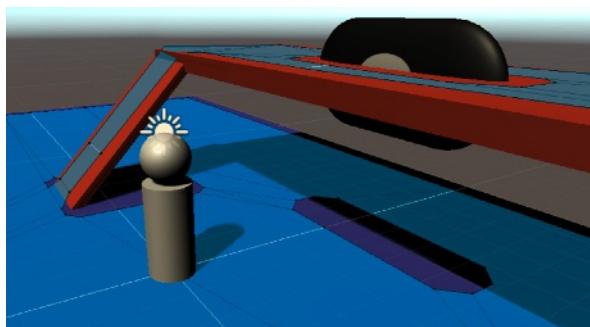
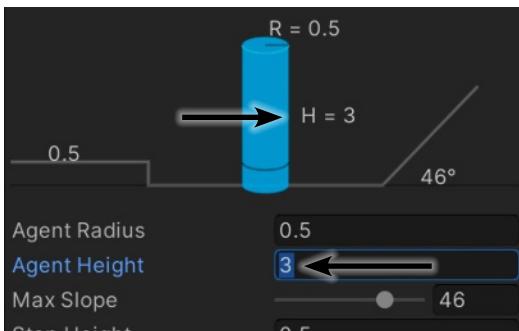


But if you move the player back to the top of the platform, it avoids the obstacle just fine. What's going on?

Look closely at the parts of the NavMesh above and below the obstacle. Notice any differences between them?



Go back to the part of the last lab where you set up the NavMesh Agent component—specifically, the part where you set the Height to 3. Now you just need to do the same for the NavMesh. Go back to the Bake options in the Navigation window and **set the Agent Height to 3, then bake your mesh again.**

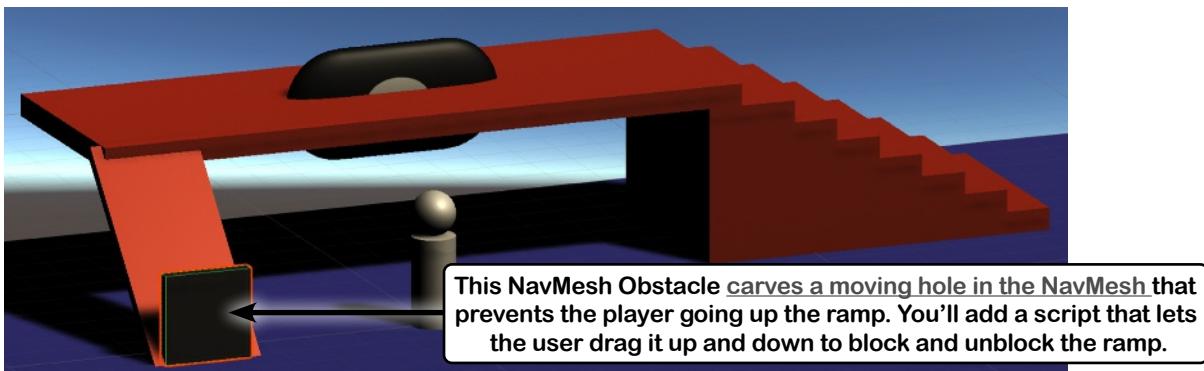


This created a gap in the NavMesh under the obstacle and expanded the gap under the ramp. Now the player doesn't hit either the obstacle or the ramp when moving around under the platform.

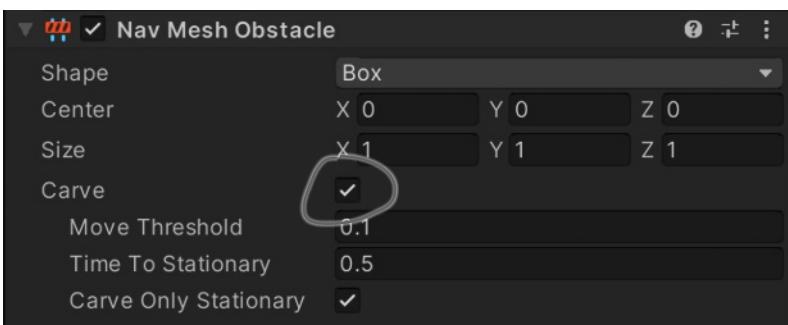
Add a NavMesh Obstacle

You already added a static obstacle in the middle of your platform: you created a stretched-out capsule and marked it non-walkable, and when you baked your NavMesh it had a hole around the obstacle so the player has to walk around it. What if you want an obstacle that moves? Try moving the obstacle—the NavMesh doesn't change! It still has a hole *where the obstacle was*, not where it currently is. If you bake it again, it just creates a hole around the obstacle's new location. To add an obstacle that moves, add a **NavMesh Obstacle component** to a GameObject.

Let's do that right now. **Add a Cube to your scene** with position $(-5.75, 1, -1)$ and scale $(2, 2, 0.25)$. Create a new material for it with a dark gray color (333333) and name your new GameObject *Moving Obstacle*. This will act as a kind of gate at the bottom of the ramp that can move up out of the way of the player or down to block it.



We just need one more thing. Click the Add Component button at the bottom of the Inspector window and choose Navigation >> Nav Mesh Obstacle to **add a NavMesh Obstacle component** to your Cube GameObject.



The Shape, Center, and Size properties let you create an obstacle that only partially blocks NavMesh Agents. If you have an oddly shaped GameObject, you can add several NavMesh Obstacle components to create a few different holes in the NavMesh.

If you leave all of the options at their default settings, you get an obstacle that the NavMesh Agent can't get through. Instead, the Agent hits it and stops. **Check the Carve box**—this causes the obstacle to **create a moving hole in the NavMesh** that follows the GameObject. Now your Moving Obstacle GameObject can block the player from navigating up and down the ramp. Since the NavMesh height is set to 3, if the obstacle is less than 3 units above the floor it will create a hole in the NavMesh underneath it. If it goes above that height, the hole disappears.

The Unity Manual has thorough—and readable!—explanations for the various components. Click the Open Reference button (?) at the top of the Nav Mesh Obstacle panel in the Inspector to open up the manual page. Take a minute to read it—it does a great job of explaining the options.

Unity Lab #6

Scene Navigation

Add a script to move the obstacle up and down

This script uses the **OnMouseDown** method. It works just like the OnMouseDown method you used in the last lab, except that it's called when the GameObject is dragged.

```
public class MoveObstacle : MonoBehaviour
{
    void OnMouseDown()
    {
        transform.position += new Vector3(0, Input.GetAxis("Mouse Y"), 0);
        if (transform.position.y < 1) {
            transform.position = new Vector3(transform.position.x, 1, transform.position.z);
        }
        if (transform.position.y > 5) {
            transform.position = new Vector3(transform.position.x, 5, transform.position.z);
        }
    }
}
```

You used `Input.GetAxis` earlier to use the scroll wheel. Now you're using the mouse's up-down movement—along the Y axis—to move the obstacle by modifying its Y position.

The first `if` statement keeps the block from moving below the floor, and the second keeps it from moving too high. Can you figure out how they work?

Drag your script onto the **Moving Obstacle** GameObject and run the game—uh-oh, something's wrong. You can click and drag the obstacle up and down, but it also moves the player. Fix this by **adding a tag** to the GameObject.



Set the tag for the obstacle just like you did in the last lab, but this time choose “Add tag...” from the dropdown, then use the **+** button to add a new tag called **Obstacle**. Now you can use the dropdown to assign the tag to the GameObject.

Then **modify your MoveToClick script** to check for the tag:

```
hit.collider if (Physics.Raycast(ray, out hit, 100))
contains a → if (hit.collider.gameObject.tag != "Obstacle")
reference to { agent.SetDestination(hit.point);
the object that } the ray hit. }
```

Run your game again. If you click on the obstacle you can drag it up and down, and it stops when it hits the floor or gets too high. Click anywhere else, and the player moves just like before. Now you can **experiment with the NavMesh Obstacle options** (this is easier if you reduce the Speed in the Player's NavMesh Agent):

- ★ Start your game. Click on *Moving Obstacle* in the Hierarchy window and **uncheck the Carve option**. Move your player to the top of the ramp, then click at the bottom of the ramp—the player will bump into the obstacle and stop. Drag the obstacle up, and the player will continue moving.
- ★ Now **check Carve** and try the same thing. As you move the obstacle up and down, the player will recalculate its route, taking the long way around to avoid the obstacle if it's down, and changing course in real time as you move the obstacle.

there are no Dumb Questions

Q: How does that `MoveObstacle` script work? It's using `+=` to update `transform.position`—does that mean it's using vector arithmetic?

A: Yes, and this is a great opportunity to understand vector arithmetic better. `Input.GetAxis` returns a number that's positive if the mouse moves up and negative if the mouse moves down (try adding a `Debug.Log` statement so you can see its value). The obstacle starts at position $(-5.75, 1, -1)$. If the player moves the mouse up and `GetAxis` returns 0.372 , the `+=` operation adds $(0, 0.372, 0)$ to the position. That means it **adds both of the X values** to get a new X value, then does the same for the Y and Z values. So the new Y position is $1 + 0.372 = 1.372$, and since we're adding 0 to the X and Z values, only the Y value changes and it moves up.

Get creative!

Can you find ways to improve your game and get practice writing code? Here are some ideas to help you get creative:

- ★ Build out the scene—add more ramps, stairs, platforms, and obstacles. Find creative ways to use materials. Search the web for new texture maps. Make it look interesting!
- ★ Make the NavMesh Agent move faster when the player holds down the Shift key. Search for “KeyCode” in the Scripting Reference to find the left/right Shift key codes.
- ★ You used OnMouseDown, Rotate, RotateAround, and Destroy in the last lab. See if you can use them to create obstacles that rotate or disappear when you click them.
- ★ We don’t actually have a game just yet, just a player navigating around a scene. Can you find a way to **turn your program into a timed obstacle course?**

You already know enough about Unity to start building interesting games—and that's a great way to get practice so you can keep getting better as a developer.

This is your chance to experiment. Using your creativity is a really effective way to quickly build up your coding skills.

BULLET POINTS

- When you bake the NavMesh, you can specify a **maximum slope and step height** to let NavMesh Agents navigate ramps and stairs in the scene.
- You can also **specify the agent height** to create holes in the mesh around obstacles that are too low for the agent to get around.
- When a NavMesh Agent moves a GameObject around a scene, it will **avoid obstacles** (and, optionally, other NavMesh Agents).
- The label under the Scene Gizmo shows an icon to indicate if it is in **perspective** mode (distant objects look smaller than near objects) or **isometric** mode (all objects appear the same size no matter how far away they are). You can use this icon to toggle between the two views.
- The **transform.LookAt method** makes a GameObject look at a position. You can use it to make the camera point at a GameObject in the scene.
- Calling **Input.GetAxis("Mouse ScrollWheel")** returns a number (usually between -0.4 and 0.4) that represents how much the scroll wheel moved (or 0 if it didn't move).
- Calling **Input.GetAxis("Mouse Y")** lets you capture mouse movements up and down. You can combine it with OnMouseDrag to move a GameObject with the mouse.
- Add a **NavMesh Obstacle** component to create obstacles that can carve moving holes in the NavMesh.
- The Input class has methods to capture input during the Update method, like **Input.GetAxis** for mouse movement and **Input.GetKey** for keyboard input.

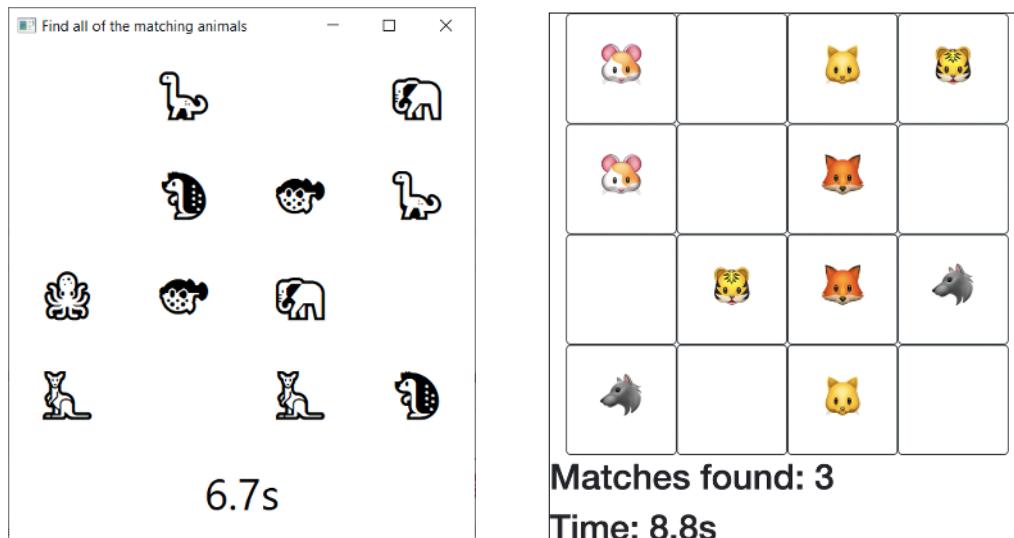
PDF



Downloadable exercise: Animal match boss battle

If you've played a lot of video games (and we're pretty sure that you have!), then you've had to play through a whole lot of boss battles—those fights at the end of a level or section where you face off against an opponent that's bigger and stronger than what you've seen so far. We have one last challenge for you before the end of the book—consider it the *Head First C#* boss battle.

In Chapter 1 you built an animal matching game. It was a great start, but it's missing... something. Can you figure out how to turn your animal matching game into a memory game? Go to our GitHub page and download the PDF for this project—or if you want to play this boss battle in Hard mode, just dive right in and see if you can do it on your own.



There's so much more downloadable material! The book may be over, but we can keep the learning going. We've put together even more downloadable material on important C# topics. We've also continued the Unity learning path with additional Unity Labs and even a Unity boss battle.

We hope you've learned a lot—and even more importantly, we hope your C# learning journey is just beginning. Great developers never stop learning.

Head to our GitHub page for more: <https://github.com/head-first-csharp/fourth-edition>.

Thank you for reading our book!

Pat yourself on the back—this is a real accomplishment! We hope this journey has been as rewarding for you as it has been for us, and that you've enjoyed all of the projects and the code that you've written along the way.

But wait, there's more! Your journey's just begun...

In some chapters we gave you additional projects that you could download from our GitHub page: <https://github.com/head-first-csharp/fourth-edition>.

The GitHub page contains **lots of additional material**. There's still more to learn, and more projects to do!

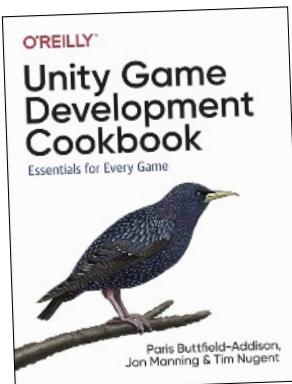
Keep your C# learning journey going by downloading PDFs that continue the *Head First C#* story and cover **essential C# topics**, including:

- ★ Event handlers
- ★ Delegates
- ★ The MVVM pattern (including a retro arcade game project)
- ★ ...and more!

And while you're there, there's **more to learn about Unity**. You can download:

- ★ PDF versions of all of the Unity Labs in this book
- ★ **Even more Unity Labs** that cover physics, collisions, and more!
- ★ A **Unity Lab boss battle** to put your Unity development skills to the test
- ★ A complete **Unity Lab project** to create a game from the ground up

And check out these essential (and amazing!) books by some of our friends and colleagues, also published by O'REILLY®



Unity Game Development Cookbook will help you take your Unity skills to the next level. It's full of incredibly valuable tools and techniques, all in a "recipe" form that you can use right away in your own projects.

C# 8.0 in a Nutshell is a must-have for every C# developer. If you need to do a deep dive into any part of C#, we think you can't do better than this book.

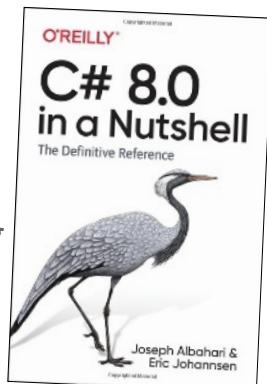


Check out these great C# and .Net resources!

Get connected with the .NET Developer Community: <https://dotnet.microsoft.com/platform/community>.

Watch live streams and chat with the team that builds .NET and C#: <https://dotnet.microsoft.com/platform/community/standup>.

Learn more in the docs: <https://docs.microsoft.com/en-us/dotnet>.



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.

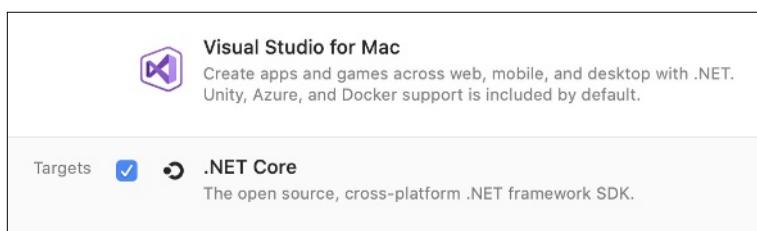
Visual Studio is your gateway to C#

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.

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:



- ➊ **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

- ➋ **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.

- ➌ **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.

- ➍ **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#.

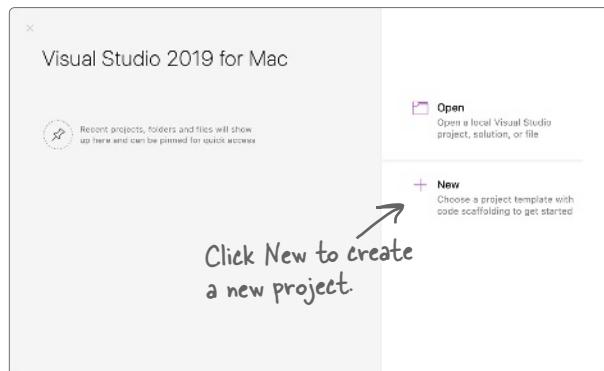
jump right in

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!

1 Create a new Console Project.

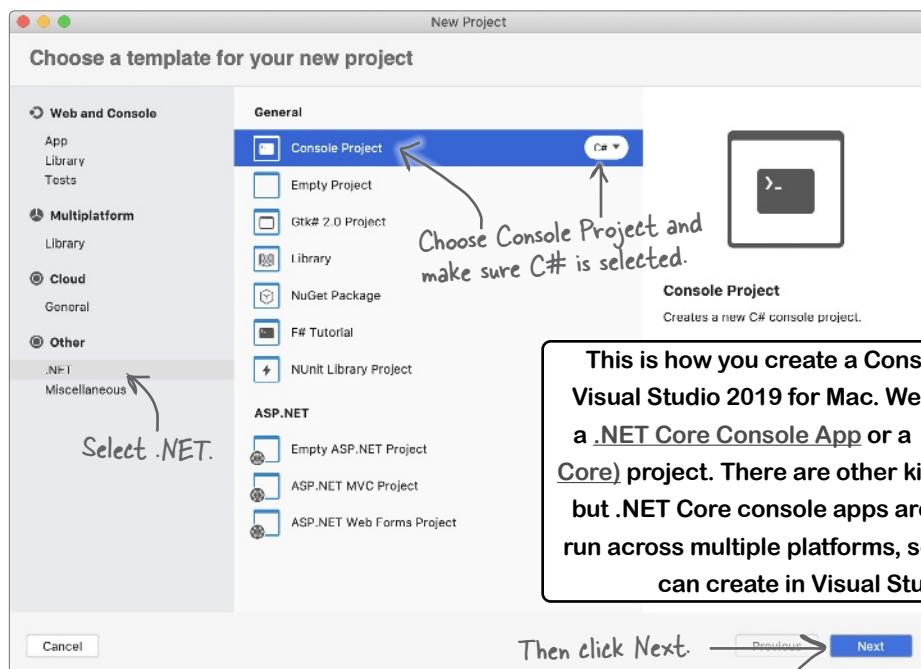
Start up Visual Studio 2019 for Mac. When it starts, it shows you a window that lets you create a new project or open an existing one. **Click New** to create a new project. Don't worry if you dismiss the window—you can always get it back by choosing *File >> New Solution... (⇧⌘N)* from the menu.



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.

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**

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 x
No selection
1  using System;
2
3  namespace MyFirstConsoleApp
4  {
5      class MainClass
6      {
7          public static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
13 }
```

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.****Watch it!**

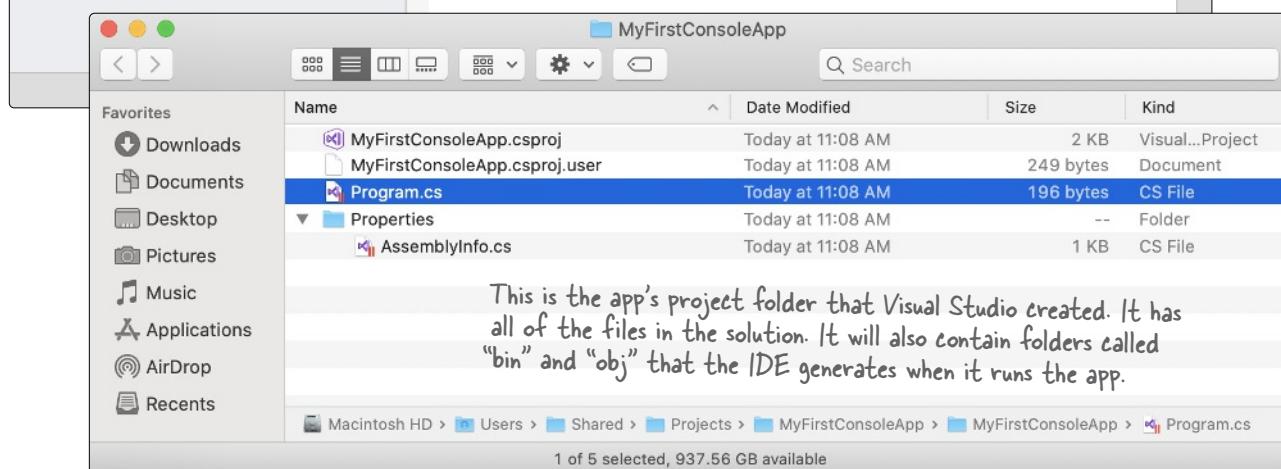
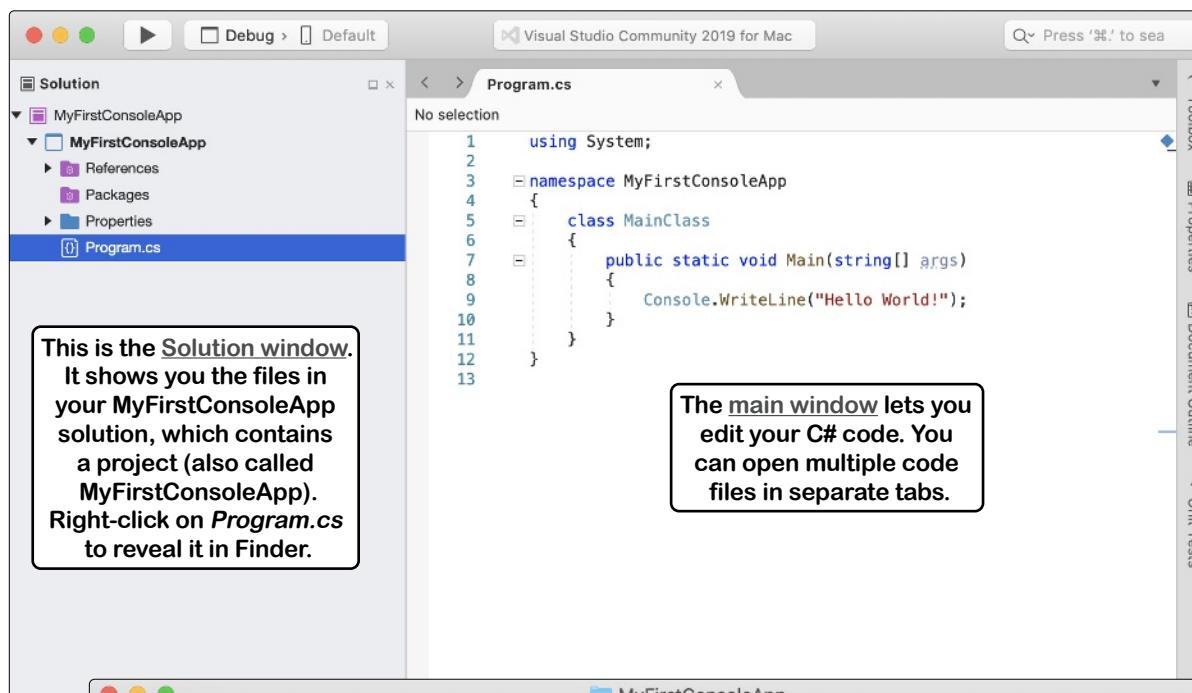
When you create a console app in Visual Studio for Windows, it generates almost exactly the same code as Visual Studio for Mac. The one difference is that on a Mac the main class is called MainClass, while on Windows the main class is called Program. That won't make a difference in most of the projects in this book. We'll make sure to point out any place that it does.

use the ide to run your app

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.



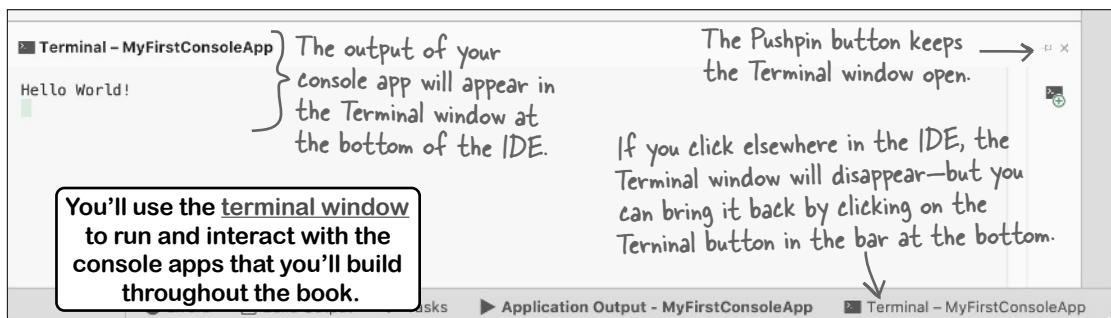
② Run your new app.

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:



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 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 ($\text{⌘} \leftarrow$). 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:



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:

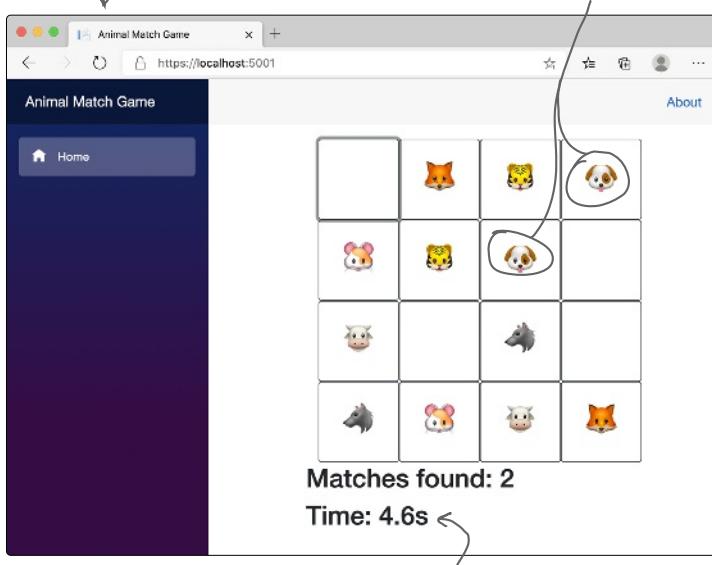


Let's build a game!

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.



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 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.

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.

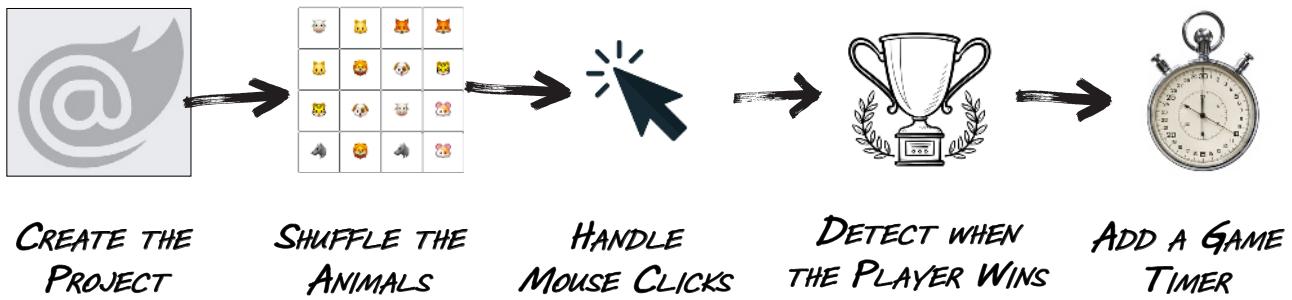
By the time you're done with this project, you'll be a lot more familiar with the tools that you'll rely on throughout this book to learn and explore C#.

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 any kind of project, not just video games.



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:

Game design... and beyond

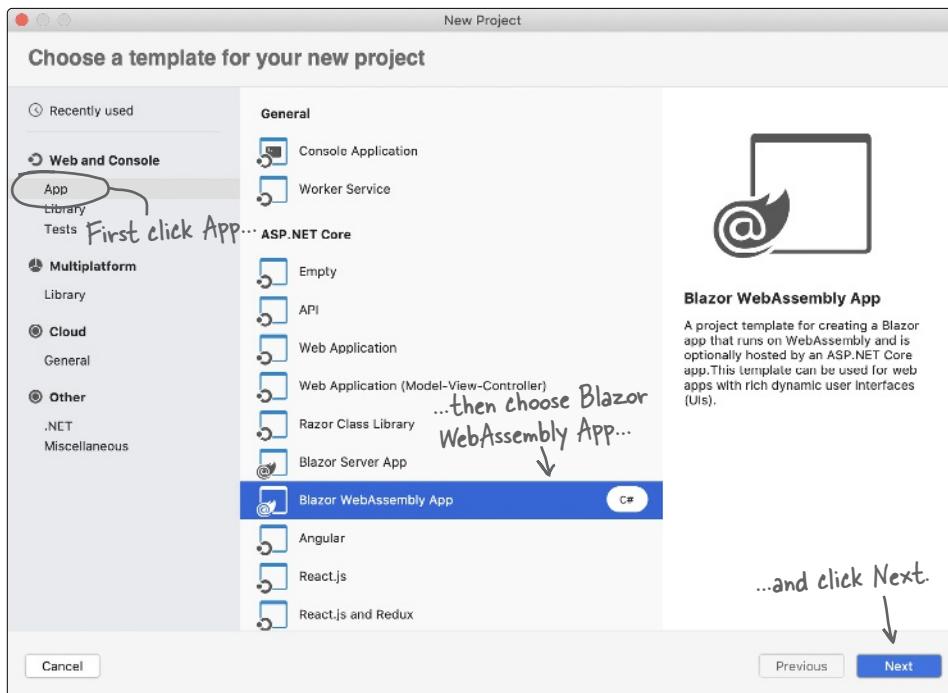
A game is a program that lets you play with it in a way that (hopefully) is at least as entertaining to play as it is to build.



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.



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.



Leave all of the options set to their default values and click **Next**.

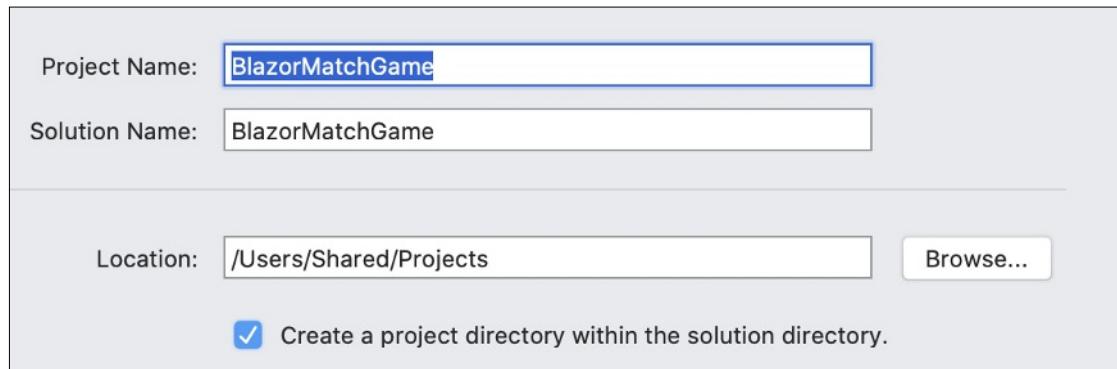


If you run into any trouble with this project, go to our GitHub page and look for a link to a video walkthrough:

<https://github.com/head-first-csharp/fourth-edition>

start building with c#

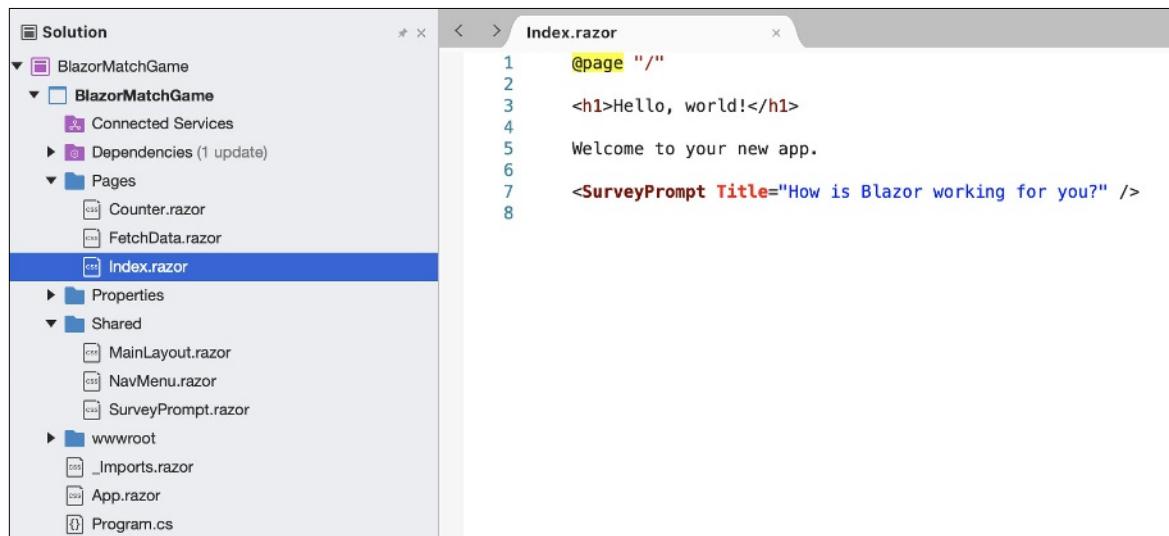
- 3 Enter **BlazorMatchGame** as the project name, just like you did with your Console App project.



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.



your app runs in a browser

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.

➊ Choose the browser to run your web application.

Find the triangle-shaped Run button at the top of the Visual Studio IDE:

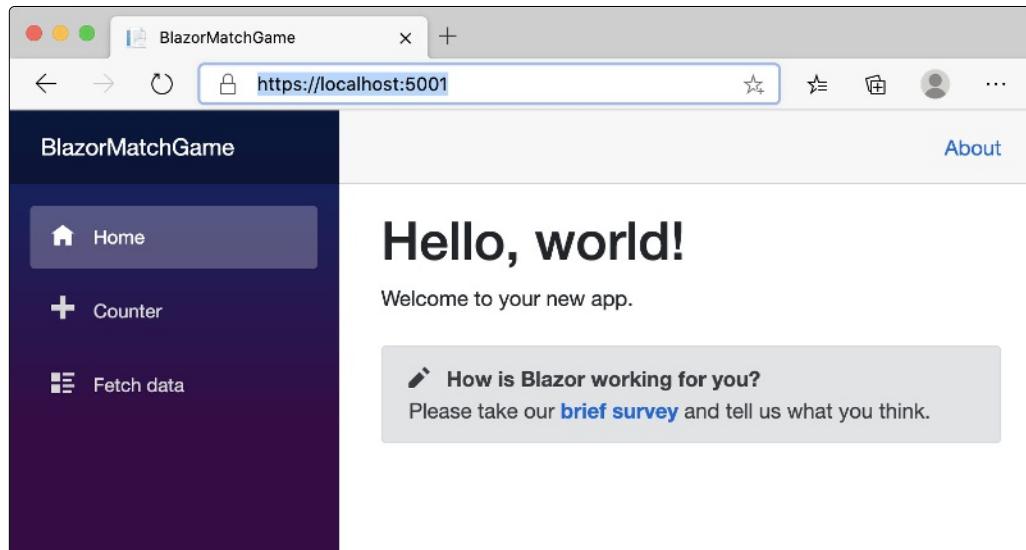
Do this!



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**.

➋ Run your web application.

Click the **Run button** to start your application. You can also choose Start Debugging ($\text{⌘} \leftarrow$) 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.



Watch it!

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.

```

1 @page "/"
2
3 <h1>Hello, world!</h1>
4
5 Welcome to your new app.
6
7 <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 ($\square \text{⌘} \leftarrow$) 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.

BlazorMatchGame

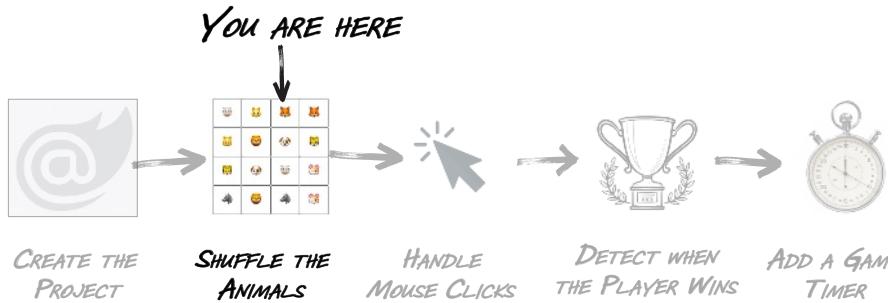
Elementary, my dear Watson.

Welcome to your new app.

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

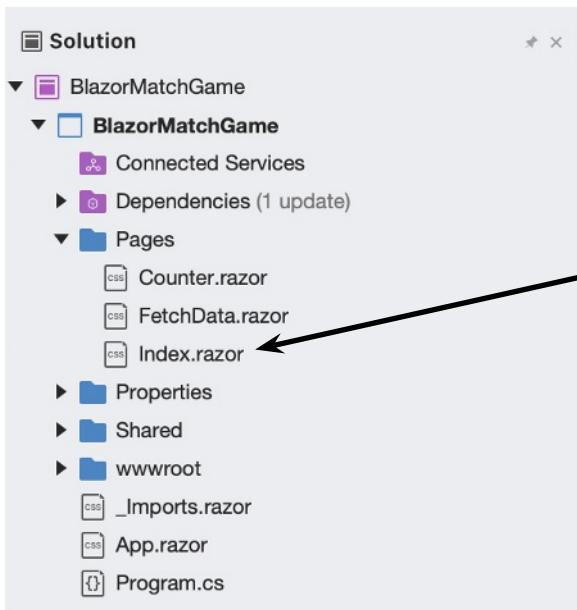
Do you have extra instances of your browser open? Visual Studio opens a new browser each time you run your Blazor web app. Get in the habit of closing the browser ($\text{⌘} Q$) before you stop your app ($\square \text{⌘} \leftarrow$).

start building your game



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).



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.

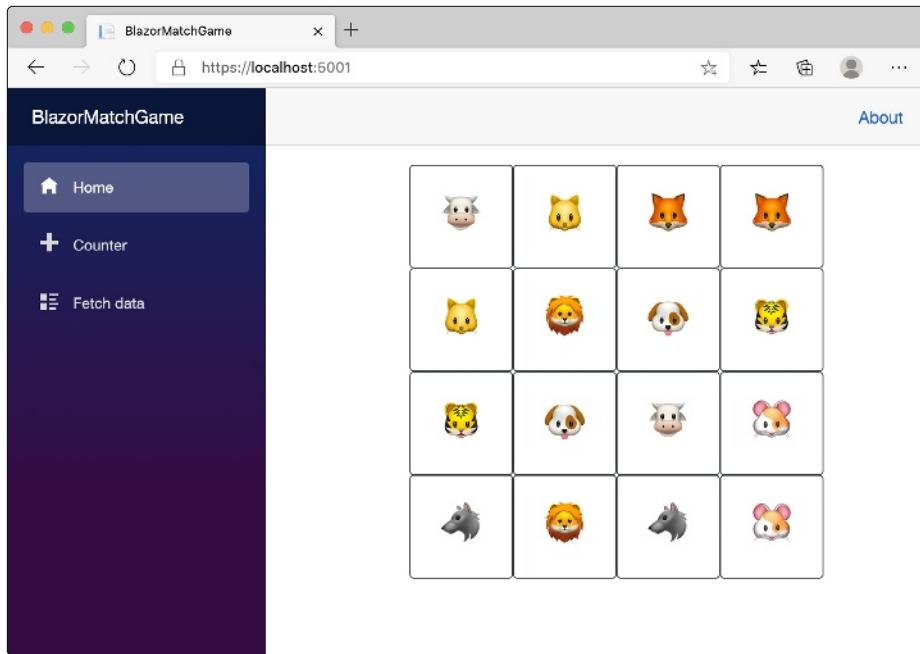
When you created your console app earlier in the chapter, your C# code was in a file called `Program.cs`—when you see that `.cs` file extension, it tells you that the file contains C# code.



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.



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.

add some emoji to your app

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.

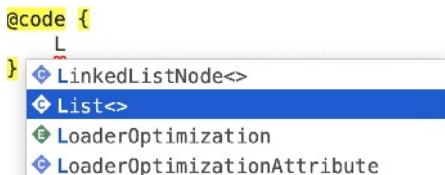
① Add C# code to your `Index.razor` file.

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:

```
9  @code {  
10  
11 }
```

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:



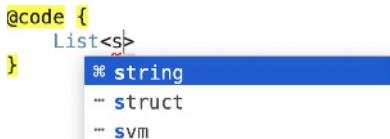
```
@code {  
    L  
}  
List<>  
LinkedListNode<>  
List<>  
LoaderOptimization  
LoaderOptimizationAttribute
```

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** (`<`)—the IDE will automatically fill in the closing bracket `>` and leave your cursor positioned between them.

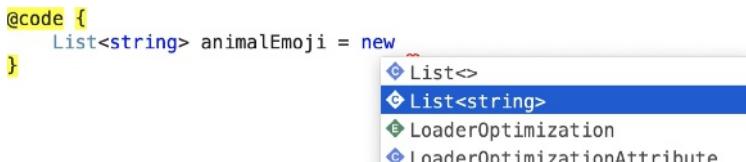
③ Start creating a List to store your animal emoji.

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”**:



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.

the plural of emoji is emoji

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>()
    {
        "🐺", "🐺",
        "🐴", "🐴",
        "🐱", "🐱",
        "🦊", "🦊",
        "🐯", "🐯",
        "🦁", "🦁",
        "🐯", "🐯",
        "🐹", "🐹",
    };
}
```

Replace the contents of the page

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.

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:

```
1  @page "/"
2
3  <st
4      > datalist
5      > strong
6      > style
```

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>
```

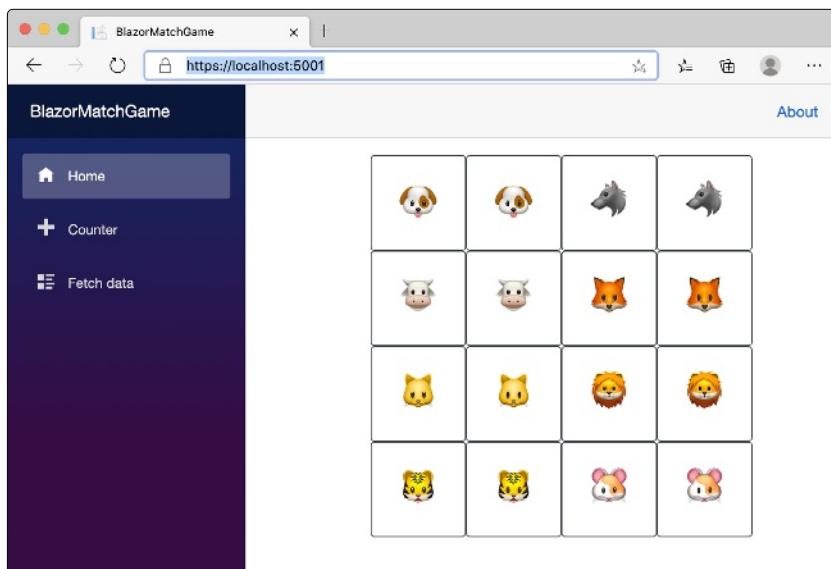
The matching game is made up of a series of buttons. This is a really simple CSS stylesheet to set the total width of the container, and the height and width of each button. Since the container is 400 pixels wide, the page will only allow four columns in a row before adding a break, making them appear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

```
M OnAfterRender(bool firstRender)
M OnAfterRenderAsync(bool firstRender)
M OnInitialized()
M OnInitializedAsync()
M OnParametersSet()
M OnParametersSetAsync()
M ShouldRender()
```

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

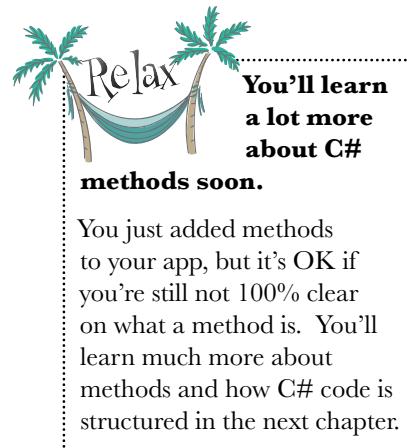
```
protected override void OnInitialized()
{
    base.OnInitialized();
}
```

- 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()) ← You'll learn more about
        .ToList();                                methods in Chapter 2.
}
```



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

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:



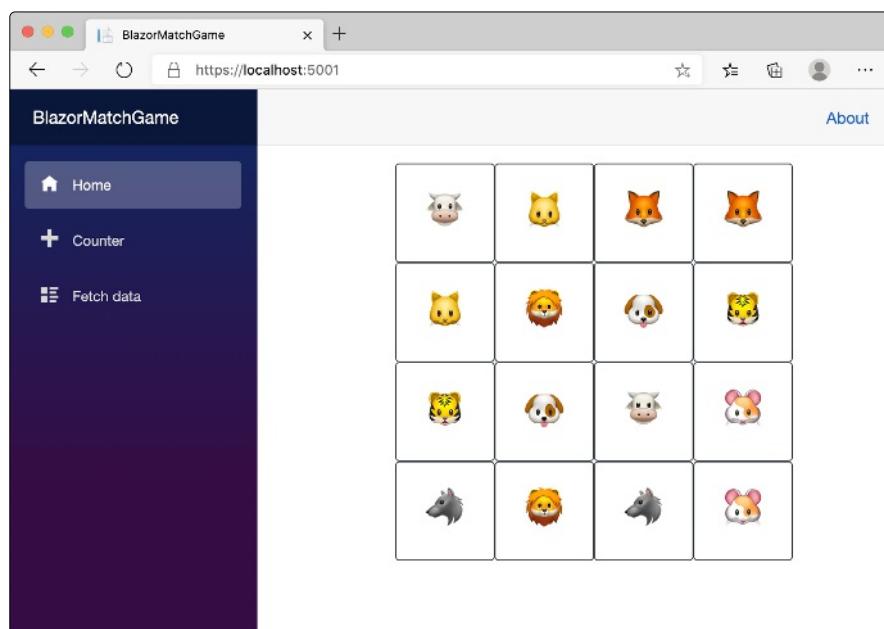
```

@foreach (var animal in s)
{
    <div class="col-md-3">
        <button type="button" value="<h1>@animal</h1>"></button>
    </div>
}

```

The screenshot shows a portion of a Blazor component's HTML code. A tooltip-like IntelliSense window is open over the variable `s` in the `@foreach` loop. The window lists several options starting with 's', with `shuffledAnimals` highlighted in blue. Below the list, a note says '(field) List<string> Index.shuffledAnimals'.

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.



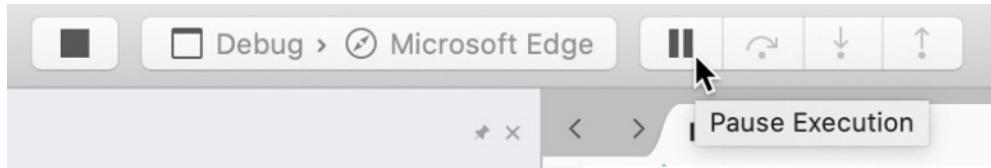
Again, make sure your app looks like this screenshot when you run it. Once it does, you'll know you entered all of the code without any typos. Don't move on until your game is reshuffling the animals every time you reload the browser page.

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

What it does

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

Create a second list to store the shuffled emoji

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

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

The beginning of a method that sets up the game

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

Create a list of eight pairs of emoji

```
private void SetUpGame()
{
```

Set up the game every time the page is reloaded

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

Create a new random number generator

```
shuffledAnimals = animalEmoji
    .OrderBy(item => random.Next())
    .ToList();
```

The end of a method that sets up the game

```
}
```

WHO DOES WHAT? solution

C# statement**What it does**

```

List<string> animalEmoji = new List<string>()
{
    "🐶", "🐱",
    "🐴", "🐎",
    "🐯", "🐅",
    "🦊", "🦊",
    "🐹", "🐹",
    "🐯", "🐯",
    "🦁", "🦁",
    "🐹", "🐹",
};

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();
}

```

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



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 mistakes (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

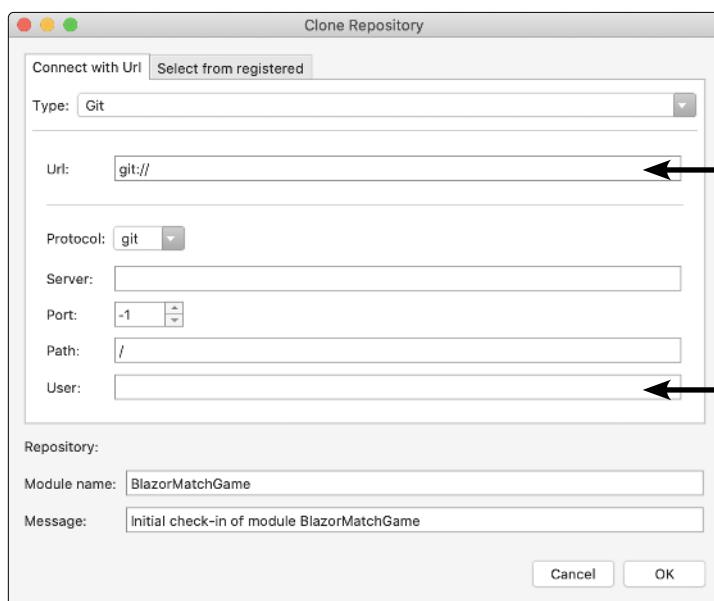
- Visual Studio is **Microsoft's IDE**—or **integrated development environment**—that simplifies and assists in editing and managing your C# code files.
- **.NET Core console apps** are cross-platform apps that use text for input and output.
- **Blazor WebAssembly apps** let you build rich interactive 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.

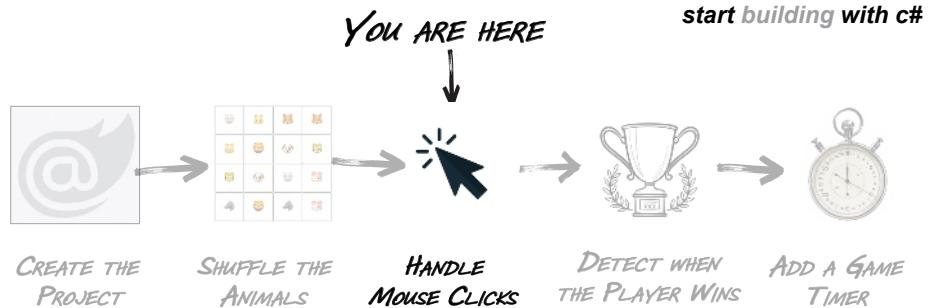
Once you have your GitHub account set up, you can use the built-in version control features of the IDE. **Choose Version Control >> Publish in Version Control... from the menu** to bring up the Clone Repository window:



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.

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.

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;
    }
}
```

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.

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();
```

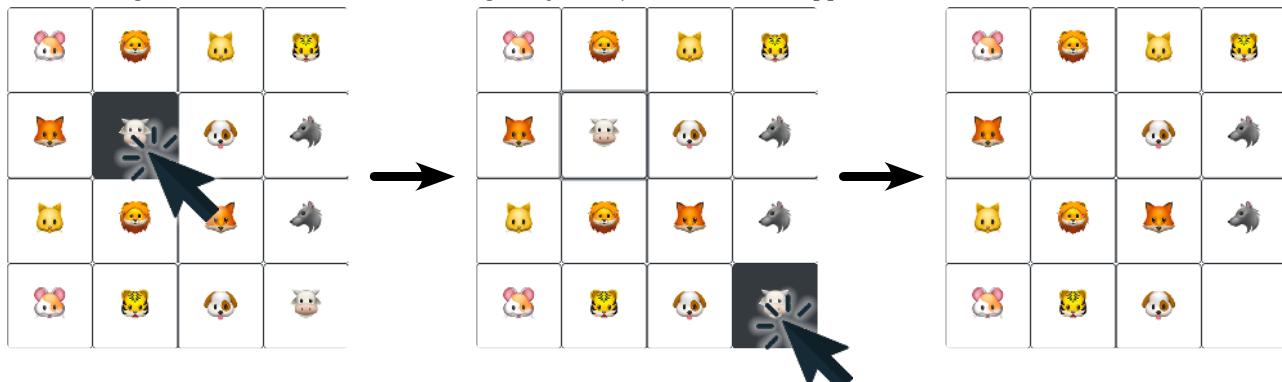
You'll find this statement in the code twice: in the → `lastAnimalFound = string.Empty;` section that's run if the second animal the player clicked matches the first, and in the section that's run if the second animal doesn't match. It blanks out the last animal found to reset the game so the next button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

**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.**

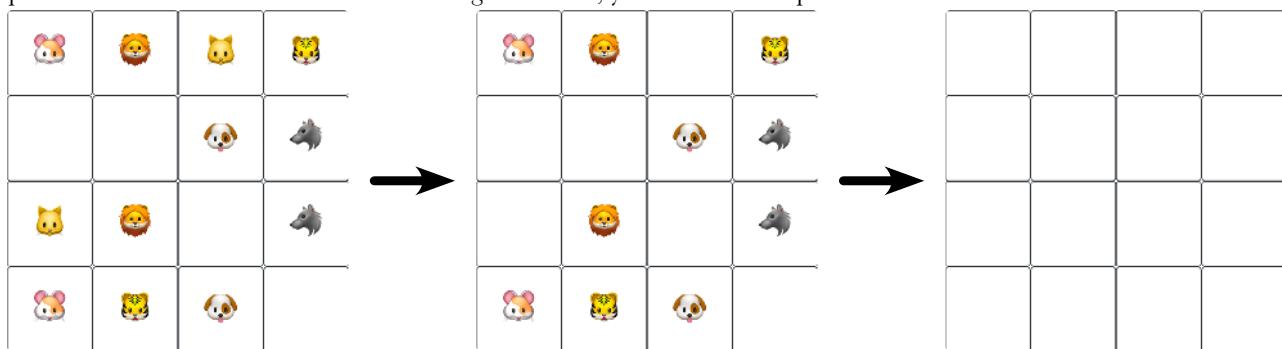
oops there's a bug

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.



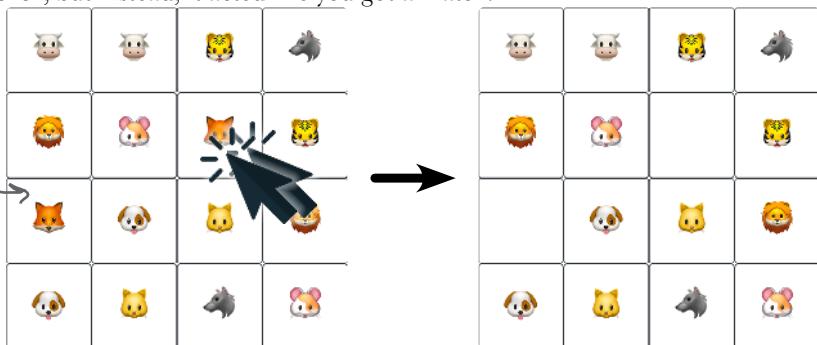
Click on another, then another, then another. You should be able to keep clicking on pairs until all of the buttons are blank. Congratulations, you found all the pairs!



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:

```
62     private void ButtonClick(string animal)
63     {
64         if (lastAnimalFound == string.Empty) ←
65         {
66             //First selection of the pair. Remember it.
67             lastAnimalFound = animal;
68     }
```

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

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.

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.

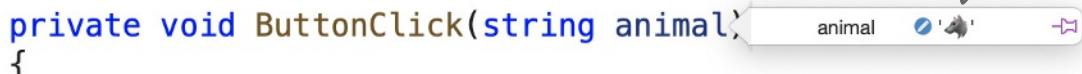
3

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:

```
62     private void ButtonClick(string animal)
63     {
64         if (lastAnimalFound == string.Empty)
65     }
```

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:



A screenshot of Visual Studio showing a tooltip for the `animal` variable. The tooltip displays the emoji `\ud83d\udcbb` (a lion) with the text "animal". A callout arrow points from the text "Hover over 'animal' to see the emoji that you clicked." to the tooltip.

```
private void ButtonClick(string 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:

```
64     if (lastAnimalFound == string.Empty)
65     {
66         //First selection of the pair. Remember it.
67         lastAnimalFound = animal;
68     }
```

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



A screenshot of Visual Studio showing a tooltip for the `lastAnimalFound` variable. The tooltip displays the emoji `\ud83d\udcbb` (a lion) with the text "lastAnimalFound". A callout arrow points from the text "The statement that you stepped over set the value of lastAnimalFound so it matches animal." to the tooltip.

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

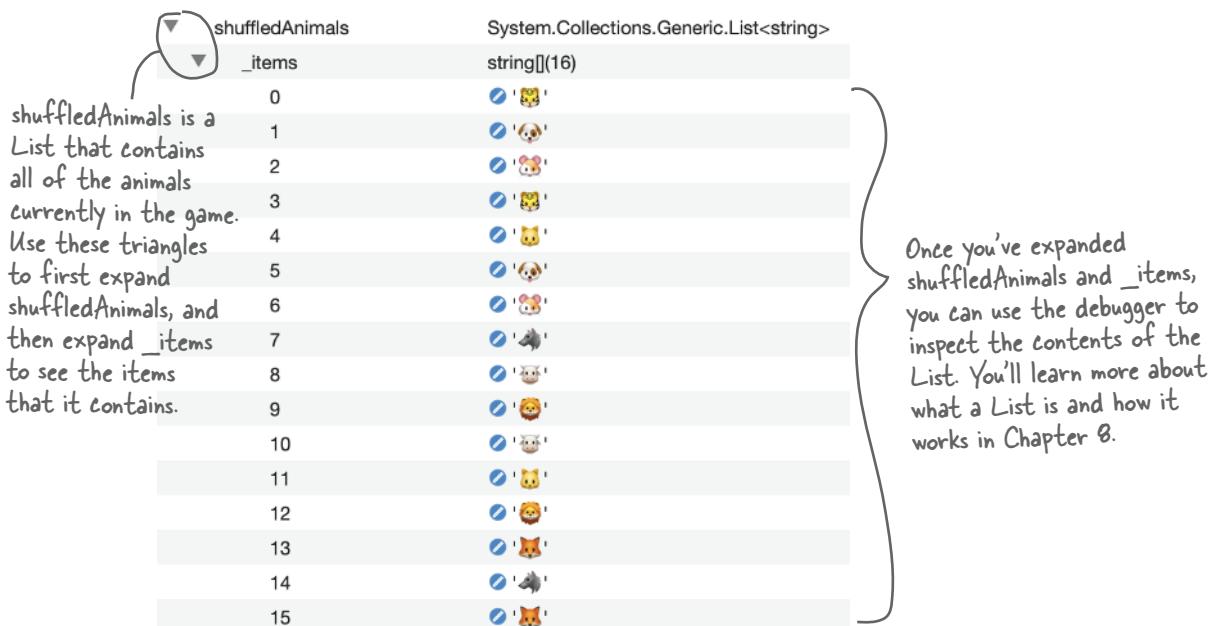
Find the button with the matching emoji and **click it**. The IDE will trigger the breakpoint and pause the app again. Press **Step Over**—it will skip the first block and jump to the second:

```
69     else if (lastAnimalFound == animal)
70     {
71         //Match found! Reset for next pair.
72         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**:

```
74
75     //Replace found animals with empty string to hide them
76     shuffledAnimals = shuffledAnimals
77         .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:



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...

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 `ButtonClick` 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}";

```

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`

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;`

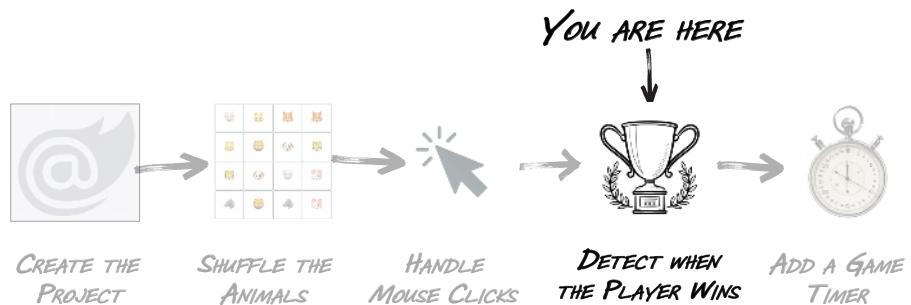
Errors				
	Line ^	Description	File	Project
✖	90	The name 'lastDescription' does not exist in the current context (CS0103)	Index.razor	Blazor...tchGame Pages/Index.razor
✖	87	The name 'lastDescription' does not exist in the current context (CS0103)	Index.razor	Blazor...tchGame Pages/Index.razor

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.

find all the animals and reset the game



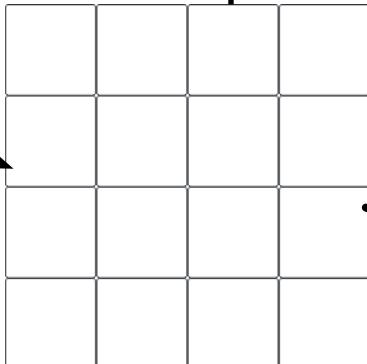
Add code to reset the game when the player wins

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.



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?



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;
```

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

```
matchesFound = 0;
```

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

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

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

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

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

Which of the four blocks of code above goes in this box?

```
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;
```

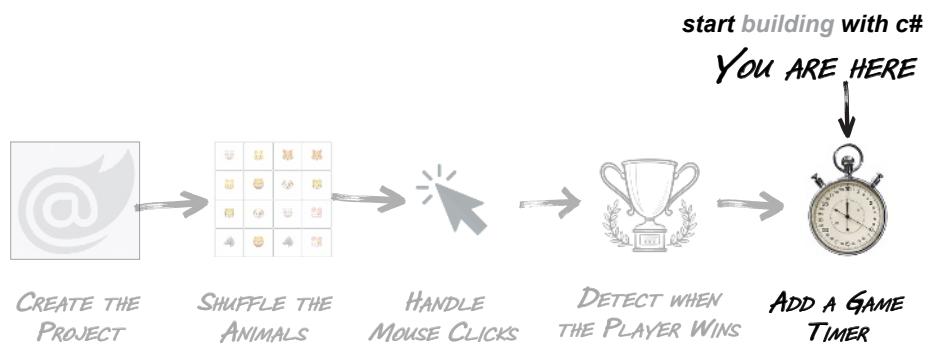
```
        // Replace found animals with empty string to hide them
        shuffledAnimals = shuffledAnimals
            .Select(a => a.Replace(animal, string.Empty))
            .ToList();
```

```
        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.



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

- 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`

 Add this!

- 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.

you've done a great job

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.



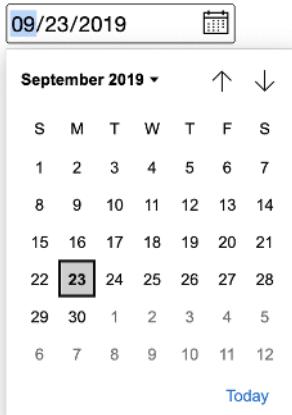
- ★ **Sliders** are used exclusively to choose a number. Phone numbers are just numbers, too, so *technically* you could use a slider to choose a phone number. Do you think that's a good choice?



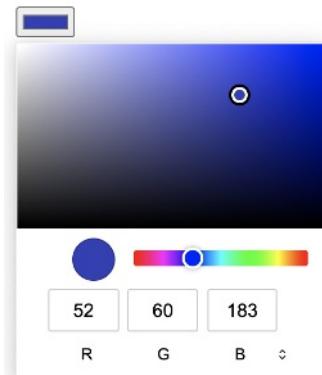
- ★ **Radio buttons** let you restrict the user's choice. They often look like circles with dots in them, but you can style them to look like regular buttons too.

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

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



- ★ **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.

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>
```

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

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.

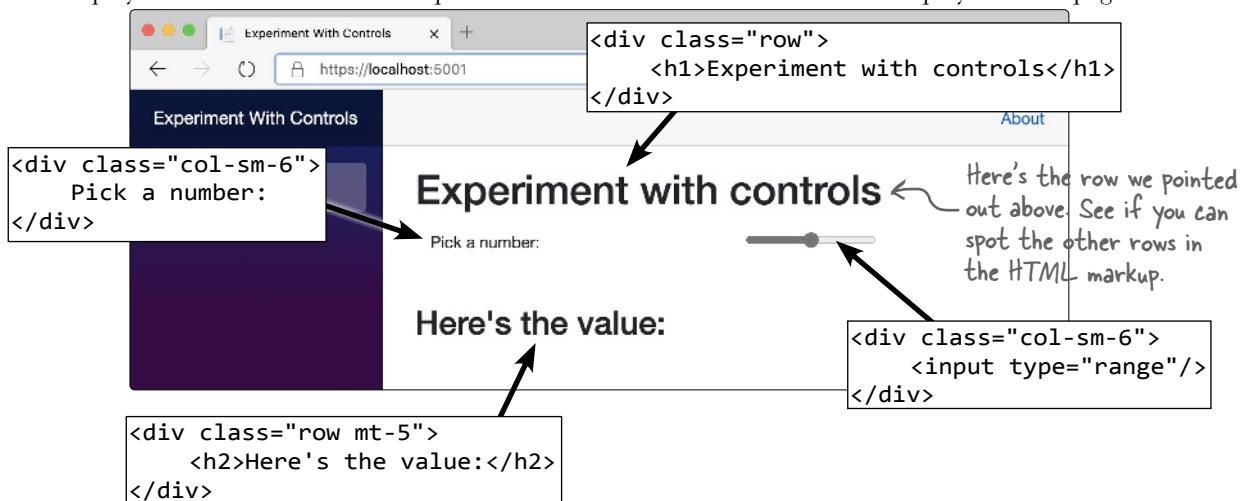
This is an `input` tag. It has a `type` attribute that determines what kind of input control appears on the page. When you set the `type` to `range`, it displays a slider:

`<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.



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(ChangeEventArg e) ←
    {
        DisplayValue = e.Value.ToString();
    }
}
```

The change event handler updates `DisplayValue` any time it's called with a value.

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.

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

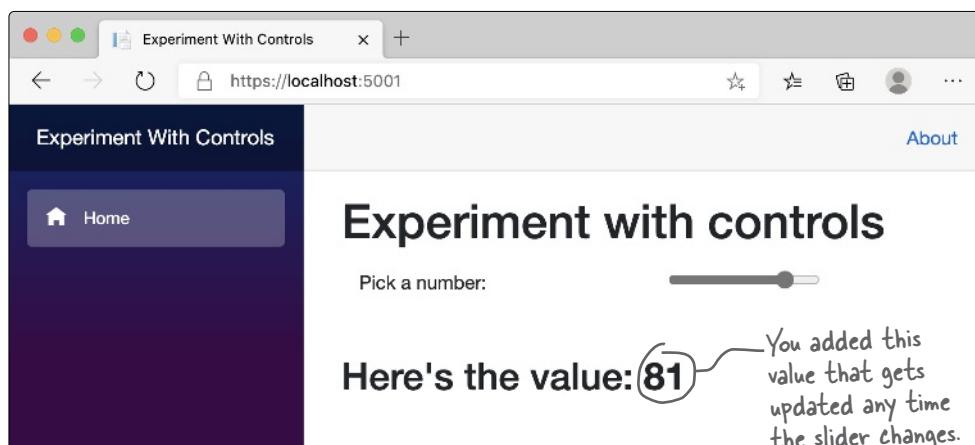
Add an `@onchange` attribute to your range control:

```
@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.



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:

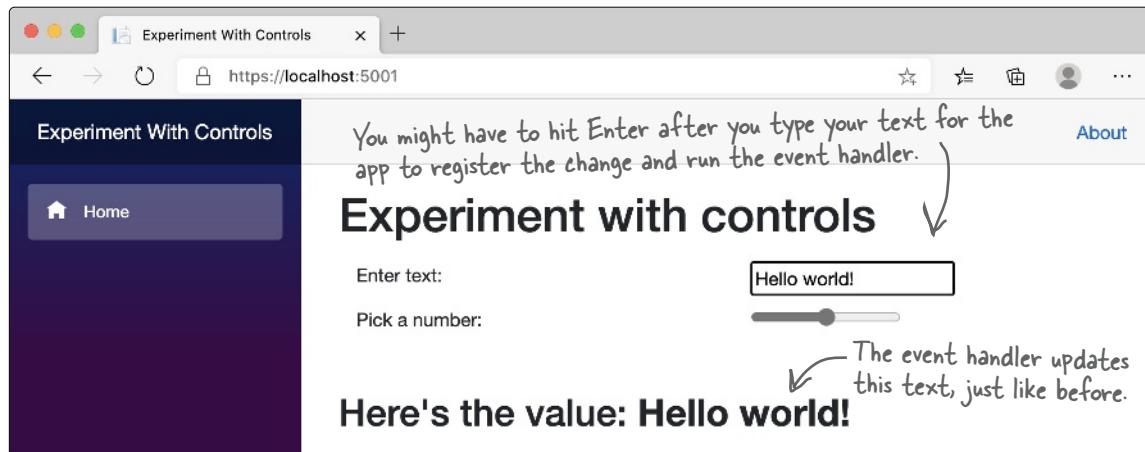
```
<div class="container">
    <div class="row">
        <h1>Experiment with controls</h1>
    </div>
    <div class="row mt-2">
        <div class="col-sm-6">
            Enter text:
        </div>
        <div class="col-sm-6">
            <input type="text" placeholder="Enter text"
             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

Enter 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.



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();
    }
}
```

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

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

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">
        <!-- This box contains the code for the six buttons -->
        <!-- This arrow points to the opening brace of the loop -->
        {
            string valueToDisplay = $"Button #{buttonNumber}";
            <button type="button" class="btn btn-secondary"
                @onclick="() => ButtonClick(valueToDisplay)">
                @buttonNumber
            </button>
        }
    </div>
</div>
<div class="row mt-5">
    <h2>
        Here's the value: <strong>@DisplayValue</strong>
    </h2>
</div>
```

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

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



Exercise Solution

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

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

Pick a number:

Click a button:

1 2 3 4 5 6

Here's the value: **Button #2**

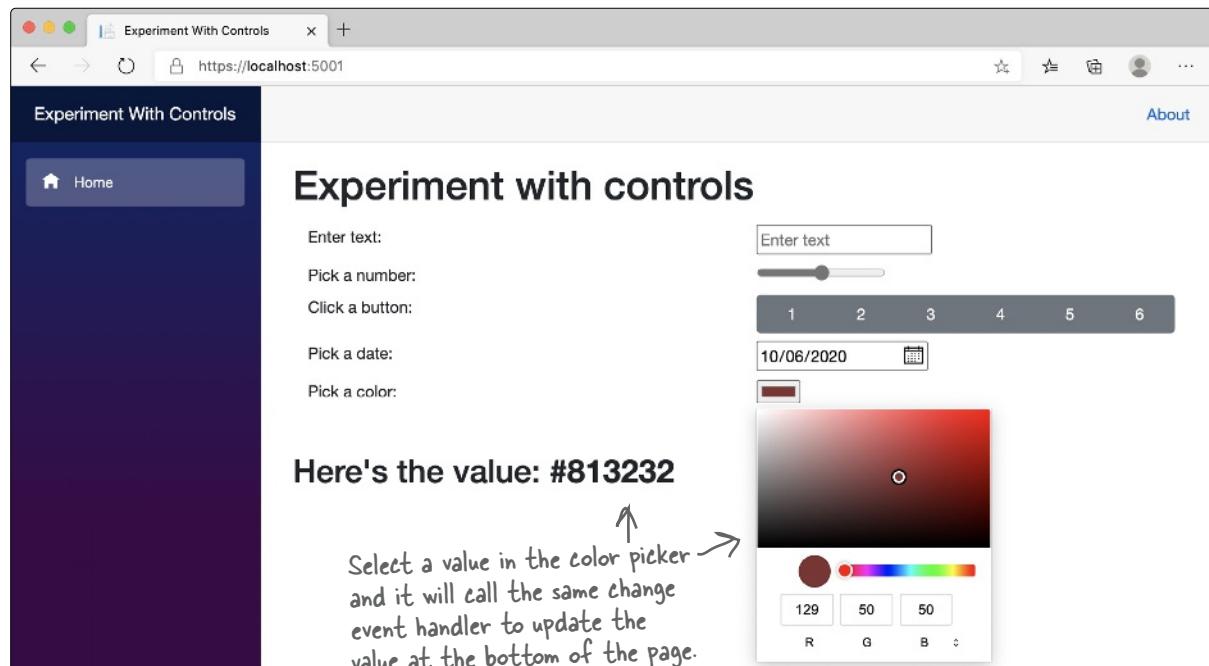
Add color and date pickers to your app

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.



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!

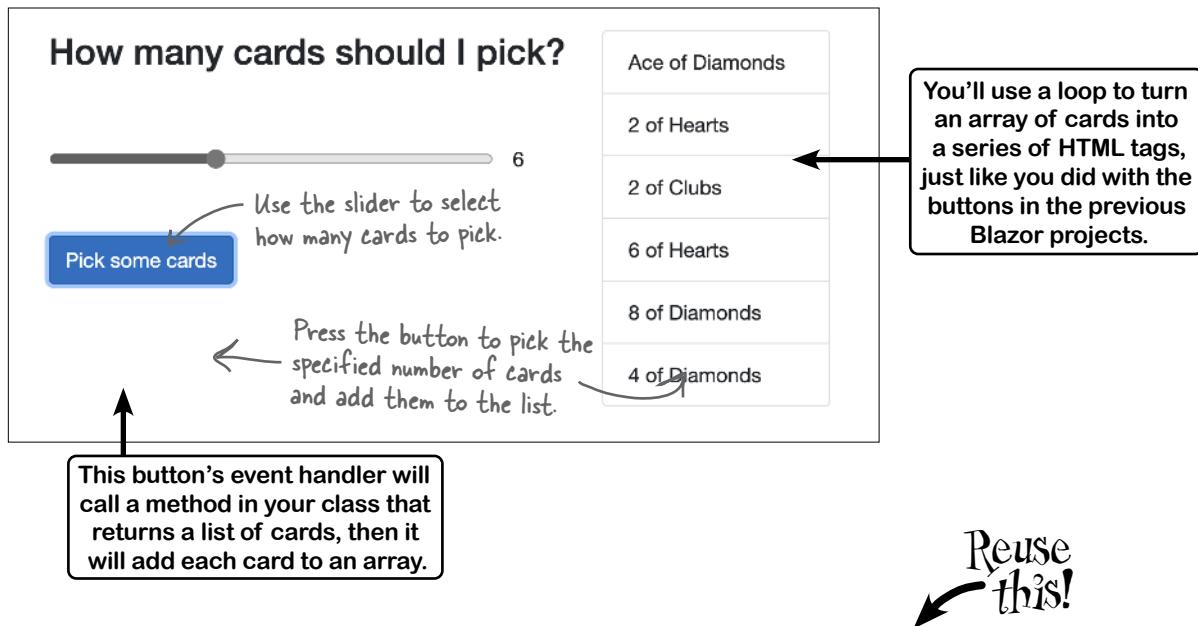
from Chapter 3 objects...get oriented!

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:



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.

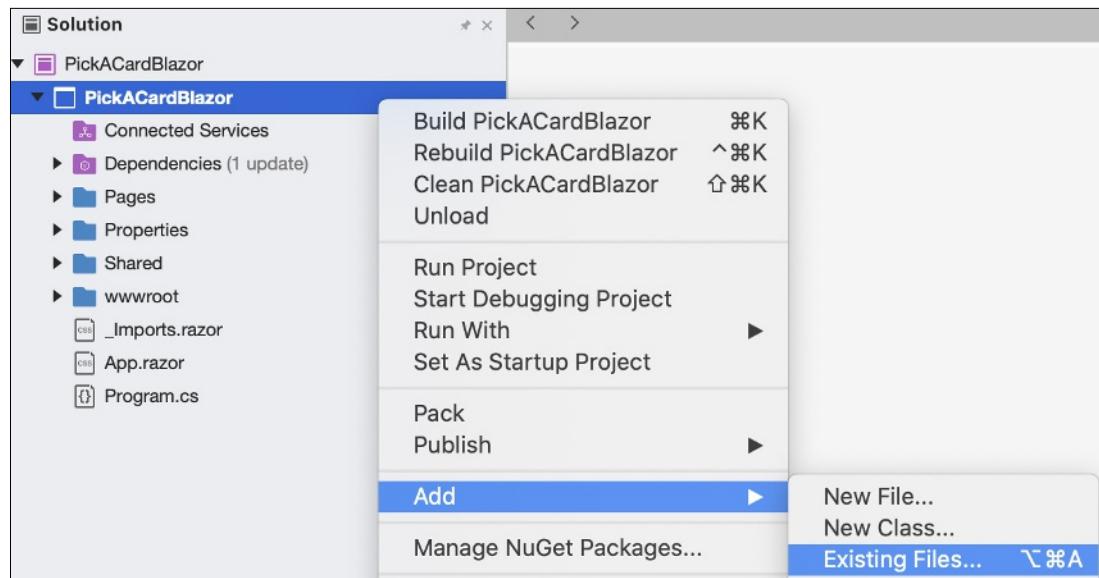
① Create a new Blazor WebAssembly App project called PickACardBlazor.

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:

```
► M: 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.

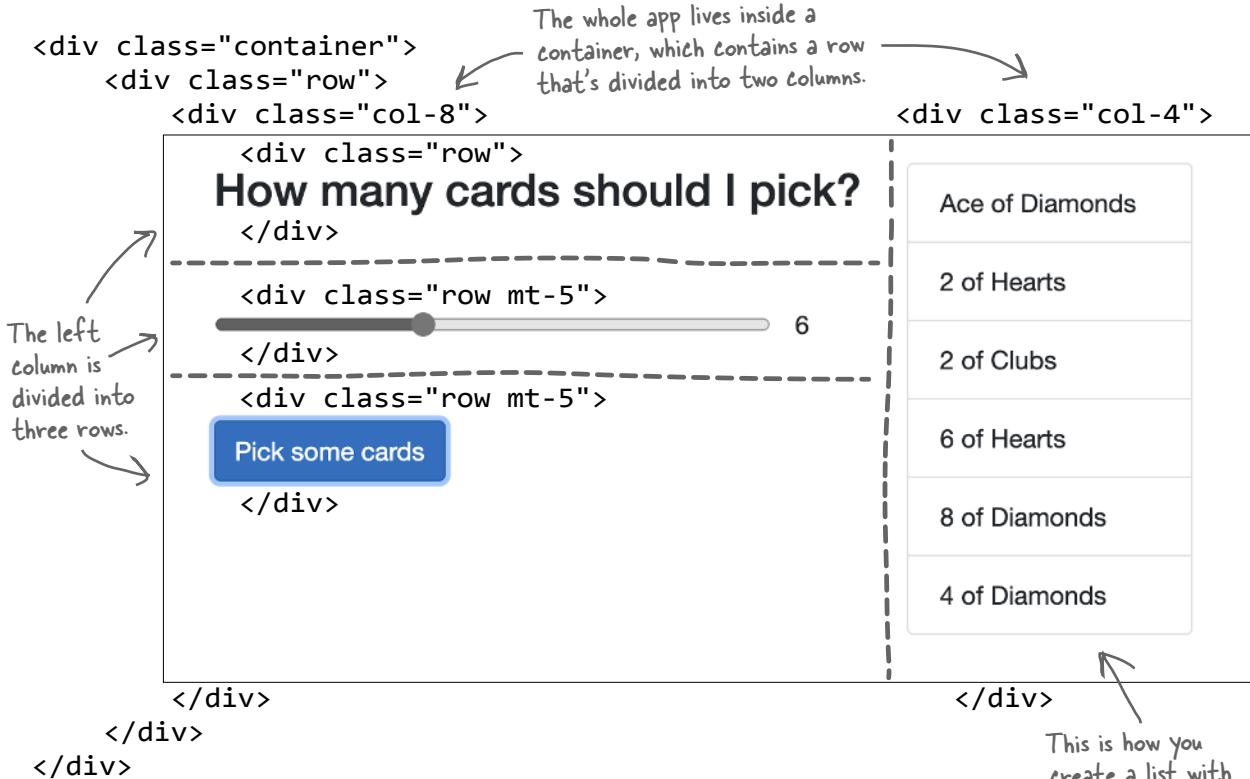
Now your *CardPicker* class should be in the *PickACardBlazor* namespace:

```
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:



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>
```

The code is annotated with curly braces and arrows to show the structure. Braces group the `ul` element and its contents. Another brace groups the entire `foreach` loop. Arrows point from the brace on the `foreach` loop to the opening brace of the `ul`, and from the brace on the `ul` to the closing brace of the `foreach` loop.

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

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

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> How many cards should I pick?
</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> 6
</div>
```

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.



We've given 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.



Exercise Solution

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

```
@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.

numberOfCards and pickedCards are special kinds of variables called **fields**. You'll learn about them later in Chapter 3.

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.

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...

sloppy joe sez: "that roast beef's not old... it's vintage"

from Chapter 4 types and references

At the end of Chapter 4 there's a Windows project. We'll build a Blazor version of it.

This is the Blazor version of the Windows desktop project in Chapter 4.

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

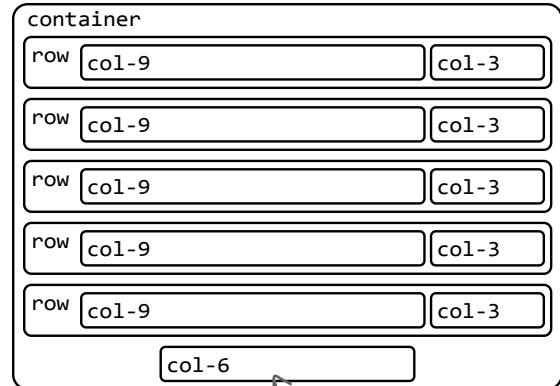
Add the page layout to your Index.razor file.

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>

@code {
    MenuItem[] menuItems = new MenuItem[5];
    string guacamolePrice;
}

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

```

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

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.

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.

Make sure you call the `Generate` method; otherwise the `MenuItem`'s fields will be empty and your page will be mostly blank.



How it works...

I EAT ALL MY MEALS AT SLOPPY JOE'S!

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.

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 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.

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.

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?

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	

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

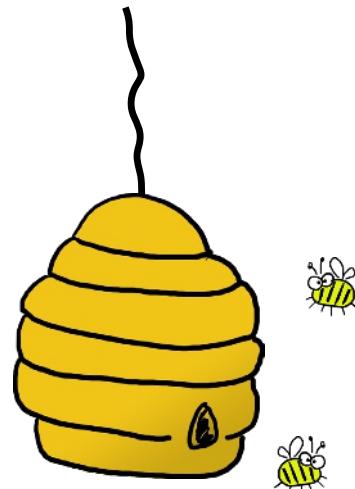
You've probably noticed that the projects in Chapters 5 and 6 are longer and more involved than the ones in the previous chapters. We want to give you the best learning experience possible—but doing that takes more pages than we can fit in this appendix! That's why we've made Chapters 5 and 6 of the 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

Symbols

+ (plus) addition operator 169
2D scene editor, Unity as a 88
3D environments, Unity Hub scenes as 92
3D scene editor, Unity as a 88
3D vectors (Unity) 222
* (asterisk) operator 135, 169
@ character 396
{ } (curly brackets) 54, 67, 173, 212, 337, 407, 435, 558
\\" (double backslash), escaping backslash in strings 547
.NET Core 2, 54
.NET desktop development 2
&& operator 62
/ (slash) division operator 169
[] (square brackets) 524
|| (OR operator) 62

A

ability score, in role-playing games 156, 174, 176–183
abstract classes
 about 334–341
 exercises on 339
 usefulness 335–336

allocate, defined 545
allocated resources 545
angle brackets 413, 420, 422, 438
animal inheritance program 280–286
anonymous, defined 492
anonymous types 492, 496
API (application programming interface) 193
Appliance project 384, 385
apps
 adding TextBlock controls to 75–76
 statements as building blocks for 55
ArgumentException 650
arguments
 compatibility with parameter types 172
 defined 172
 specifying 260
arithmetic operators
 automatic casting with 171
 defined 169
Array instance 318
array of strings 106
arrays
 about 106, 200–201
 containing reference variables 201
 difficulty in working with 412
 exercises on 202, 204
 finding length 201
 using to create deck of cards 411
 using [] to return object from 524
 versus lists 414–416
 zero-based 200
ArrowDamage class 274
as keyword 381, 385, 403, 613
ASP.NET Core 6
Assert.AreEqual method 503. *See also* unit testing
assignment, of values to variables 57
assignment operator (=) 63

assistive technology 563

attribute 503

automatic properties 373

Awake method 583

B

backing fields 254, 256

base classes

about 278, 280

building Animal base class for zoo simulator 281–282

colon 290

constructors in 308

exercises on 295–296

extending 285

subclasses accessing with base keyword 306

upcasting 384

using subclasses instead 286

with constructors 307

base keyword 306

Beehive Management System project

class model 317

converting from turn-based to real-time 330

data binding and 400–401

exercises for 320–326

feedback in 328

interfaces 366–367, 388

Queen class 318

RoboBee class 370

user interface for 319

Visual Studio for Mac project 724

bees 316–322

behavior of objects 133

Billiard Ball game 344–354, 453–465

binary data 184, 561, 564, 569

reading 570

writing 569

binary (executable) 32, 54, 86, 193, 534, 697

binary numbers 165, 171

binary operators 183

BinaryReader 570

BinaryWriter 569

black box 247

blocks (of code) 54

bool type 57, 158, 166

boxed objects and structs 604, 613, 614

Break All button 43

breakpoints 43, 218

bubble up 641

by keyword 499

byte arrays 541, 568

byte type 159, 165

C

C#

adding for controls 84–85

adding to update TextBlock 78

application code 12

benefits of learning 2

combining with XAML 11

exercise on 29

scripts 214, 215

specificity of 21

Canvas (Unity) 458

Captain Amazing 468, 534, 546, 587, 613, 621

CardPicker class 106, 110, 120, 714

casting

about 168–170

automatic 171

calling methods using 357

converting random double to other types 207

decimal value to int type 168

exercises on 169

too-large value, automatic adjustment in C# 169

versus conversion 185

wrapping numbers 169

catch-all exceptions 637

catch blocks

about 634, 650

following in debugger 635–636

letting your program keep running 648

with no specified exceptions 638

characters 24, 161, 166, 537, 541, 547, 572–574

Unicode 161, 561, 563–571, 576

character sheets, in role-playing games 157

- char type 161, 166, 566
- Checked events 230
- child 280
- CICS (candy isolation cooling system) vent procedure 139
- CIL (Common Intermediate Language) 193, 212
- C# Interactive window 154
- class diagrams 107, 133, 141, 229
- classes
 - about 50, 51–53
 - abstract (see abstract classes)
 - adding fields to 221
 - collection 413
 - concrete 334
 - debugger and 313
 - designing 145, 146
 - designing separation of concerns 310
 - encapsulation 240–245, 249–250
 - exercises on 143, 151–152
 - inheritance and 278
 - looking for common 283
 - names for 140–141
 - namespaces 52
 - never instantiated 332
 - reusing 151–152
 - similarities between 145
 - static 134
 - using to build objects 127
 - versus structs 614
 - why some should never be instantiated 336
- class hierarchy 279, 284, 380, 382
- class models 279, 287–288, 309–310
- clauses in LINQ queries 475
- Clown class exercise 135, 136
- clowns 135–137
 - clown car 395
 - Fingers the clown 184, 389–390, 510
 - terrifying 397
- CLR (Common Language Runtime) 189, 193–194, 205, 212, 237, 250, 258, 263, 272, 590, 592–597, 603, 605, 614, 628, 631
- code
 - advice for code exercises 145
 - automatically generated by IDE 11
 - avoiding duplication 281
 - compact 138
 - copying 125
 - refactoring 142
 - repeating 277
 - replacing comments with 149–150
 - reusing 104
 - similar 278
 - code-behind 21, 124, 232
 - code blocks 54
 - code comments 139
 - code points 564
 - code snippets 67, 261, 262
 - collection initializers 208, 426–427
 - collections
 - about 412
 - compared with enums 425
 - compared with sequences 524
 - dictionaries 442–455
 - generic 425
 - lists 413–430
 - performing calculations on 486
 - querying with LINQ 470
 - using join to combine into one query 491
 - colon, using to inherit from base class 290
 - colors, selecting color scheme in Visual Studio 9
 - columns (XAML grid) 16, 17
 - ComboBoxes 72, 81, 85, 319, 325–326
 - command line 575, 624
 - command-line arguments 575
 - comments 33, 56, 149–150
 - commit, Git 30–31, 47–48, 98, 501, 714
 - CompareTo() method 429. *See also* IComparable interface
 - compiler 178, 193
 - components 101
 - compound assignment operators 183
 - concatenating strings 170
 - concatenation operator (+) 171
 - Concat method 474
 - concrete classes 334
 - conditional operator 513

conditionals 68
conditional tests 64, 69
console applications 4, 50, 300
`Console.WriteLine` 237
constants 228
constructors
 about 24, 234, 237, 258–260, 263, 266
 exceptions in 641
 in base classes 307
 initializing properties using 259
container tags 11
controls
 updating automatically with data binding 399
 user interfaces and 71
conversion, compared with casting 185
`Convert` class 165, 207
`Count` method (LINQ) 474
cross-platform game engine, Unity as a 88
`CryptoStream` 538
cylinders (Unity) 220

D

damage calculator app 229–237
`Debug.DrawRay` 222
debugger
 anatomy of 43
 Bullet Points 650
 catch blocks 635–636, 638
 classes and 313
 exercises for 43
 finally block 636
 following try/catch flow 635
 importance of 61
 overriding and 298
 `Random.value` and 347
 `time.deltaTime` and 219
 troubleshooting exceptions with 42–45
 using to see changes in variables 61
 yield return statement 523
debugging mode 26, 182, 218, 255
`Debug.WriteLine` 235, 236, 237

decimal type
 about 160, 166
 attempting to assign decimal value to int variable 168
 precision of 184
declaration 254
default implementations 394, 396
default keyword 275
deferred evaluation 487
deserializing 552
design
 intuitive classes 145
 separation of concerns 310
diagnostic information, printing 235
dictionaries
 about 442–444
 building programs that use 444
 functionality of 443
 keys 442
 using [] to return object from 524
directories
 creating new 542
 getting list of files 542
`Dispose()` method. *See also* `IDisposable` interface
 about 545
 calling 645, 647
 finalizers 595, 596
`DivideByZeroException` 627, 629
double type 158, 160, 166
do/while loops 64
downcasting
 about 385
 interfaces and 386
 moving up/down class hierarchy using 382
 using as keyword 403
downloading Unity Hub 89
Dynamics Game design...and beyond element 314

E

edge cases 506
editors 53
embedded system 139

emergence 314
 emoji 7, 24–25, 559, 563–567, 564, 678–679
 encapsulation
 about 227, 237, 614
 as principle of OOP 404
 benefits for classes 247
 better, using protected modifier 392
 compared with security 246
 data safety and 252
 defined 239
 exercises on 267–270
 ideas for 248
 improving `SwordDamage` class using 251
 properties and 254
 using to control access to class methods, fields, or properties 240–245
 using to prevent null properties 610
 well encapsulated versus poorly encapsulated classes 249
 encodings 532, 541, 566
`Encoding.UTF8.GetString` 547
 end tags 11
 Enumerable class 521
 enumerable sequences 522
 enumeration 406–407
 enums
 about 407–411
 big numbers 408
 building class that holds playing card 409, 410
 compared with collections 425
 exercises on 409–410, 437–438
 representing numbers with names 408
 equality operator 62, 63
 equals sign operator 57, 63
 Error List window 9, 32
 errors
 avoiding file system errors with using statements 546
 `DivideByZero` 627
 invalid arguments 172
 escape sequences 161, 396, 544, 547, 567, 576
 event handlers
 adding 33, 79–80
 calling 38
 defined 34
 exception handling 42–45, 628, 629, 631, 650
 about 623–662
 Bullet Points 650
 catch block 634
 catching specific exception types 638
 `DivideByZeroException` 627, 629
 dividing any number by zero 627
 Exception object generated when program throws exception 628
 exceptions in constructors 641
 filters and 646
 finally block 636
 `FormatException` 629
 handled versus unhandled exceptions 638
 handling, not burying 648
 handling versus fixing 649
 `IndexOutOfRangeException` 629
 `NullReferenceException` 627
 `OverFlowException` 629
 program stopping with exceptions 638
 try block 634
 using exceptions to find bugs 631
 executable binaries. *See* binary (executable)
 exercises
 ability score 179–180
 abstract classes 339
 Beehive Management System project 320–326
 Billiard Ball game 351
 class diagrams 229
 code blocks 293–294
 comparing 433–434
 creating a new WPF project 73–74
 encapsulation 267–270
 enumerable classes 526–527
 enums 409–410
 extending base classes 295–296
 filling in Main method 111–112
 for cards 452, 549–550
 garbage-collection 194–196
 interfaces 371–372, 389–390
 `JsonSerializer` 557
 LINQ 498–499
 private keyword 243–244
 queues and stacks 449–450
 radio buttons 80–82
 replacing comments with code 149–150

reusing classes 151–152
Sharpen your pencil
ability score 177
arrays 202, 204
broken code 625–627
building paper prototypes 117
casting 169
C# code 29
classes 143
class models 287–288, 309–310
Clown class 135, 136
code errors 265–266
conditionals 68
debugging 43
determining game improvements 48
enums 437
File and Directory classes 543–544
interfaces 362–363
is keyword 381, 383
keywords 163–164
lambda expressions 511–512, 518, 519–520
LINQ 473–474, 493–494
lists 415–416
loops 65, 68
object references 191–192, 584–586
Random class 132
statements 59
TextBlock_MouseDown method 36–37
types 167
Unicode 569
values 161, 162
Visual Studio IDE 9–10
writing to the console 153
switch statement 275–276
test 528
TextBlocks 19–20
transform 463–465
unit testing 519
Who Does What? 27–28
writing code 311–312, 655–656
expression-bodied member 510
extend 280
Extensible Application Markup Language. *See XAML*
extension methods 617, 618

F

feedback loop 328
fields
about 110
adding to classes 221
backing fields, set by properties 254
defined 37
instances and 133–134
masking 266
private 239–244, 253, 395
public 248, 253
versus properties 392
with no access 242
XML documentation for 140
FIFO (First In, First Out), queues 446
File class
about 542
AppendAllText() method 542
Close() method 571
CreateDirectory() method 542
Create() method 542
Delete() method 542
Exists() method 542
GetLastAccessTime() method 542
Get LastWriteTime() method 542
OpenRead() method 542
OpenWrite() method 542
ReadAllBytes() method 566, 568, 571
ReadAllLines() method 571
ReadAllText() method 571
static methods 571
WriteAllBytes() method 566, 568, 571
WriteAllLines() method 571
WriteAllText() method 571
FileInfo class 542
files
appending text to 542
finding out if exists 542
getting information about 542
opening in Solution Explorer 14
reading from or writing to 542
(see also streams)
FileStreams
about 531

- created and managed by StreamWriter 533, 541
 - reading and writing bytes to file 532
 - versus StreamReader and StreamWriter 571
 - finalizers
 - about 592
 - depending on references being valid 595
 - Dispose() method 595, 596
 - fields and methods 597
 - when they run 593
 - finally block 636, 647
 - fired events 78
 - floating-point types 160, 166, 172, 173, 184–185
 - float type 158, 160, 166, 169
 - foreach loops
 - about 44
 - accessing all members in stack of queue 448
 - from clause in LINQ queries compared to 484
 - lists 422
 - updating 436
 - using IEnumerable<T> 439
 - for loops 64
 - FormatException 629
 - format strings 541
 - {0} and {1} 541
 - frame rate 217
 - frames 217, 218
 - Func parameter 514
 - functions 404
- ## G
- Game design... and beyond elements
 - Accessibility 562
 - Aesthetics 264
 - Dynamics 314
 - Mechanics 70
 - Prototypes 115
 - Tabletop Games 206
 - Testing 501
 - What Is a Game? 7
 - game modes 455
 - GameObjects
 - about 93
- ## H
- adding behavior to 214
 - adding C# script to 215
 - adding material to 96–97
 - anchored 459
 - components of 95
 - creating 580
 - deleting 353
 - moving using Move Gizmo 94
 - garbage collection 188–189, 193–194, 197, 203, 205, 212, 369, 414, 590–597, 600, 603, 614
 - GC.Collect() method 593, 597
 - generic collections 420–423, 425, 445–447, 451
 - generic data types 425
 - generic lists 420
 - get accessor (getter) 254, 257, 373
 - GetReviews method 505
 - GetType method 246
 - Git 30, 47
 - GitHub 30, 98, 688
 - Go to Declaration 421
 - Go to Definition 421
 - greater than operator (>) 62
 - grid (XAML)
 - adding rows and columns to 16
 - adding sliders to 83
 - adding TextBlock controls 18
 - sizing rows and columns 17
 - troubleshooting 19
 - using to lay out main window 121
 - group by clause (LINQ) 490, 499
 - group clause (LINQ) 499
 - GZipStream object 531

using different reference to call hidden methods 303
using new keyword 302

hierarchy

about 279, 351
creating class hierarchy 284
defined

Hierarchy window (Unity) 216, 353

hit count 218

HorizontalAlignment property 18

HTML, compared with XAML 32

HUD (head-up display) 116

I

IClown interface 360, 389–390

IComparable interface 429

IComparer interface

complex comparisons 432
creating instance 431
multiple classes 431
SortBy field 432
sorting lists using 430

IDisposable interface

about 545, 647
Dispose() as alternative to finalizers 595
try/finally and 645

IEnumerable interface

about 521
foreach loops using 439
LINQ and 472
upcasting entire list with 440

IEnumerator interface 525

if/else statements 63, 294

if statements 63

Implement interface (Visual Studio) 510

index 200–201, 416, 524

IndexOutOfRangeException 628, 629

infinite loops 69

inheritance

about 277–354
as principle of OOP 404
building class models 279

classes and 278
creating class hierarchy 284
designing zoo simulator 280
each subclass extending its base class 285
interface 388
looking for classes with much in common 283
multiple 341
subclass accessing base class using base keyword 306
subclasses 291–292
terminology 280
using colon to inherit from a base class 290
using override and virtual keywords to inherit behavior 304
using subclass in place of base class 286
using to avoid duplicate code in subclasses 281

inherit, defined 279

initialization 148

initializer 64

Inspector window (Unity) 93, 95, 99, 101, 215–216, 220, 352, 459–460, 466

installing GitHub for Unity 98

instances

defined 128
fields and 133–134
requirement for, versus static methods 134

Instantiate method (Unity) 349

instantiation 133, 366

IntelliSense window 132

interfaces

abstract classes and 364
defining methods and properties using 358
defining two 379
downcasting and 386
exercises on 362–363, 371–372, 389–390
inheriting from other interfaces 388
naming 360
new keyword 366
object references versus interface references 392
public 373
references 366–367, 392
static members in 395
upcasting and 384, 386
void method 360
why use 387, 392

int type
 about 57, 158, 165, 167
 adding to float type, conversion with + operator 169
 attempting to assign decimal value to int variable 168
 no automatic conversion to string 172
 invalid arguments error 172
InvokeRepeating (Unity) 349
IScaryClown interface 397
 is keyword 84, 375, 379, 380, 381, 383
 isometric view (Unity) 653
 iterator 64

J

join clause (LINQ) 491, 496, 499
JSON serialization 556, 558, 559
JsonSerializer 560

K

keywords 163–164

L

labels for objects (see reference variables)
lambda expressions 508–514
Last method 474
Length property, arrays 201
 less than operator (<) 62
LIFO (Last In, First Out), stacks 447
LINQ (Language Integrated Query)
 about 467–528, 618
 difference from most of C# syntax 485
 exercises on 473–474, 493–494, 498–499
First method 474
 group queries 488–490
IEnumerable interface and 472
 join queries 491, 496
 lambda expressions and 514
 method chaining and 471
 methods 470–473, 516
 modifying items 486
 objects and 477

performing calculations on collections 486
 queries 475, 515
 querying collections with 470
 query syntax 475
 using join to combine two collections into one query 491
 var keyword and 494

List class 413–430
 Sort() method 428

lists

exercises on 415–416
 flexibility of 414
 generic 420
 sorting 428
 using [] to return object from 524

literals 162

logical operators 62

long type 159, 165

loops

about 64
 exercises on 65, 68
foreach. *See* foreach loops
 infinite 69
 writing using code snippets 67

M

macOS line endings 551
Mac requirements 2
Main method 50, 111–112, 255, 575
MainWindow.xaml 21
Margin property (XAML) 18
Materials folder 345
Math.Ceiling method 274
Max method (LINQ) 474
MDA (Mechanics-Dynamics-Aesthetics) framework 329
 mechanics 71, 314
Mechanics Game design... and beyond element 70
 memory
 about 136
 allocating 167
 stack versus heap 603–605
 types and 166

- MemoryStream 531, 547
Message property, Exception object 629
method chaining 471
methods
 about 51
 abstract 334
 accessing private fields with public methods 242
 accessing with is keyword 376–377
 body 59, 334, 337, 342, 510
 calling most specific 285
 calling on classes in same namespace 52
 defining with interfaces 358
 extension (see extension methods)
 generating to set up game 22–25
 generating to work with variables 58
 get and set accessors versus 257
 hidden, using different references to call 303
 hiding versus overriding 302
 LINQ 516
 names for 140–141
 object 127
 optional parameters, using to set default values 608
 overloaded 536
 overriding 282
 passing arguments by reference 607
 public 248
 returning more than one value with out parameters 606
 risk in calling 633
 signature 263
 this keyword with 198
 using override and virtual keywords 304
 XML documentation for 140
Microsoft Visual Studio Debug Console window 5
Min method (LINQ) 474
mod function 185
mouse clicks 33, 34
Move Gizmo 94
multiple inheritance 341
- ## N
- \n (line feed character) 161, 396
names, for variables 56
- namespace keyword 50
namespaces 51, 52
navigation system (Unity) 581
NavMesh 581, 655, 657, 658
.NET Framework
 garbage collection 593
 namespaces 51, 164
 sealed classes 617
 streams, reading and writing data 530
 structs 599
NetworkStreams 531
new keyword
 about 126
 interfaces 366
 using to create anonymous types 492
 using when hiding methods 302
new statements 127, 200
NextDouble method 207
non-nullable reference types 610
non-numeric types 166
NotImplementedException 642
null 77, 161–162, 203, 212, 235, 381, 393, 412, 483, 609–612, 695
Nullable<T> 612
null-coalescing operator (??) 611
null reference 412, 609–611, 638
NullReferenceException (NRE) 77, 609, 627, 650
numbers, representing with names using enums 408
- ## O
- object initializers 148, 263
object references 392, 584–586
objects
 about 103–154, 590
 accessing fields inside object 239
 accidentally misusing 238
 behavior of 133
 boxed 604
 building from classes 127
 checking type of 375
 encapsulation (see encapsulation)

- exercises on 191–192
- finalizers (see finalizers)
- garbage collection 188–189
- LINQ and 477
 - misusing 238
 - null keyword 203
 - object type 161
 - reading entire with serialization 556
 - references 369
 - reference variables (see reference variables)
 - storage in heap memory 136
 - talking to other objects 198, 205
 - upcasting 384
 - using to program Enemy class (example) 126
 - value types versus 600
 - versus structs 601, 613
- on...equals clause (LINQ) 491, 496, 499
- OnMouseDown method 659
- OOP (object-oriented programming) 404
- OperatorExamples method 58
- operators 59, 62
 - optional parameters, using to set default values 608
- OrderBy method 515
- out parameters 606
- Output window 235, 236
- OverflowException 629
- overloaded constructors 537
- overloaded methods 536
- override keyword 300–301
 - overriding methods
 - about 305
 - hiding versus 302
 - override keyword 292, 299, 304
- P**
- PaintBallGun class 253–260
- paper prototype 115, 117, 354
- parameters
 - about 105
 - defined 172
 - masking fields 259, 266
- parent 280
- partial keyword 52
- physics engine (Unity) 353
- Physics.Raycast method (Unity) 585
- PickRandomCards app
 - CardPicker class 110
 - creating 106–107
 - creating a WPF version of 118
 - laying out desktop window for 122–124
 - PickSomeCards method 108–109
 - reusing CardPicker class in 120
 - StackPanel control 119
 - using grids and StackPanel controls to lay out main window 121
- placeholder objects (Unity) 578
- plane (Unity) 578
- platforms, in scenes (Unity) 653–654
- polymorphism 403
- prefab 348
- Preview changes window 22
- primitive objects 578
- private fields 239–244, 253, 395
- private keyword 245–246, 246–249, 248
- private setter 257
- programs
 - anatomy of C# program 51
 - loops in 64
 - operators in 59
 - using debugger to see variables change 60, 61
- Project window 215
- properties 254–257, 263, 272
 - auto-implemented 256
 - creating with prop snippet 256
 - defining with interfaces 358
 - encapsulation and 254
 - in interfaces 373
 - initializing using constructors 259
 - statements in 257
 - versus fields 392
- Properties window
 - showing changes in 15
 - switching to Events view 84
 - toggling 34
- protected keyword 297

Prototype Game design... and beyond element 115
prototypes 113–115, 116–117
pseudo-random number generator 211
public fields 248, 253
public interfaces 373
public methods 242, 248
publishing to GitHub 31, 47, 98, 688

Q

queries. *See LINQ (Language Integrated Query)*
queues

about 445
converting to lists 448
enqueueing and dequeuing 446
exercises on 449–450
FIFO (First In, First Out) 446
foreach loop 448
Quick Actions icon 22

R

RadioButton controls 72
radio buttons 80
Random class 132–133, 207, 409
randomizing results 208–209
random number generators 206
Random.value (Unity) 347
Raspberry PI 138
raycasting 577–586
readonly keyword 391
read-only property 257
real-time game 330
refactoring code 142
references
 exercises on 191–192
 interface 366–367
 object 369
 object versus interface 392
 passing by reference using ref modifier 607
 versus values 600

reference variables
 about 186–188
 arrays of 201
 garbage collection 188–189
 how they work 205
 interface type, pointing to object implementing interface 403
multiple references to single object
 accessing different methods and properties 386
 side effects of 190
 unintentional changes 197
objects talking to other objects 198
setting equal to instance of different class 403
substituting subclass reference in place of base class 294
ref keyword 607
relational operators 62
remainders 185
rendered 93
resource management game 327
Restart button 43
return statement 105, 109, 124
return type 105, 124
Reverse method (Unity) 474
risky code (LINQ) 633–650
RoboBee class 370
robust design 506, 633
Rotate Tool (Unity) 99
rows
 adding to XAML grid 16
 making equal size in XAML grid 17
 troubleshooting in XAML grid 19

S

sbyte type 159
Scene Gizmo (Unity) 100
Scene view (Unity) 91, 223, 225
security, compared with encapsulation 246
select clause (LINQ) 499
select new clause (LINQ) 499
separation of concerns 310, 374

- sequences, compared with collections 524
- serialization
 - about 552–561
 - finalizers and 595
 - reading entire object 556
 - what happens to objects 553, 555
- set accessor (setter) 254, 257, 373
- SetupGame method 22–25
- Sharpen Your Pencil exercises
 - ability scores 177
 - arrays 202, 204
 - broken code 625–627
 - building paper prototypes 117
 - casting 169
 - C# code 29
 - classes 143–144
 - class models 287–288, 309–310
 - Clown class 135, 136
 - code errors 265–266
 - conditionals 68
 - debugging 43
 - determining game improvements 48
 - enums 437–438
 - File and Directory classes 543–544
 - interfaces 362–363
 - is keyword 381, 383
 - keywords 163–164
 - lambda expressions 511–512, 518
 - LINQ 473–474, 493–494
 - lists 415–416
 - loops 65, 68
 - object references 191–192, 584–586
 - Random class 132–133
 - statements 59
 - TextBlock_MouseDown method 36–37
 - types 167
 - Unicode 569
 - values 161, 162
 - Visual Studio IDE 9–10
 - writing to the console 153
- Shoe Storage app 417–419
- short type 159, 165, 167
- Show Next Statement button 43
- signature (method) 263
- signed types 159
- significant digits 160
- similar behaviors 278
- similar code 278
- Skip method 474
- sliders, adding to bottom row of grid 83
- Sloppy Joe’s Random Menu Item project 208–211, 720–722
- Solution Explorer, opening files in 14
- source control 30–31, 47, 688
- Spy project 240–242
- square brackets [] 106
- StackPanel control 119, 121
- stacks
 - about 445, 614
 - converting to lists 448
 - exercises on 449–450
 - foreach loop 448
 - LIFO (Last In, First Out) 447
 - popping items off 447
- Start Debugging button 23
- statements
 - about 50
 - as building blocks for apps 55
 - exercises on 59
 - in loops 64
 - role of 51
- static Convert class 165
- static fields 394
- static keyword 134
- static methods 134, 394
- static properties 394
- status bar 30
- Step Into button 43
- Step Out button 43
- Step Over button 43
- Stop Debugging button 23, 43
- Stream object 530, 574
- StreamReader
 - about 537, 541
 - building hex dumper using 573

- ReadBlock method 573
- versus FileStreams 571
- streams
 - about 530
 - chaining 538
 - closing 541
 - different types 531
 - forgetting to close 532
 - reading bytes from, using Stream.Read() 574
 - things you can do with 531
 - using file streams to build hex dumper 572
 - using statements 546
 - writing text to files 533
- StreamWriter
 - {0} and {1}. *See* format strings
 - about 533–537, 541
 - Close() method 533
 - using with StreamReader 537
 - versus FileStreams 571
 - Write() and WriteLine() methods 533, 534
- String interpolation 235
- string literals 396
- strings
 - about 161
 - array of 106
 - concatenating 170
 - concatenating, automatic type conversions with + operator 171
 - converting to byte array 541
 - extending 619
 - storage of data in memory as Unicode 566
- String.Substring method 573
- string type 57, 158, 166, 171
- structs
 - about 599
 - boxed 604, 613
 - setting one equal to another 602, 614
 - versus classes 614
 - versus objects 601
- subclasses
 - about 278, 285, 291–293, 306
 - avoiding duplicate code, using inheritance 281
 - child and 280
 - constructors in 308
 - implementing in 297
- inheriting from base class 290
- modifying 291–292
- overriding methods 292
- upcasting 384, 403
- using instead of base classes 286
- using is keyword to access methods in 376–377
- subtraction operator 169
- Sum method (LINQ) 474
- superclass 280, 286, 289, 303
- switch expressions 517
- switch statement 275, 294, 419
- System.Exception 637, 642

T

- \t (tab character) 161, 533
- Tabletop Games Game design... and beyond element 206
- tags (XAML) 11–12, 15–16, 25, 29, 82, 84, 319, 462
- TakeLast method (LINQ) 474
- Take method (LINQ) 474
- T? alias for Nullable<T> 612
- Team Explorer window 30
- test-driven development 528
- Test Explorer window 500
- Testing Game design... and beyond element 501
- TextBlock_MouseDown 37
- TextBlocks
 - adding C# code to update 78
 - adding to XAML grid 18
 - calling event handlers 38
 - exercise on 19–20
 - exercises on 36
 - responding to mouse clicks 34
- TextBox 75–76, 319
- Text property 18
- TextWrapping property 18
- this keyword 198–199, 205, 212, 259, 265, 272
 - distinguishing fields from parameters with same name 266
 - in extension method's first parameter 617
- throw, using to rethrow exceptions 650

time.deltaTime (Unity) 217, 219
 timers, adding 39–41
 Toolbox window 9
 ToString() method 435, 436
 Transform component (Unity) 95, 463–465, 580
 transform.Rotate method (Unity) 217
 triggered events 78
 try blocks 634, 635–636, 650. *See also* exception handling
 try/catch/finally sequence for error handling. *See also* exception handling
 try/finally block 647. *See also* exception handling
 TryParse method 607
 turn-based game 330
 type argument 422
 types
 about 155–226
 arrays 200–201
 automatic casting in C# 171
 char 161
 different types holding different-sized values 205
 exercises on 167
 generic 425
 int, string, and bool types 57
 memory and 166
 multiple references and their side effects 190–192
 object 161
 of literals 162
 referring to objects with reference variables 186–188
 return type 105
 value types 169
 variable 56–57, 165

U

\u escape sequences 567
 uint type 159
 UI (user interface)
 Billard Ball game 453–465
 creating using Visual Designer 3
 mechanics of 71
 ulong type 159
 Unchecked events 230

unhandled exceptions 638
 Unicode 161–162, 561, 563–570, 576
 unit testing 502–506
 Unit Test tool window 500
 Unity
 building games in 344–354
 creating projects using 90
 downloading 89
 GameObject 93–97
 GameObject instances 343–354
 installing with Unity Hub 2, 89–90
 layout of 91
 power of 87, 98
 raycasting 577–586
 scene navigation 651–660
 Unity Hub 89–90
 user interfaces 453–466
 versions 89
 writing C# code for 213–226
 unsigned types 159
 upcasting
 about 384
 entire list using IEnumarable<T> 440
 example of 383
 interfaces and 386
 moving up/down class hierarchy using 382
 using subclass instead of base class 403
 Update method (Unity) 217
 ushort type 159
 using directive 50
 using statements 546, 647
 UTF-8 564, 566–567, 569, 571, 572, 576
 UTF-16 564, 566–567, 576
 UTF-32 567, 576

V

value parameter, set accessors 257
 values 161, 162, 600
 value types
 bool (see bool type)
 byte (see byte type)
 casting 168–170

- changing 205
 - char (see char type)
 - decimal (see decimal type)
 - double (see double type)
 - floating-point types 160
 - int (see int type)
 - long (see long type)
 - more information on 169
 - nullable xxv, 612
 - sbtye 159
 - short (see short type)
 - signed types 159
 - structs as 601
 - TryParse method 607
 - uint 159
 - ulong 159
 - unsigned types 159
 - ushort 159
 - versus objects 600
 - variables
 - about 165
 - assigning values to data type and 168
 - declaring 56
 - generating methods to work with 58
 - reference (see reference variables)
 - using debugger to see changes in 60, 61
 - using operators to work with 62
 - values of 56
 - var keyword 480, 484, 494, 496
 - VerticalAlignment property (XAML) 18
 - virtual keyword
 - about 292, 299
 - apps and 300–301
 - using to inherit behavior 304
 - virtual methods 303
 - Visual Designer 3, 9, 10
 - Visual Studio
 - 2019 Community Edition for Windows 16
 - about 3
 - as Unity script editor 90
 - creating new project 4–5
 - creating new WPF project 8
 - editions and versions of 11
 - for Mac 3–4, 6, 61, 409, 500, 503, 663–674
 - helping you code 53
 - Sharpen your pencil exercise 9–10
 - Test Explorer window 500
 - Unicode and 565
 - Unit Test tool window 500
 - website 2
 - void method 360
 - void return type 124, 141
- ## W
- What Is a Game? game design... and beyond element 7
 - Where method 515
 - while loops 64
 - Who Does What? exercise 27–28
 - Windows
 - emoji panel 25
 - line endings 551
 - requirements for 2
 - Windows UI controls. *See* controls
 - WPF (Windows Presentation Foundation) 3, 6, 11–19, 29, 32, 72, 118–124, 208–211, 230–233, 319–326, 398–402
 - data binding 399
 - data context 399
 - wrench button 79
- ## X
- XAML
 - application code 12
 - combining with C#, creating visual programs 3
 - compared with HTML 32
 - defined 11
 - designing window using 12
 - importance of learning 13
 - setting window size and title 14–15
 - XML documentation 140
- ## Y
- yield return statement 523, 524

Z

zero-based arrays 200

Zoo Simulator project

about 280–286

class hierarchy 284

extending base class 285

inheriting from base class 285



There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning