

O'REILLY®

Fourth
Edition

Head First

C#

A Learner's Guide to
Real-World Programming
with C#, XAML & .NET

Jennifer Greene

Early
Release
RAW &
UNEDITED



A Brain-Friendly Guide

1. 1. Start Building with c#: Build Something Great... Fast!

a. Why you should learn C#

i. Visual Studio is your gateway to C#

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

c. Create your first project in Visual Studio

d. Let's build a game!

i. Your animal match game is a WPF app

e. Here's how you'll build your game

f. Create a WPF project in Visual Studio

i. Visual Studio created a project folder full of files for you

g. Use XAML to design your window

h. Design the window for your game

i. Set the window size and title with XAML properties

j. Add rows and columns to the XAML grid

k. Make the rows and columns equal size

- l. Add a TextBlock control to your Grid
 - m. Now you're ready to start writing code for your game
 - n. Generate a method to set up the game
 - o. Finish your SetUpGame method
 - p. Run your program
 - q. Add your new project to source control
 - r. The next step to build the game is handling mouse clicks
 - s. Make your TextBlocks respond to mouse clicks
 - t. Add the TextBlock_MouseDown code
 - u. Make the rest of the TextBlocks call the same MouseDown event handler
 - v. Finish the game by adding a timer
 - w. Add a timer to your game's code
 - x. Use the debugger to troubleshoot the exception
 - y. Add the rest of the code and finish the game
 - z. Update your code in source control
- aa. Even better if's...

2. I. Unity Lab 1: Start Exploring Unity
3. 2. Dive Into C#: Statements, Classes, and Code
 - a. Let's take a closer look at the files for a console app

- i. A statement performs one single action
- b. Two classes can be in the same namespace (and file!)
- i. The IDE helps you build your code right.
- c. Statements are the building blocks for your apps
- d. Your programs use variables to work with data
 - i. Declare your variables
 - ii. Variables vary
 - iii. You need to assign values to variables before you use them
 - iv. A few useful types
- e. Generate a new method to work with variables
- f. Add code that uses operators to your method
- g. Use the debugger to watch your variables change
- h. Use operators to work with variables
- i. if statements make decisions
 - i. if/else statements also do something if a condition isn't true
- j. Loops perform an action over and over

- i. while loops keep looping statements while a condition is true
- ii. do/while loops run the statements then check the condition
- iii. for loops run a statement after each loop
- k. Use code snippets help to write loops
- l. Some useful things to keep in mind about C# code
- m. Controls drive the mechanics of your user interfaces
- n. Create a WPF app to experiment with controls
- o. Add a TextBox control to your app
- p. Add C# code to update the TextBlock
- q. Add an event handler that only allows number input
- r. Add sliders to the bottom row of the grid
- s. Add C# code to make the rest of the controls work

4. II. Unity Lab 2: Write C# Code for Unity

5. 3. Objects... Get Oriented! Making code make sense

- a. If code is useful, it gets reused
- b. Some methods return a value
- c. Let's build a program that picks some cards

- i. `string[] cards = PickSomeCards(5);`
- d. Create your `PickRandomCards` console app
- e. Finish your `PickSomeCards` method
- f. Your finished `CardPicker` class
- g. Ana's working on her next game
- h. Ana's game is evolving...
- i. ... so how can Ana make things easier for herself?
- j. Build a simple paper prototype for a classic game
- k. Up next: build a WPF version of your card picking app
- l. A `StackPanel` is a container that stacks other controls
- m. Reuse your `CardPicker` class in a new WPF app
- n. Use a `Grid` and `StackPanel` to lay out the main window
- o. Lay out your Card Picker desktop app's window
- p. Ana's prototypes look great...
- q. Ana can use objects to solve her problem
- r. You use a class to build an object
 - i. An object gets its methods from its class

- s. When you create a new object from a class, it's called an instance of that class
- t. A better solution for Ana... brought to you by objects
- u. Theory and practice
- v. An instance uses fields to keep track of things
- w. Methods are what an object does. Fields are what the object knows.
- x. Thanks for the memory
- y. What's on your program's mind
- z. Sometimes code can be difficult to read
- aa. Extremely compact code can be especially problematic
- ab. Most code doesn't come with a manual
- ac. Use intuitive class and method names
- ad. Build a class to work with some guys
- ae. There's an easier way to initialize objects with C#
- af. Use the C# Interactive window to run C# code

6. III. Unity Lab 3: GameObject Instances

7. 4. Types and References: Getting the Reference

- a. Ryan is looking to improve his game
 - i. Storytelling, fantasy, and mechanics

- b. Character sheets store different types of data on paper
- c. A variable's type determines what kind of data it can store
- d. C# has several types for whole numbers
- e. Types for storing really HUGE and really tiny numbers
- f. Let's talk about strings
- g. A literal is a value written directly into your code
 - i. Use suffixes to give your literals types
- h. A variable is like a data to-go cup
 - i. Use the Convert class to explore bits and bytes
 - i. Other types come in different sizes, too
 - j. 10 pounds of data in a 5-pound bag
- k. Casting lets you copy values that C# can't automatically convert to another type
 - i. So what happened?
- l. When you cast a value that's too big, C# adjusts it to fit its new container
- m. C# does some conversion automatically

- n. When you call a method, the arguments need to be compatible with the types of the parameters
 - o. Ryan is constantly improving his game
 - p. Let's help Ryan experiment with ability scores
 - q. And now we can finally fix Ryan's bug
 - r. Use reference variables to access your objects
 - s. References are like sticky notes for your objects
 - t. If there aren't any more references, your object gets garbage-collected
 - u. Multiple references and their side effects
 - v. Two references mean TWO variables that can change the same object's data
 - w. Objects use references to talk to each other
 - x. Arrays hold multiple values
 - i. Use each element in an array like it's a normal variable
 - y. Arrays can contain reference variables
 - z. null means a reference points to nothing
- aa. A Random Test Drive
 - ab. Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!

8. IV. Unity Lab 4: User Interfaces

9. 5. Encapsulation: Keep your privates... private

- a. Let's help Ryan roll for damage
- b. Create a console app to calculate damage
- c. Design the XAML for a WPF version of the damage calculator
- d. The code-behind for the WPF damage calculator
- e. Tabletop talk (or maybe... dice discussion?)
- f. Let's try to fix that bug
- g. Use `Debug.WriteLine` to print diagnostic information
- h. It's easy to accidentally misuse your objects
- i. Encapsulation means keeping some of the data in a class private
- j. Use encapsulation to control access to your class's methods and fields
- k. But is the `RealName` field REALLY protected?
- l. Private fields and methods can only be accessed from instances of the same class
- m. Think of an object as a black box
 - i. Encapsulation makes your classes...
- n. A few ideas for encapsulating classes
- o. Let's use encapsulation to improve the `SwordDamage` class

- i. Is every member of the SwordDamage class public?
- ii. Are fields or methods being misused?
- iii. Is there calculation required after setting a field?
- iv. So what fields and methods really need to be public?

- p. Encapsulation keeps your data safe
 - i. Let's use encapsulation in a simple class

- q. Write a console app to test the MachineGun class
 - i. Our class is well-encapsulated, but...

- r. Properties make encapsulation easier
 - i. Replace the GetBullets and SetBullets methods with a property
- s. Modify your Main method to use the Bullets property
 - i. Debug your MachineGun class to understand how the property works
- t. Auto-implemented properties simplify your code

- i. Use the prop snippet to create an auto-implemented property
- u. Use a private setter to create a read-only property
 - i. Make the BulletsLoaded setter private
- v. What if we want to change the magazine size?
 - i. But there's a problem... how do we initialize MagazineSize?
- w. Use a constructor with parameters to initialize properties
- x. Specify arguments when you use the new keyword
- y. A few useful facts about methods and properties
- z. Objectcross
 - aa. Objectcross solution

10. V. Unity Lab 5: Raycasting

11. 6. Inheritance Your object's family tree

- a. Calculate damage for MORE weapons
- b. Use a switch statement to match several candidates

- c. One more thing... can we calculate damage for a dagger? and a mace? and a staff? and...
- d. When your classes use inheritance, you only need to write your code once
- e. Build up your class model by starting general and getting more specific
- f. How would you design a zoo simulator?
- g. Use inheritance to avoid duplicate code in subclasses
- h. Different animals make different noises
 - i. Think about what you need to override
 - i. Think about how to group the animals
 - j. Create the class hierarchy
 - k. Every subclass extends its base class
 - i. C# always calls the most specific method
 - l. Any place where you can use a base class, you can use one of its subclasses instead
 - m. Use a colon to extend a base class
 - n. We know that inheritance adds the base class fields, properties, and methods to the subclass...
 - i. ...but some birds don't fly!

- o. A subclass can override methods to change or replace members it inherited
- p. Some members are only implemented in a subclass
- q. Use the debugger to understand how overriding works
- r. Build an app to explore virtual and override
- s. A subclass can hide methods in the superclass
 - i. Hiding methods versus overriding methods
 - ii. Use the new keyword when you're hiding methods
 - iii. Use different references to call hidden methods
- t. Use the override and virtual keywords to inherit behavior
- u. A subclass can access its base class using the base keyword
- v. When a base class has a constructor, your subclass needs to call it
- w. A subclass and base class can have different constructors
- x. It's time to finish the job for Ryan
- y. When your classes overlap as little as possible, that's an important design principle called separation of concerns

- i. Use the debugger to really understand how these classes work
- z. Build a beehive management system
 - aa. The beehive management system class model
 - ab. The UI: add the XAML for the main window
 - ac. The Queen class: how she manages the worker bees
 - ad. Feedback drives your Beehive Management game
 - i. Workers and honey are in a feedback loop
 - ae. Your game is turn-based... now let's convert it to a real-time game
 - af. Some classes should never be instantiated
 - ag. An abstract class is an intentionally incomplete class
 - ah. Like we said, some classes should never be instantiated
 - i. Solution: use an abstract class
 - ai. An abstract method doesn't have a body
 - aj. Abstract properties work just like abstract methods

ak. The Deadly Diamond of Death

12. VI. Unity Lab 6: Scene Navigation

13. 7. Interfaces, Casting, and “is” Making Classes keep their Promises

a. The beehive is under attack!

- i. So we need a DefendHive method, because enemies can attack at any time
- ii. We can use casting to call the DefendHive method...
- iii. ... but what if we add more Bee subclasses that can defend?
- iv. An interface defines methods and properties that a class must implement...
- v. ... but there's no limit on interfaces that a class can implement
- vi. Interfaces let unrelated classes do the same job
- vii. Get a little practice using interfaces

b. You can't instantiate an interface, but you can reference an interface

- i. If you try to instantiate an interface, your code won't build...

- ii. ...but use the interface to reference an object you already have
- c. Interface references are ordinary object references
- d. The RoboBee 4000 can do a worker bee's job without using valuable honey
- e. The IWorker's Job property is a hack
- f. Use "is" to check the type of an object
- g. Use the "is" keyword to access methods in a subclass
- h. What if we want different animals to swim or hunt in packs?
- i. Use interfaces to work with classes that do the same job
 - i. Use the is keyword to check if the Animal is a swimmer or pack hunter
- j. C# helps you safely navigate your class hierarchy
- k. C# has another tool for safe type conversion: the as keyword
- l. Use upcasting and downcasting to move up and down a class hierarchy
- m. A CoffeeMaker is also an Appliance
- n. Upcasting turns your CoffeeMaker into an Appliance

- o. Downcasting turns your Appliance back into a CoffeeMaker
 - p. Upcasting and downcasting work with interfaces, too
 - q. Interfaces can inherit from other interfaces
 - r. Interfaces static members work exactly like in classes
 - s. Default implementations give bodies to interface methods
 - t. Add a ScareAdults method with a default implementation
 - u. Data binding updates WPF controls automatically
 - v. Modify the beehive management system to use data binding
 - w. Polymorphism means that one object can take many different forms
 - i. Keep your eyes open for polymorphism!
 - ii. The four core principles of object-oriented programming
 - x. OOPcross
 - y. OOPcross Solution

14. 8. Enums and Collections Organizing your data

- a. Strings don't always work for storing categories of data
- b. Enums let you work with a set of valid values
 - i. An enum defines a new type
- c. Enums let you represent numbers with names
- d. We could use an array to create a deck of cards...
 - i. ...but what if you wanted to do more?
- e. Arrays can be annoying to work with
- f. Lists make it easy to store collections of... anything
- g. Lists are more flexible than arrays
- h. Let's build an app to store shoes
 - i. Generic collections can store any type
 - i. Generic lists are declared using <angle brackets>
- j. Code Magnets
- k. Code Magnets Solution
- l. Collection initializers are similar to object initializers
- m. Let's create a List of Ducks

- i. Here's the initializer for your List of Ducks
- n. Lists are easy, but SORTING can be tricky
 - i. Lists know how to sort themselves
- o. `IComparable< Duck>` helps your list sort its ducks
 - i. An object's `CompareTo` method compares it to another object
- p. Use `IComparer` to tell your List how to sort
 - i. Add an `IComparer` to your project
- q. Create an instance of your comparer object
 - i. Multiple `IComparer` implementations, multiple ways to sort your objects
- r. Comparers can do complex comparisons
- s. Overriding a `ToString` method lets an object describe itself
- t. Override the `ToString` method to see your Ducks in the IDE
- u. Update your foreach loops to let your Ducks and Cards print themselves

- i. Add a `ToString` method to your `Card` object, too
- v. You can upcast an entire list using `IEnumerable<T>`
- w. Use a Dictionary to store keys and values
 - x. The Dictionary functionality rundown
 - y. Your key and value can be different types
 - z. Build a program that uses a dictionary
 - aa. And yet MORE collection types...
 - ab. Generic .NET collections implement `IEnumerable`
 - ac. A queue is FIFO—First In, First Out
 - ad. A stack is LIFO—Last In, First Out
 - ae. Let's build an app to work with decks of cards
 - af. Add a `Deck` class to hold the cards
 - i. Create a `Deck` that extends `ObservableCollection`
 - ii. Add a `Window.Resources` with two instances of the `Deck` class
 - ag. Use Visual Studio to set up data binding
 - 15. 9. LINQ and Lambdas Get Control of your data

- a. Jimmy's a Captain Amazing super-fan...
- b. ...but his collection's all over the place
- c. Use LINQ to query your collections
- d. LINQ works with any `IEnumerable<T>`
 - i. LINQ methods enumerate your sequences
- e. LINQ works with objects
- f. LINQ's query syntax
- g. Anatomy of a query
- h. The var keyword lets C# figure out variable types for you
 - i. When you use var, C# figures out the variable's type automatically
- j. LINQ magnets
- k. LINQ Magnets Solution
- l. LINQ is versatile
- m. LINQ queries aren't run until you access their results
- n. Use a group query to separate your sequence into groups
 - o. Anatomy of a group query
- p. Use join queries to merge data from two sequences

- q. Use the new keyword to create anonymous types
- r. Unit tests help you make sure your code works
 - i. Visual Studio for Windows has the Test Explorer window
 - ii. Visual Studio for Mac has the Unit Test pad
- s. Add a unit test project to your solution
- t. Write your first unit test
- u. Write a unit test for the GetReviews method
- v. Write unit tests to handle edge cases and weird data
- w. Use the => operator to create lambda expressions
- x. A Lambda Test Drive
- y. Refactor a clown with lambdas
- z. Use the ?: operator to make your lambdas make choices
 - aa. Lambda expressions and LINQ
 - ab. Use lambda expressions with methods that take a Func parameter
 - ac. LINQ queries can be written as chained LINQ methods
 - i. The OrderBy LINQ method sorts a sequence

ii. The Where LINQ method pulls out a subset of a sequence

ad. Use the => operator to create switch expressions

ae. Explore the Enumerable class

i. Enumerable.Empty creates an empty sequence of any type

ii. Enumerable.Repeat repeats a value a number of times

iii. So what exactly is an IEnumerable< T > ?

af. Create an enumerable sequence by hand

ag. Use yield return to create your own sequences

ah. Use the debugger to explore yield return

ai. Use yield return to refactor ManualSportSequence

i. Add an indexer to BetterSportSequence

aj. Collectioncross

ak. Collectioncross solution

16. 10. Reading and writing files Save the last byte for me!

a. .NET uses streams to read and write data

b. Different streams read and write different things

- i. Things you can do with a stream:
 - c. FileStream reads and writes bytes to a file
 - d. Write text to a file in three simple steps
 - e. The Swindler launches another diabolical plan
 - f. StreamWriter Magnets
 - g. StreamWriter Magnets Solution
 - h. Use a StreamReader to read a file
 - i. Data can go through more than one stream
 - j. Pool Puzzle
 - k. Pool Puzzle Solution
 - l. Use the File and Directory classes to work with files and directories
 - m. IDisposable makes sure objects are closed properly
 - i. Use the IDE to explore IDisposable
 - n. Avoid filesystem errors with using statements
 - i. Use multiple using statements for multiple objects
 - o. What happens to an object when it's serialized?
 - p. But what exactly IS an object's state? What needs to be saved?

- q. When an object is serialized, all of the objects it refers to get serialized, too...
- r. Use JsonSerializer to serialize your objects
- s. JSON only includes data, not specific C# types
- t. C# strings are encoded with Unicode
- u. Visual Studio works really well with Unicode
- v. .NET uses Unicode to store characters and text
- w. C# can use byte arrays to move data around
- x. Use a BinaryWriter to write binary data
- y. Use BinaryReader to read the data back in
- z. A hex dump lets you see the bytes in your files
- aa. How to make a hex dump of some plain text
- ab. Use StreamReader to build a simple hex dumper
- ac. Use Stream.Read to read bytes from a stream
- ad. Modify your hex dumper to use command-line arguments

Head First C#

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

Andrew Stellman and Jennifer Greene

Head First C#

by Jennifer Greene and Andrew Stellman

Copyright © 2020 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 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 .

Editors: Nicole Tache and Amanda Quinn

Production Editor: FILL IN PRODUCTION EDITOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

December 2020: Fourth Edition

Revision History for the Fourth Edition

- 2020-04-08: First Early Release
- 2020-05-15: Second Early Release
- 2020-07-17: Third Early Release

See [http://oreilly.com/catalog/errata.csp?
isbn=9781491976708](http://oreilly.com/catalog/errata.csp?isbn=9781491976708) for release details.

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

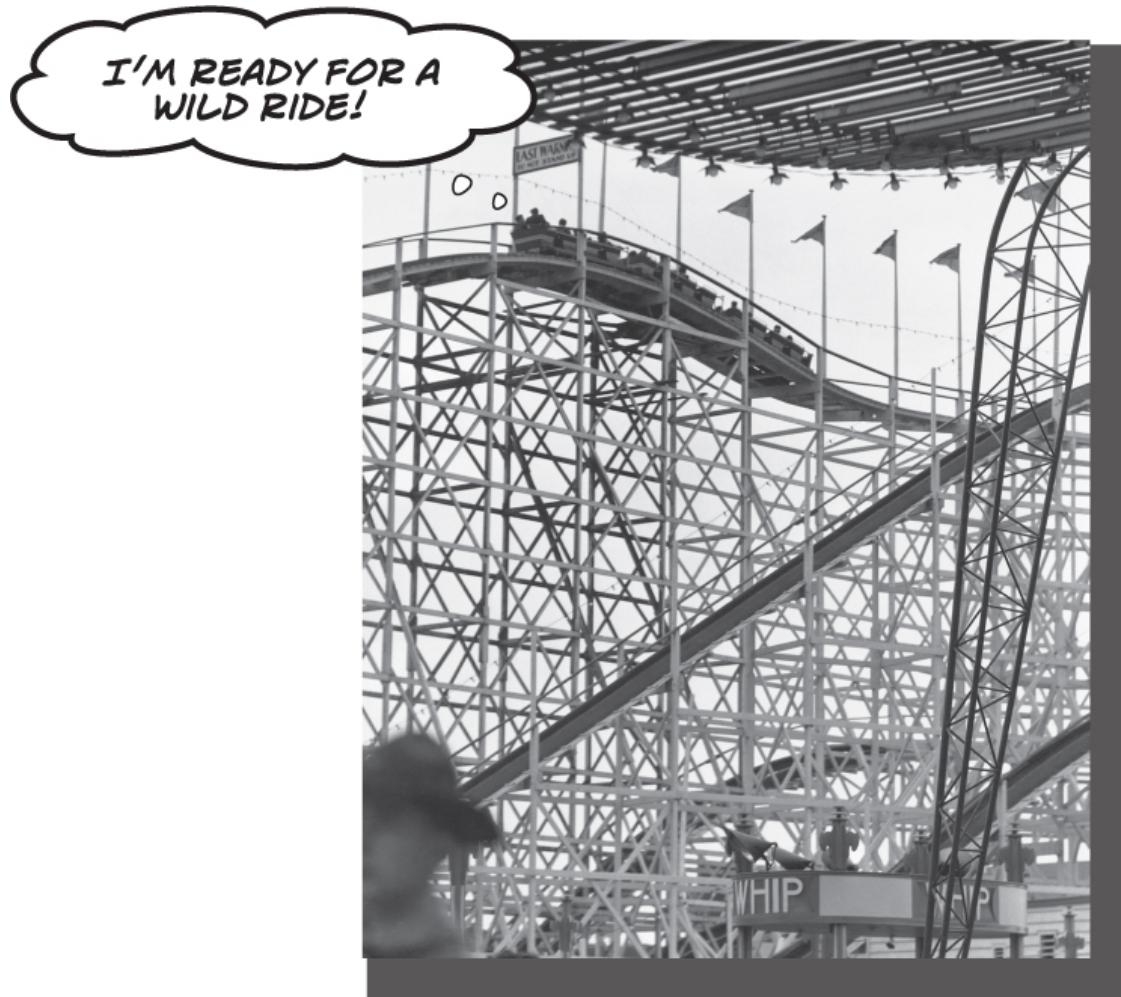
The views expressed in this work are those of the author(s), and do not represent the publisher's views. While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all

responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-97670-8

[FILL IN]

Chapter 1. Start Building with c#: Build Something Great... Fast!



Want to build great apps... right now?

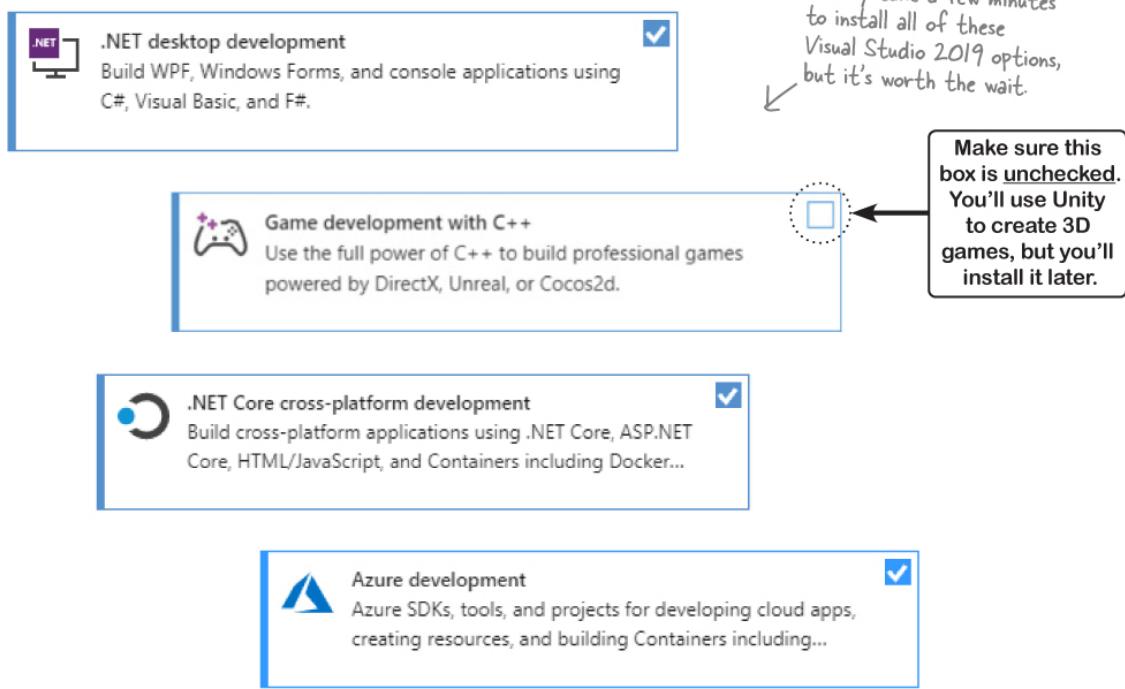
With C#, you've got a **modern programming language** and a **valuable tool** at your fingertips. And with the **Visual Studio IDE**, you've got an **amazing development environment** with highly intuitive features that make coding as easy as possible. But not only is Visual Studio a great tool for writing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

Why you should learn C#

C# is a simple, modern language that lets you do incredible things. And when you learn C#, you're learning more than just a language: C# unlocks the whole world of .NET, an incredibly powerful open source platform for building all sorts of programs: desktop, web, and mobile apps; cloud computing; machine learning and artificial intelligence; 2D and 3D gaming; and much, much more.

Visual Studio is your gateway to C#

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



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

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

You could use Notepad or another text editor to write your C# code, but there's a better way. An **IDE**—that's short for **integrated development environment**—is a text editor, visual designer, file manager, debugger... it's like a multitool for everything you need to write code.

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

- 1. Build an application, FAST.** The C# language is flexible and easy to learn, and the Visual Studio IDE makes it easier by doing a lot of manual work for you automatically. Here are just a few things that Visual Studio does for you:
 - Manages all your project files
 - Makes it easy to edit your project's code
 - Keeps track of your project's graphics, audio, icons, and other resources
 - Helps you debug your code by stepping through it line by line
- 2. Design a great-looking user interface.** The Visual Designer in the Visual Studio IDE is one of the easiest-to-use design tools out there. It does so much for you that you'll find that creating user interfaces for your programs is one of the most satisfying parts of developing a C# application. You can build full-featured professional programs without having to spend hours tweaking your user interface (unless you want to).



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

NOTE

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

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

4. **Learn and explore C# and .NET.** Visual Studio is a world-class development tool, but lucky for us it's also a

fantastic learning tool. **We're going to use the IDE to explore C#,** which gives us a fast track for getting important programming concepts into your brain *fast.*

NOTE

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

Create your first project in Visual Studio

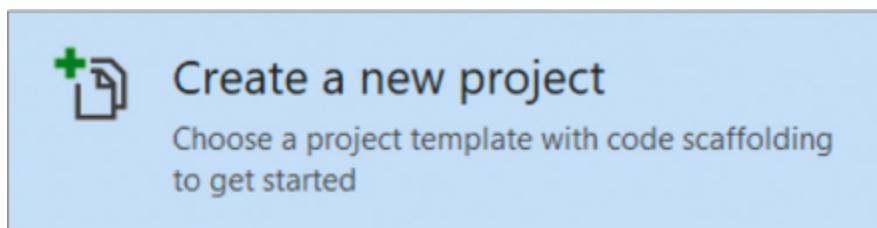
The best way to learn C# is to start writing code, so we're going to use Visual Studio to **create a new project...** and start writing code immediately!

Do this!

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

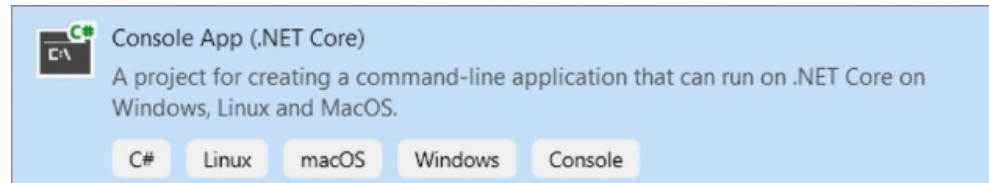
1. Create a new Console App (.NET Core) project.

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

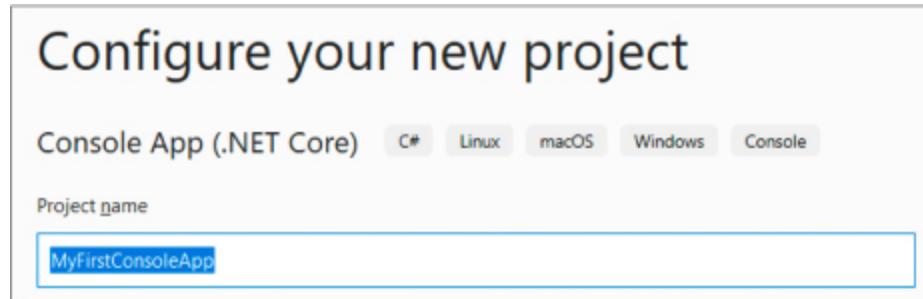


If hit a key and now you see a File menu at the top of the screen, you can create a new project by choosing New >> Project... from the File menu (that is, click File,

then click New, then click Project...). Choose the **Console App (.NET Core)** project type, then press the **Next button**.



Name your project MyFirstConsoleApp.



2. Look at the code for your new app.

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

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

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

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

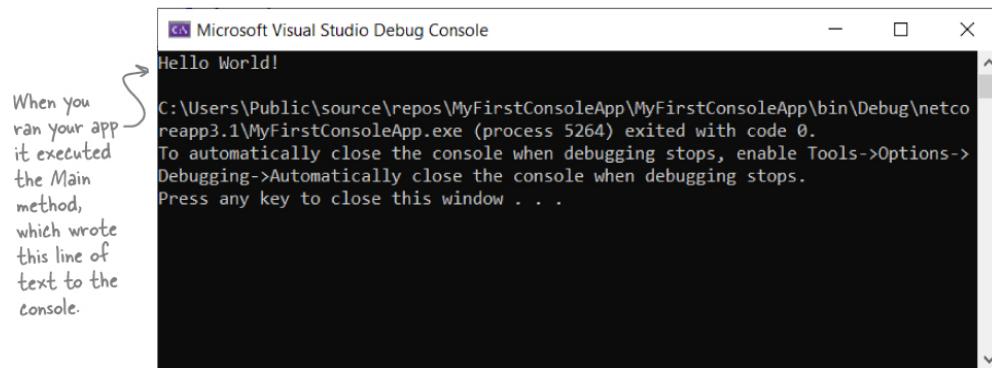
3. Run your new app.

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



4. Look at your program's output.

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



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

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

Hello World!

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

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

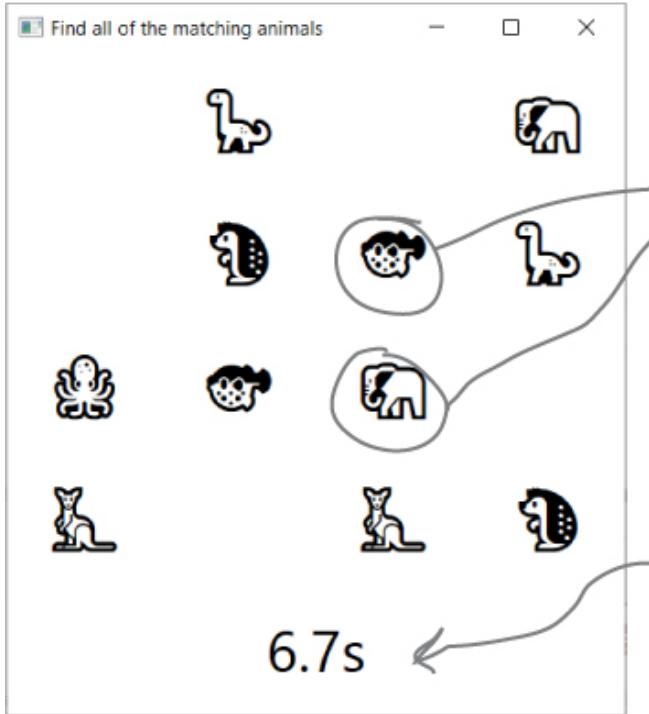
You'll see the same message at the bottom of every Debug Console window. Your program printed a single line of text ("Hello World! ") and then exited. Visual Studio is keeping the output window open until you press a key to close it so you can see the output before the window disappears.

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

Let's build a game!

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

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



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

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

NOTE

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

But C# isn't just for Windows!

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

Your animal match game is a WPF app

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

NOTE

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

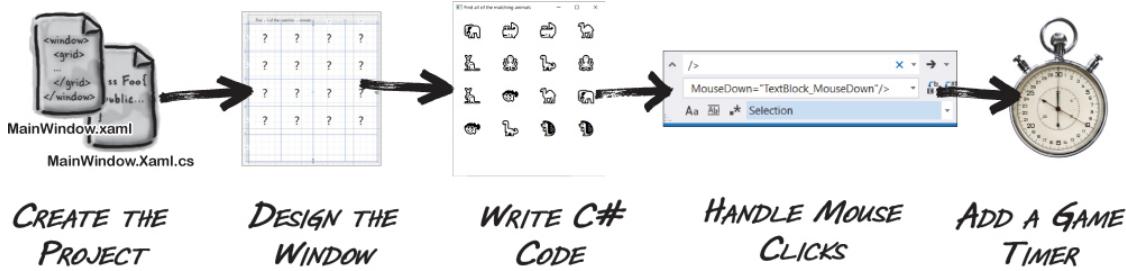
Here’s how you’ll build your game

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

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

NOTE

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



NOTE

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



GAME DESIGN... AND BEYOND

What is a game?

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

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

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

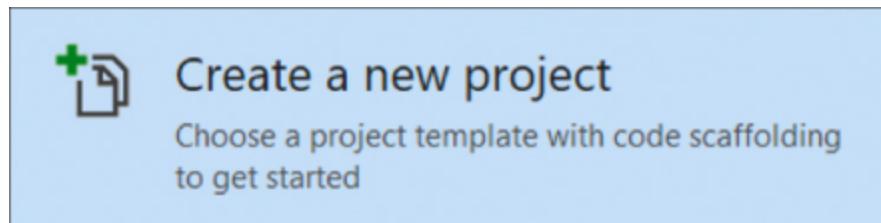


Create a WPF project in Visual Studio

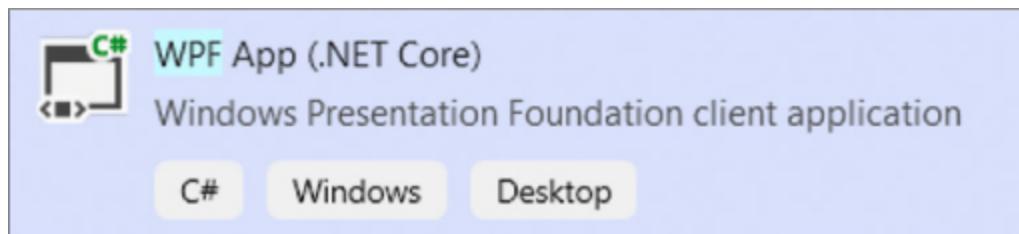
We're going to build an **animal matching game**, where a player is shown a grid of 16 animals and needs to click on pairs to make them disappear. Go ahead and **start up a new instance of Visual Studio 2019** and create a new project:

NOTE

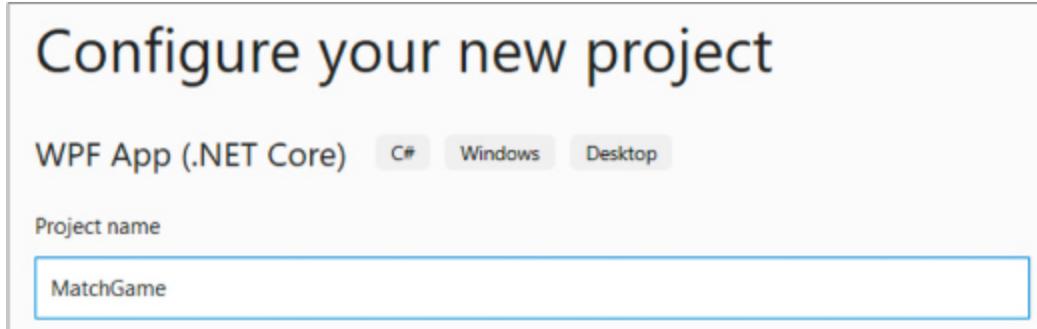
We're done with the Console App project you created in the first part of this chapter, so feel free to close that instance of Visual Studio.



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



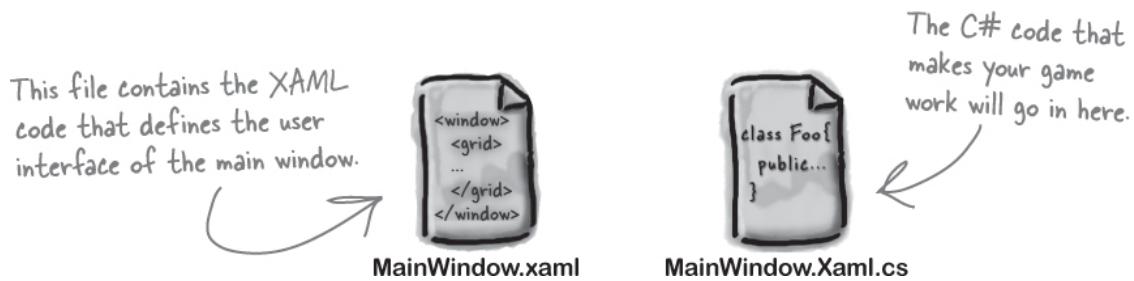
Visual Studio will ask you to configure your project. **Enter MatchGame as the project name** and click Next (you can also change the location to create the project if you'd like):



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

Visual Studio created a project folder full of files for you

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



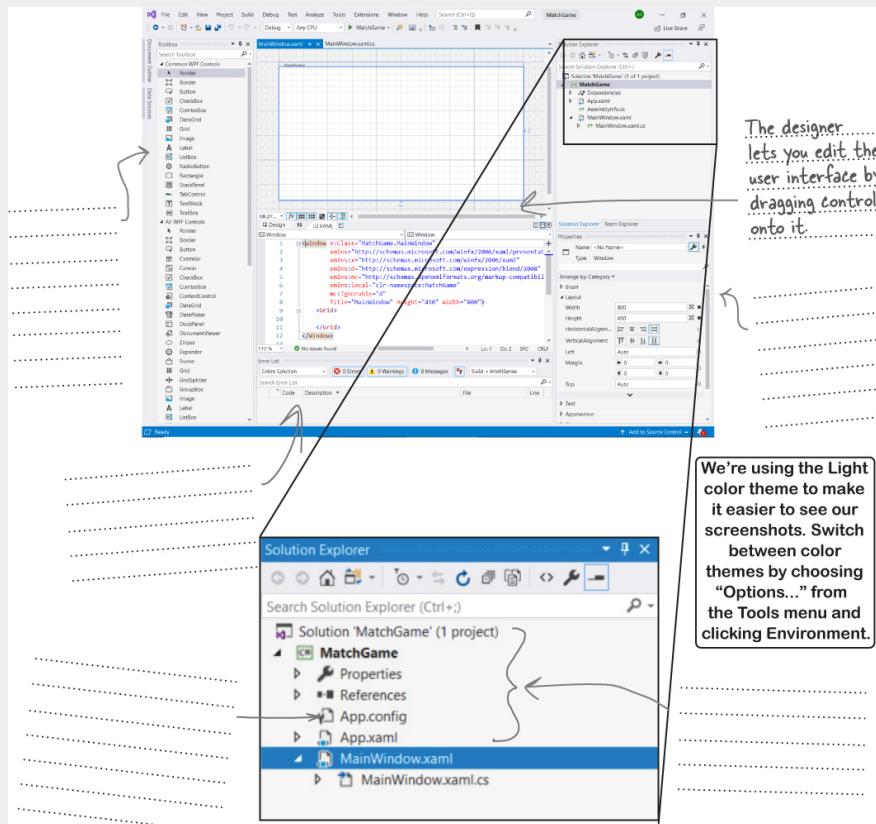


SHARPEN YOUR PENCIL

NOTE

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

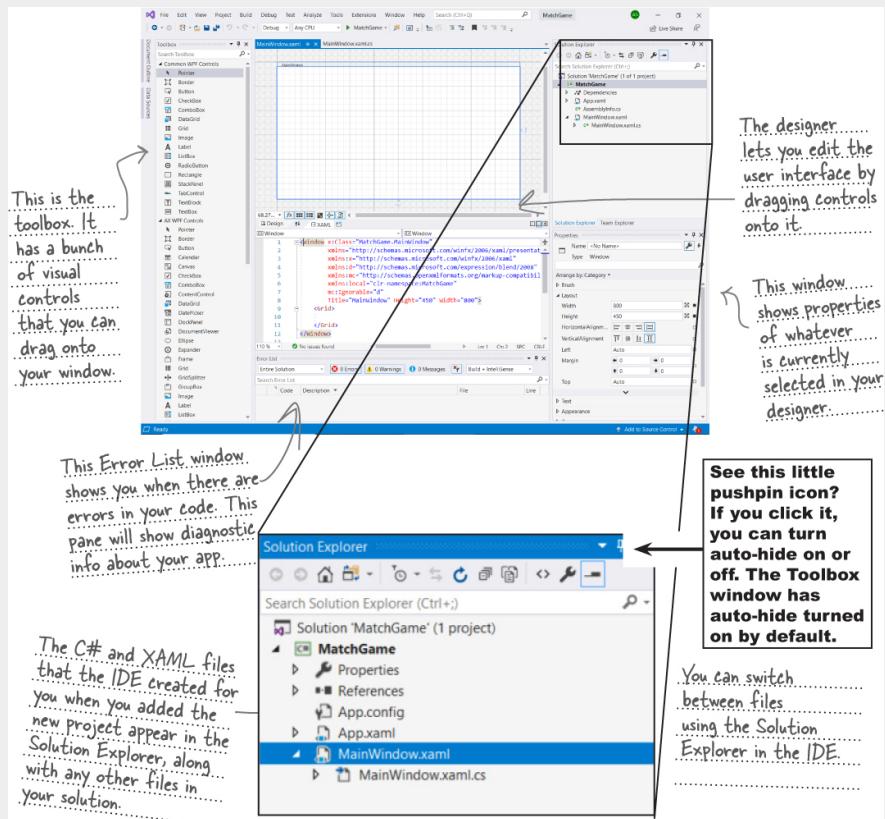
Adjust your IDE to match the screenshot below. First, open **MainWindow.xaml** by double-clicking on it in the Solution Explorer window. Then open the Toolbox and Error List windows by choosing them from the View menu. You can actually figure out the purpose of many of these windows and files based on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





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.



THERE ARE NO DUMB QUESTIONS

NOTE

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

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

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

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

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

Q: Is it OK that I downloaded and installed Visual Studio Community Edition? Or do I need to use one of the versions of Visual Studio that isn't free in order to do everything in this book?

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

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

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

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

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

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

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

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

NOTE

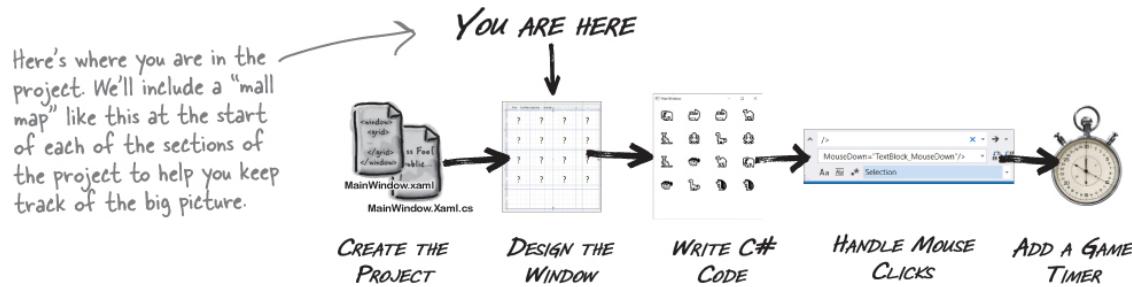
Visual Studio will generate code you can use as a starting point for your applications.

NOTE

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

NOTE

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



Use XAML to design your window

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

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

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

```

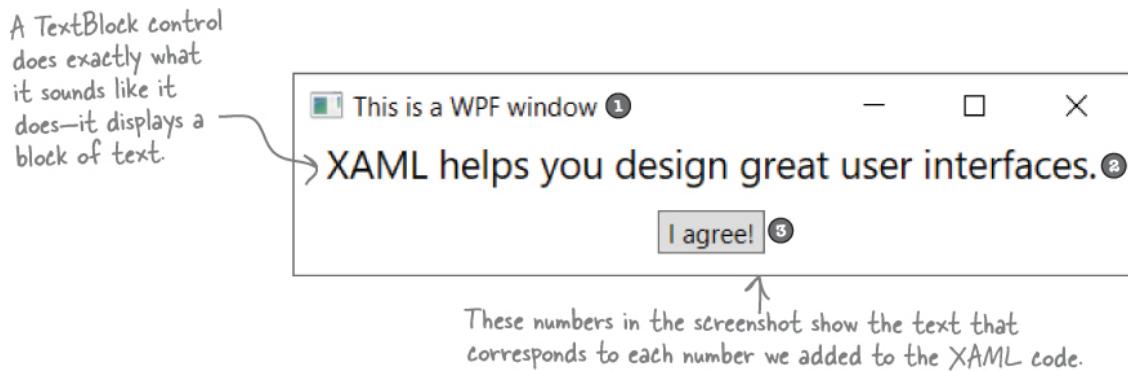
<Window x:Class="MyWPFApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="This is a WPF window" Height="100" Width="400">❶
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
        <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.

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

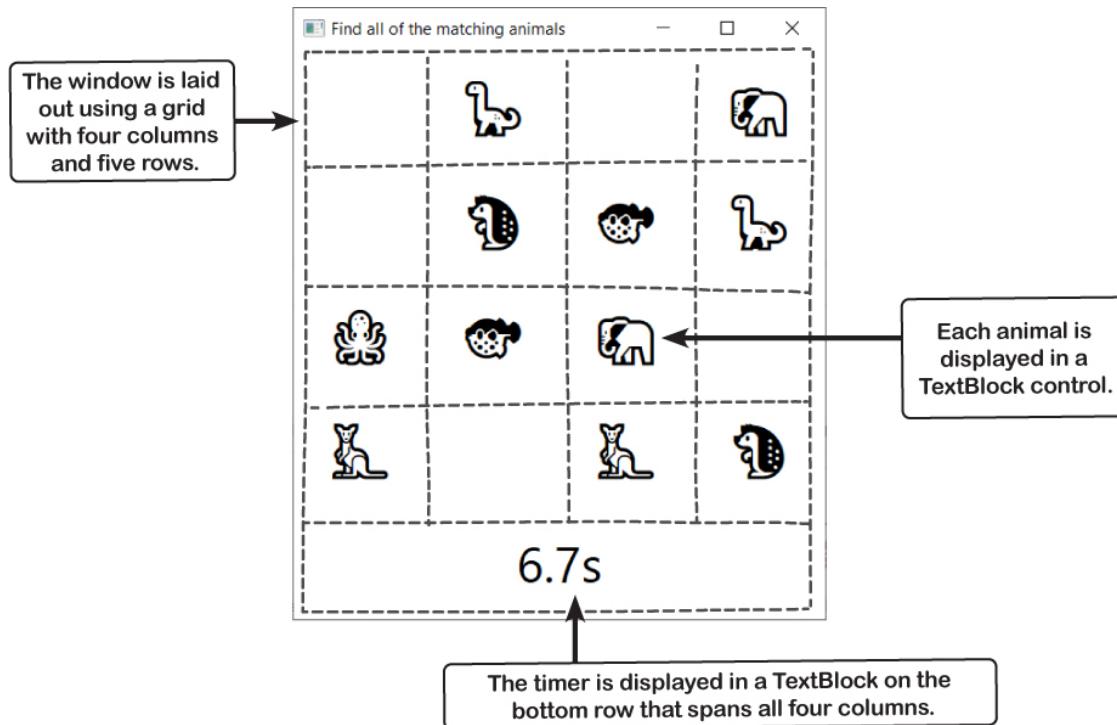


Design the window for your game

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:





RELAX: XAML IS AN IMPORTANT SKILL FOR C# DEVELOPERS.

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

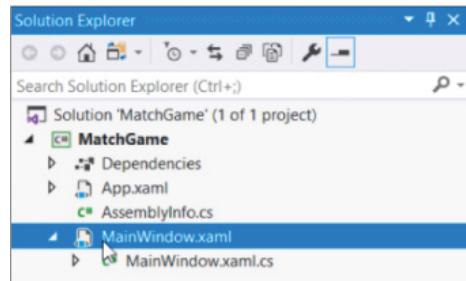
WPF applications use XAML for user interface design—and so do other kinds of C# projects. Not only can you use it for desktop apps, you can also use the same skills to build C# Android and iOS mobile apps with Xamarin Forms, which uses a variant of XAML (with a *slightly* different set of controls).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

Set the window size and title with XAML properties

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

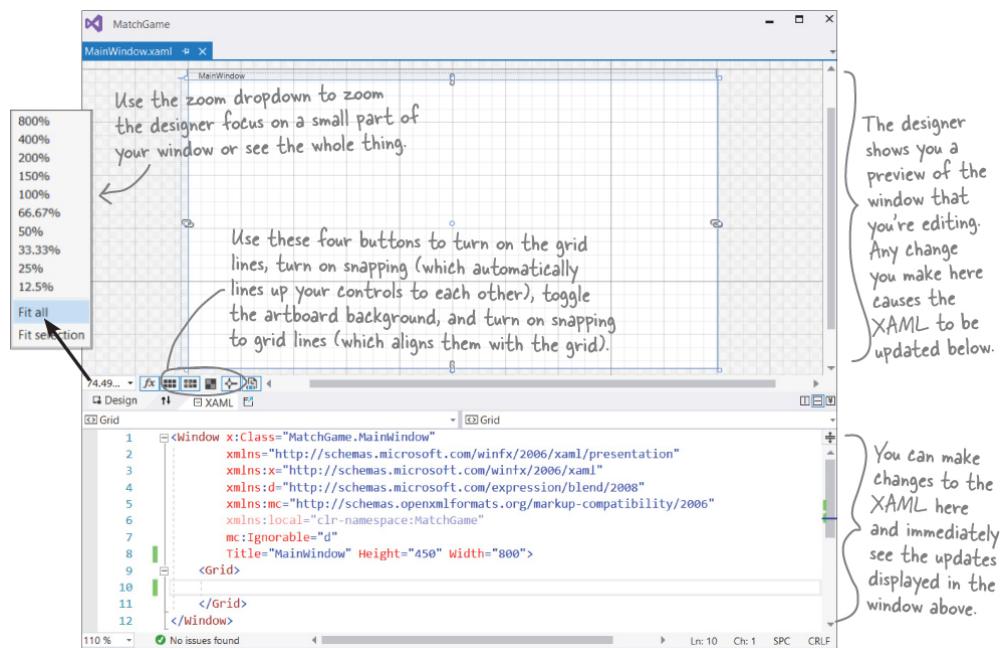
1. Select the main window.

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



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

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

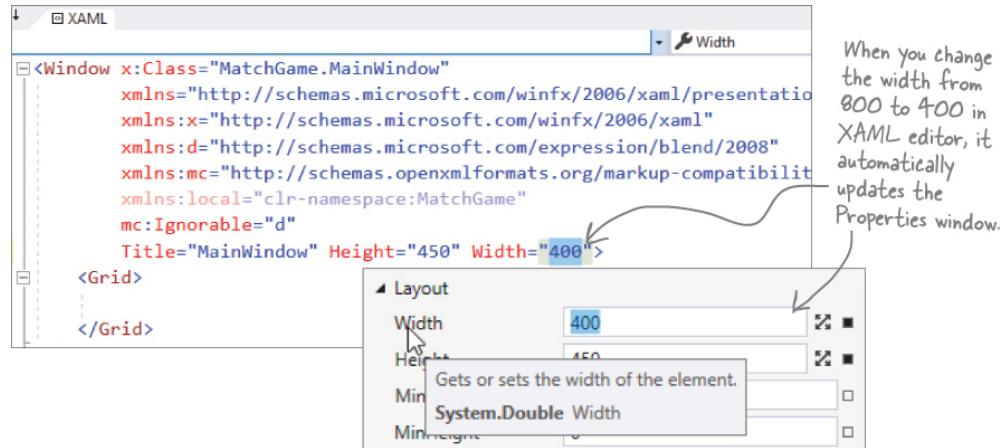


2. Change the size of the window.

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

Expand the Layout section and **change the Width to 400**. The window in the Design pane will immediately

get narrower. Look closely at the XAML code—the Width property is now 400.



3. Change the window title.

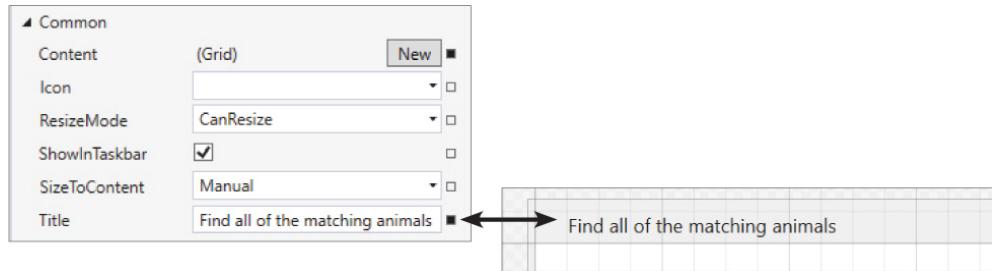
Find this line in the XAML code at the very end of the Window tag:

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

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

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

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



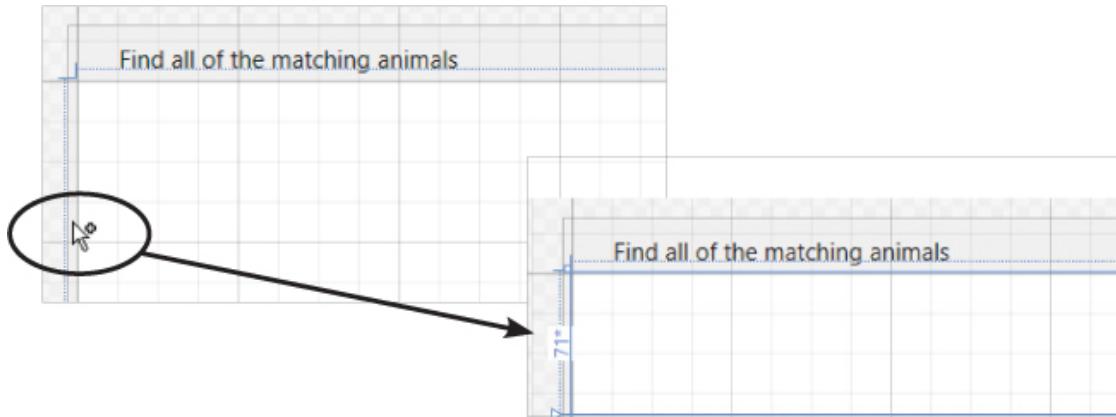
NOTE

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

Add rows and columns to the XAML grid

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

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

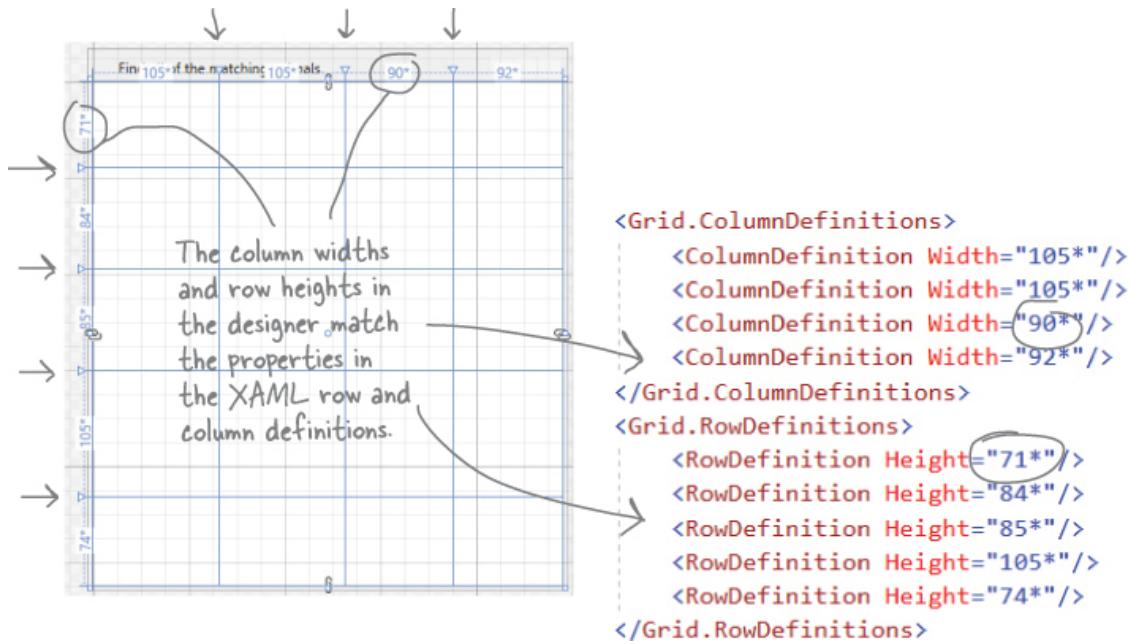


NOTE

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

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

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



NOTE

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



WATCH IT!

Things may look a bit different in your IDE.

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

Make the rows and columns equal size

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



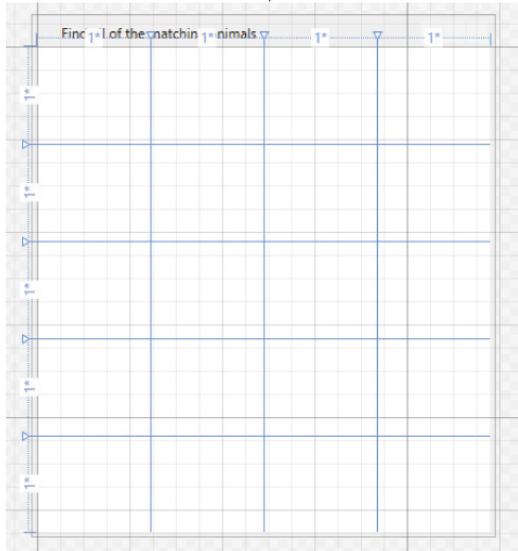
NOTE

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.

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

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

Here's what you should see in the designer:



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

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

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

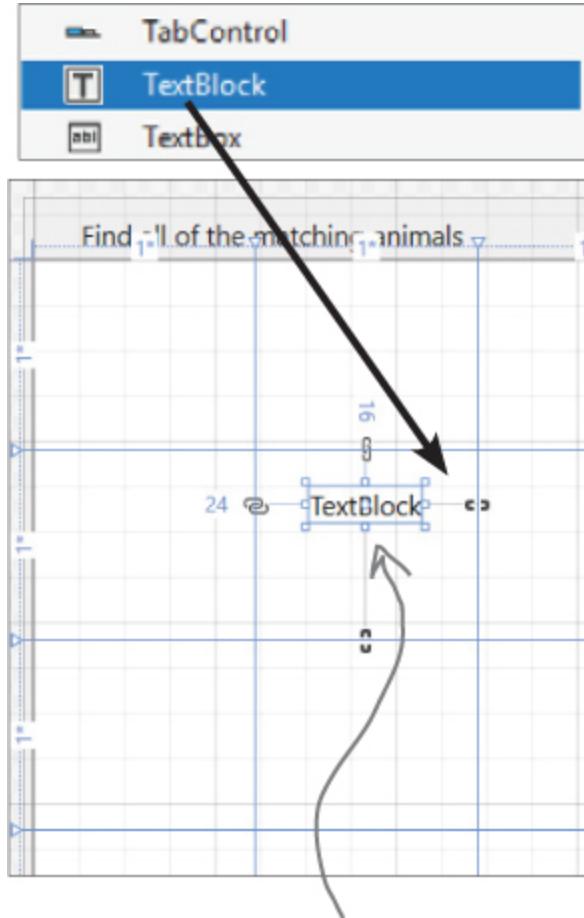
Add a TextBlock control to your Grid

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

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

```
<TextBlock x:Name="textBlock" Text="TextBlock"
  Grid.Column="1" Grid.Row="1"
```

```
HorizontalAlignment="Left" VerticalAlignment="Top"  
Margin="24,16,0,0" TextWrapping="Wrap" />
```



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

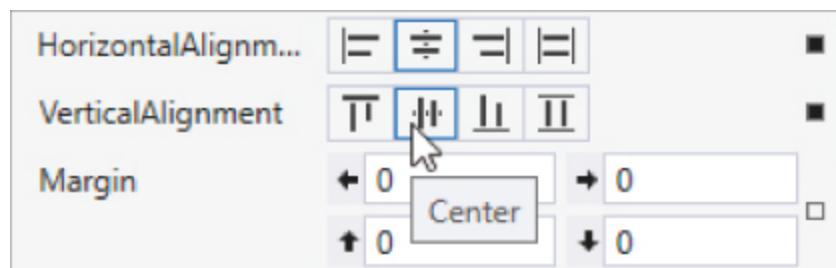
Your properties may be in a different order, and the Margin property will have different numbers because it depends on where in the cell you dragged it.

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

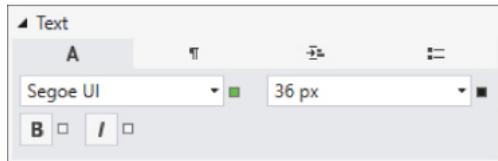


Select the name and delete it—it **will be replaced** with `<No Name>`. Now check your XAML. The `x:Name="textBlock"` property should be gone.

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



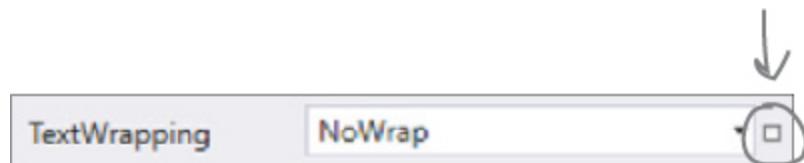
We also want the animals to be bigger, so **expand the Text section** in the Properties window and **change the font size** to `36 px`. Then go to the Common section and change the Text property to `?` to make it display a question mark.



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



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



THERE ARE NO DUMB QUESTIONS

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

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

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

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



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.

D_O

NOTE

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



EXERCISE

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

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

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

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

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



EXERCISE SOLUTION

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

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

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

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

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

?	?	?	?
?	?	?	?
?	?	?	?
?	?	?	?
?	?	?	?

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

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

<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"/>

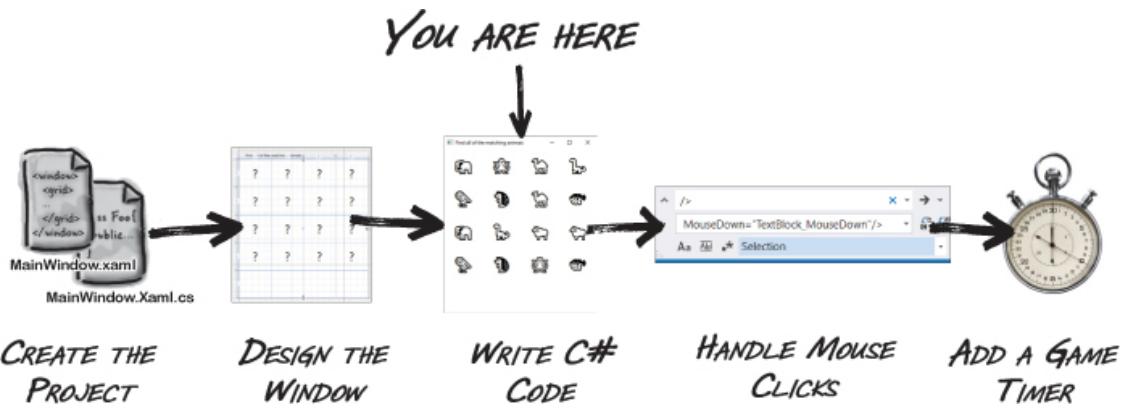
<TextBlock Text="?" FontSize="36" Grid.Row="4" HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="4" Grid.Column="1" HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="4" Grid.Column="2" HorizontalAlignment="Center" 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 okay if you included Grid.Row or Grid.Column properties with a value of 0. We left them out because 0 is the default value.

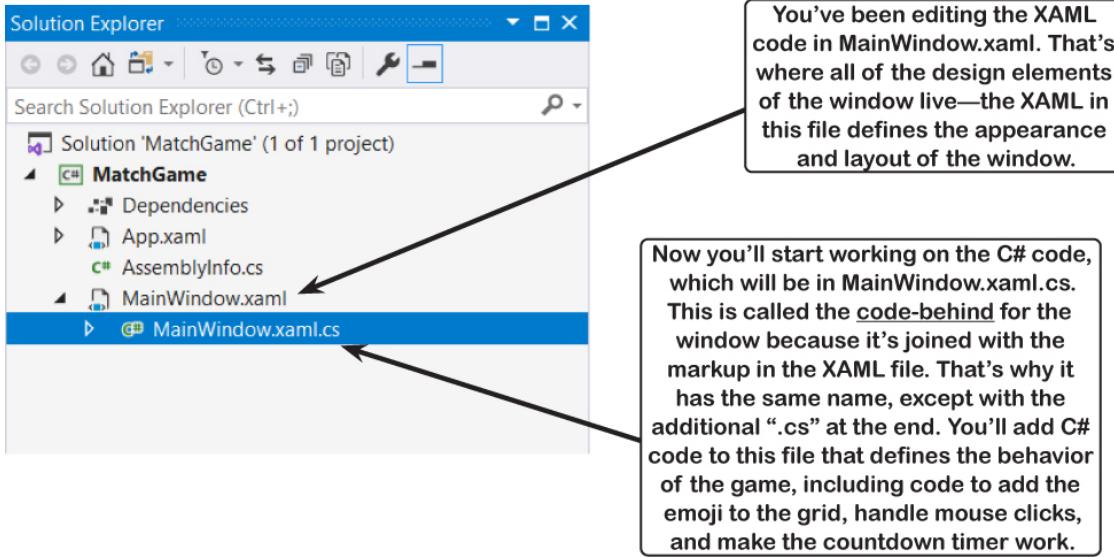
NOTE

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



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

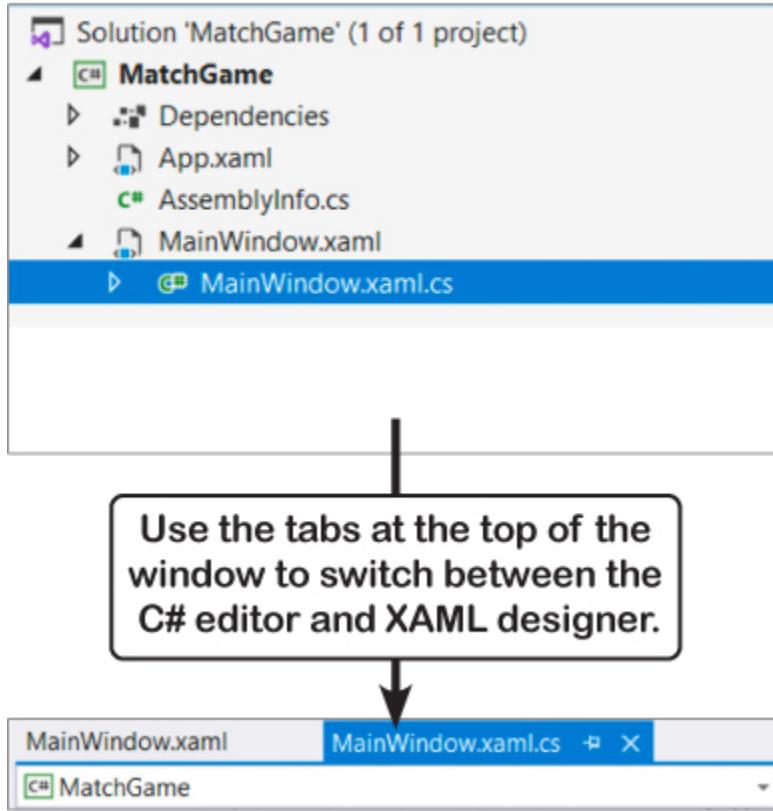
You've finished designing the main window—or at least enough of it to get the next part of your game working. Now it's time to add C# code to make your game work.



Generate a method to set up the game

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.

← Generate this!



1. Open **MainWindow.xaml.cs** in the editor.

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

2. Generate a method called **SetupGame**.

Find this part of the code that you opened:

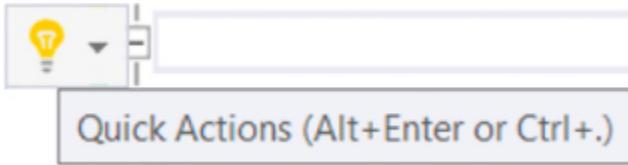
```
public MainWindow();
{
    InitializeComponent();
}
```

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

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



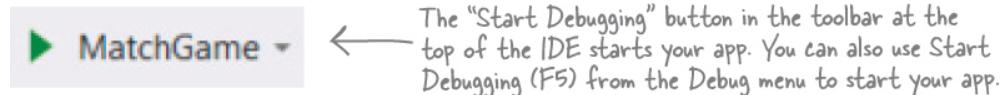
NOTE



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

3. Try running your code.

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



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

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

Exception User-Unhandled
System.NotImplementedException: 'The method or operation is not implemented.'

[View Details](#) | [Copy Details](#) | [Start Live Share session...](#)
[Exception Settings](#)

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

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

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

System.NotImplementedException: ‘The method or operation is not implemented.’

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



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

NOTE

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

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.

NOTE

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



You'll learn a lot more about methods soon.

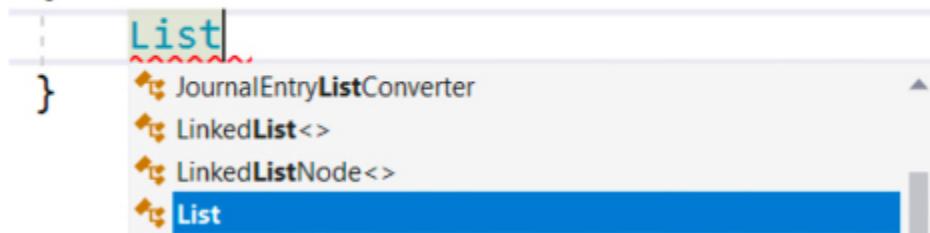
You just used the IDE to help you add a method to your app, but it's okay if you're still not 100% clear on what a method is. You'll learn much more about methods and how C# code is structured in the next chapter.

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

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

```
private void SetUpGame()
```

```
{  
}
```



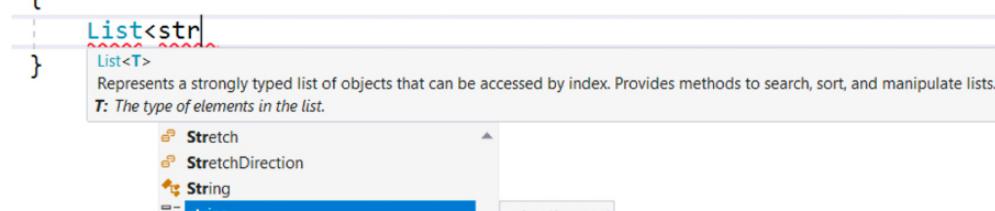
```
List
```

- JournalEntryListConverter
- LinkedList<>
- LinkedListNode<>
- List**

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

```
private void SetUpGame()
```

```
{  
}
```



```
List<str>
```

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

- Stretch
- StretchDirection
- String
- string**

string Keyword

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

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

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

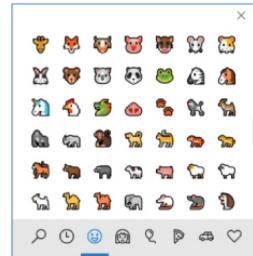
You're using the new keyword to create your List, and you'll learn about that in Chapter 3.

2. Add values to your List.

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

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

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



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);  
}
```

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

The red squiggly line under `mainGrid` is the IDE telling you there's an error: your program won't build because there's nothing with that name anywhere in the code.

Go back to the XAML editor and click on the `<Grid>` tag, then go to the Properties window and enter `mainGrid` in the Name box.

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

NOTE

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

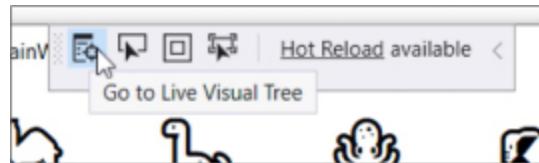
Run your program

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

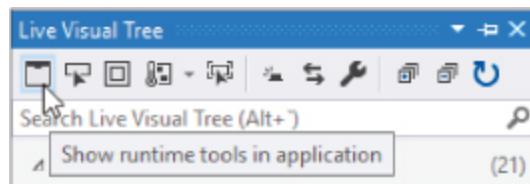


NOTE

When your program first runs, you'll see the runtime tools hovering at the top of the window:



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



Then click the first button in the Live Visual Tree to disable the runtime tools.

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

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



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

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

NOTE

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
<pre>List<string> animalEmoji = new List<string>() { "🐹", "🐹", "🐹", "🐹", "🐹", "🐹", "🐹", "🐹", "🐹", "🐹", "🐹", "🐹", "🐹", "🐹", "🐹", "🐹"; };</pre>	Update the TextBlock with the random emoji from the list
<pre>Random random = new Random();</pre>	Find every TextBlock in the main grid and repeat the following statements for each of them
<pre>foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>()) { int index = random.Next(animalEmoji.Count);</pre>	Remove the random emoji from the list
<pre> string nextEmoji = animalEmoji[index];</pre>	Create a list of eight pairs of emoji
<pre> textBlock.Text = nextEmoji;</pre>	Pick a random number between 0 and the number of emoji left in the list and call it “index”
<pre> animalEmoji.RemoveAt(index);</pre>	Create a new random number generator
	Use the random number called “index” to get a random emoji from the list

WHO DOES WHAT SOLUTION?

C# statement

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

Random random = new Random();

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

What it does

Update the TextBlock with the random emoji from the list

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

Remove the random emoji from the list

Create a list of eight pairs of emoji

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

Create a new random number generator

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



MINI SHARPEN YOUR PENCIL

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

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



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

NOTE

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



BULLET POINTS

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

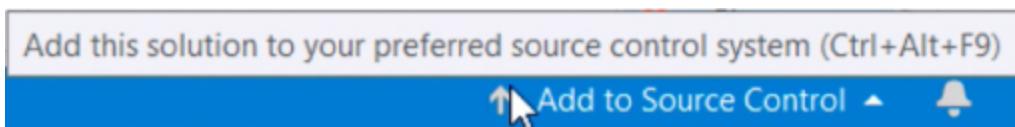
Add your new project to source control

You're going to be building a lot of different projects in this book. Wouldn't it be great if there was an easy way to back them up and access them from anywhere? What if you make a

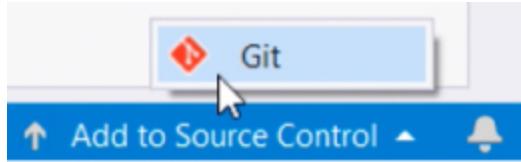
mistake—wouldn’t it be great to roll back to a previous version of your code? Well, you’re in luck! Because that’s exactly what **source control** does: it gives you an easy way to back up all of your code, and keeps track of every change that you make. And Visual Studio makes it really easy for you to add your projects to source control.



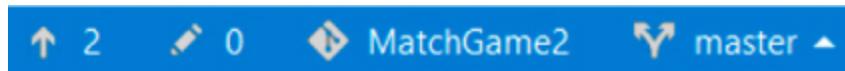
Find **Add to Source Control** in the status bar at the bottom of the IDE.



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



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



Your code is now under source control. Now hover your mouse over **↑ 2**:

2 unpushed commits (Ctrl+E, Ctrl+C)

NOTE

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

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

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

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

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

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

Team Explorer - Synchronization

Push | MatchGame

Backup and share your code. Publish it to a Git service.

Once you've created an account at github.com, click this button to publish your source to GitHub.

▲ Publish to GitHub

 GitHub
GitHub, Inc.

Powerful collaboration, code review, and code management for open source and private projects.

[Publish to GitHub](#)

▲ Push to Azure DevOps Services

 Azure DevOps
Microsoft Corporation

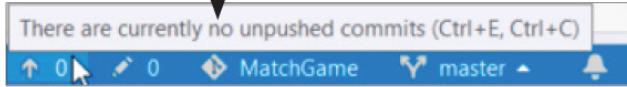
Unlimited free private Git repos, code review, work items, build, and more. [Learn more](#)

Solution Explorer Team Explorer

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

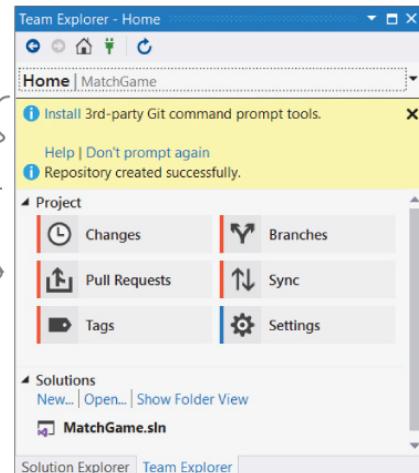
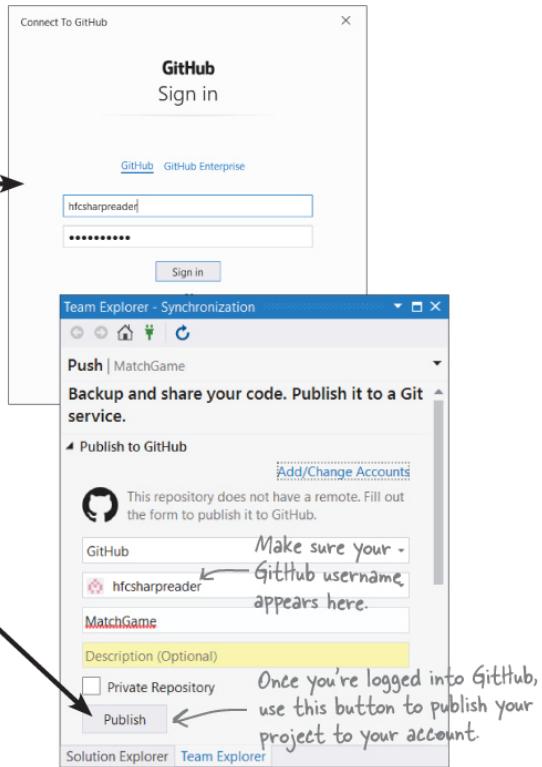
After the IDE logs into GitHub it will display a "Publish to GitHub" form, where you specify the Git provider, user, project name, and indicate whether it should be a private repository. You can keep all of the default options. Press the **Publish button** to publish to GitHub.

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



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

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



NOTE

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

THERE ARE NO DUMB QUESTIONS

Q: Is XAML really code?

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

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

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

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

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

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

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

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

IDE TIP: THE ERROR LIST

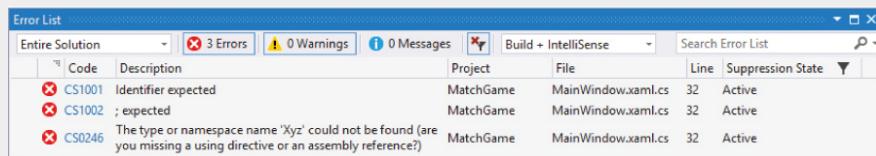
Look at the bottom of the code editor—notice how it says



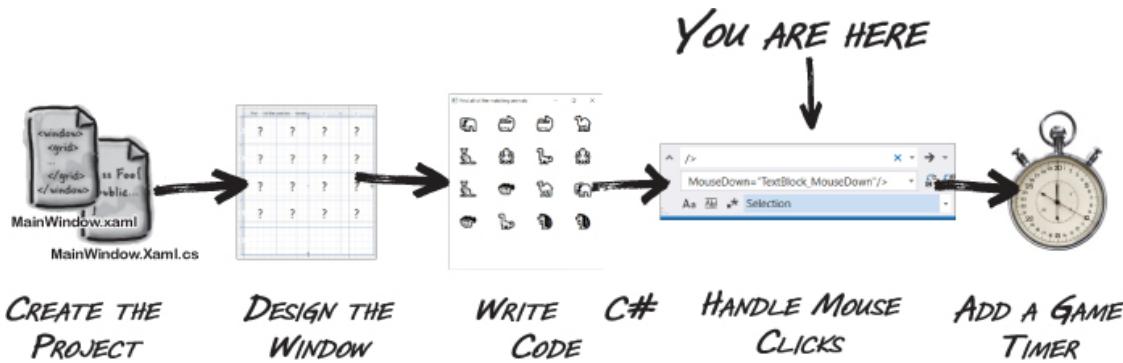
No issues found. That means your code **builds**, which is what the IDE does to turn your code into a **binary** that the operating system can run. Let's break your code.

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

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



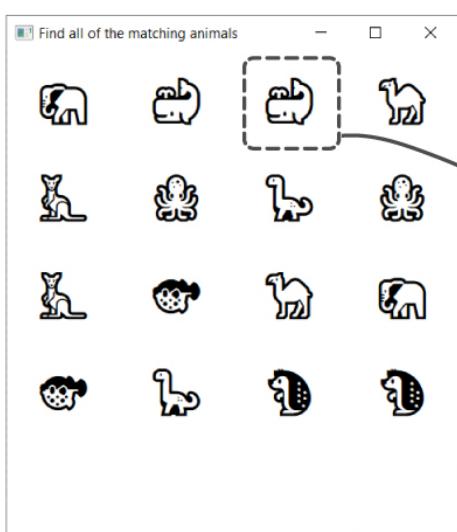
The IDE displayed these errors because Xyz is not valid C# code, and these prevent the IDE from building your code. As long as there are errors in your code it won't run, so go ahead and delete the Xyz line that you added.



The next step to build the game is handling mouse clicks

Now that the game is displaying the animals for the player to click on, we need to add code that makes the gameplay work.

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



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

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

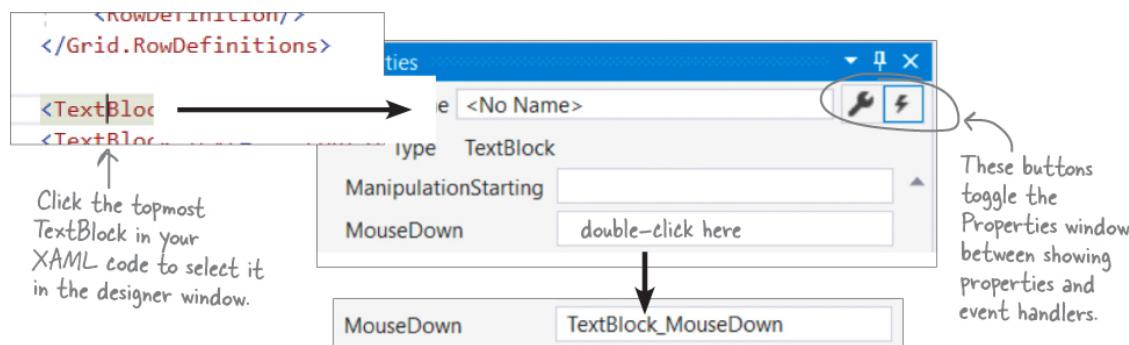
This is a comment. Everything between `/*` and `*/` is ignored by C#. We added this comment to tell you what your `TextBlock_MouseDown` method will do — and also to show you what a comment looks like.

Make your TextBlocks respond to mouse clicks

Your `SetUpGame` method changes the `TextBlocks` to show animal emoji, so you've seen how your code can modify controls in your application. Now we need to write code that

goes in the other direction: your controls need to call your code—and the IDE can help.

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



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

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

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

```
private void TextBlock_MouseDown(object sender,  
    MouseEventArgs e)  
{  
    '  
    '  
    '  
    '  
}
```

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

NOTE

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



SHARPEN YOUR PENCIL

Here's the code for the `TextBlock_MouseDown` method. Before you add this code to your program, read through it and try to figure out what it does. It's okay if you're not 100% right! The goal is to start training your brain to recognize C# as something you can read and make sense of.

```
TextBlock lastTextBlockClikcked;
bool findingMatch = false;

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

1. What does `findingMatch` do?

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

3. What does the block of code starting with `else if (textBlock.Text == 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 okay if you're not 100% right! The goal is to start training your brain to recognize C# as something you can read and make sense of.

```
TextBlock lastTextBlockClikcked;
bool findingMatch = false;

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

NOTE

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:

The screenshot shows the Visual Studio IDE with the file `MainWindow.xaml.cs` open. The code defines a class `MatchGame.MainWindow` with fields `TextBlock lastTextBlockClicked;` and `bool findingMatch = false;`. It contains a method `private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)` that handles mouse down events for `TextBlock` controls. The IDE displays "1 reference" above the method signature. A callout box on the left explains that these are fields: "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." Another callout box on the right explains the reference count: "The IDE displays "1 reference" above the TextBlock_MouseDown method because one TextBlock control has it hooked up to its MouseDown event."

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

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.

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;
    }
}

89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106

110 %  No issues found  Ln: 85  Ch: 9  SPC  CRLF
```

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 `TextBlock`s to it, too. You *could* do it by selecting each one in the designer and entering `TextBlock_MouseDown` into the box next to `MouseDown`. But we already know that just adds a property to the XAML code, so let's take a shortcut.

1. Select the last 15 TextBlocks in the XAML editor.

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

2. Use Quick Replace to add MouseDown event handlers.

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



3. Run the replace over all 15 selected TextBlocks.

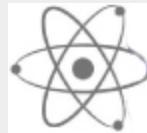
Click the Replace All button () to add the `MouseDown` property to the TextBlocks—it should tell you that 15 occurrences were replaced. Carefully examine the XAML code to make sure they each have a `MouseDown` property that exactly matches the one in the first TextBlock.

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.

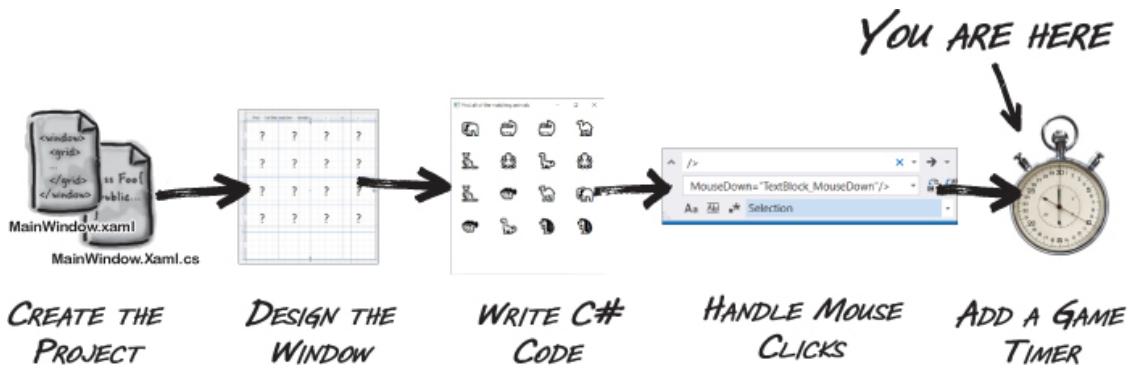
NOTE

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



BRAIN POWER

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



Finish the game by adding a timer

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



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

NOTE



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

Let’s dive right in and add that timer.

Add this!

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

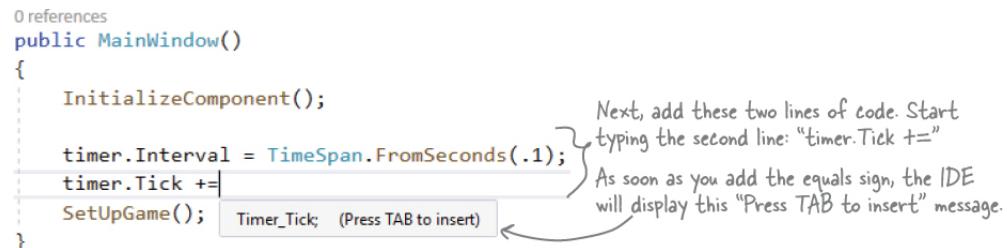
```
namespace MatchGame
{
    using
    System.Windows.Threading;
```

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

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

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

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



```
0 references
public MainWindow()
{
    InitializeComponent();
    timer.Interval = TimeSpan.FromSeconds(.1);
    timer.Tick +=
    SetUpGame();
```

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();

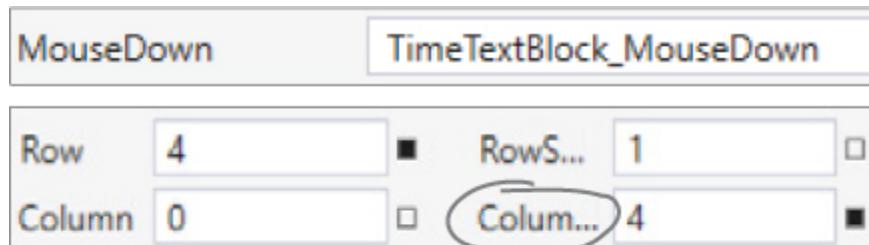
    timer.Interval = TimeSpan.FromSeconds(.1);
    timer.Tick += Timer_Tick;
    SetUpGame();
}

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:



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

- Drag a TextBlock into the lower left square and give it the name timeTextBlock

- Reset its margins, center it in the cell, and set the FontSize to 36px and Text to “Elapsed time”
- Find the ColumnSpan property and set it to 4
- Add a MouseDown event handler called TimeTextBlock_MouseDown

Here's what the XAML will look like:

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

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

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

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

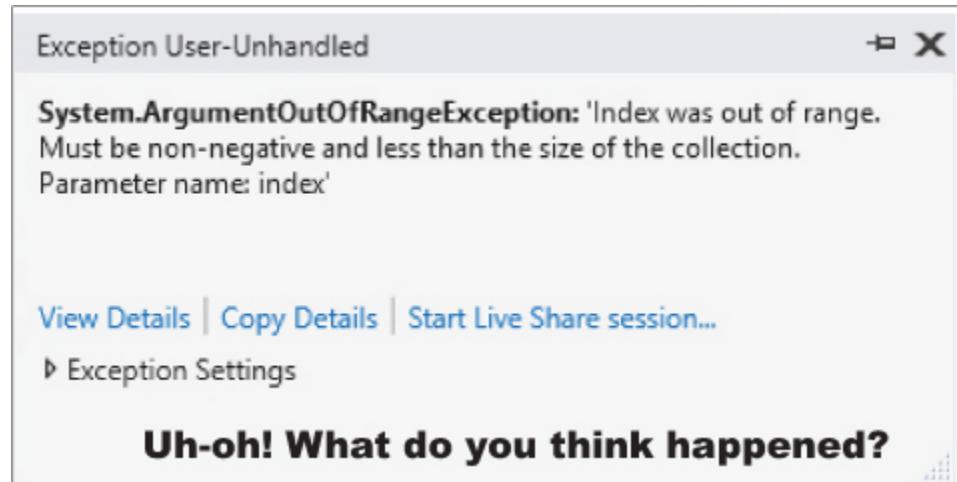
7. And now you have everything you need to finish the Timer_Tick method, which updates the new TextBlock with the elapsed time and stops the timer once the player has found all of the matches:

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

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

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

Can you guess what caused the error?



Use the debugger to troubleshoot the exception

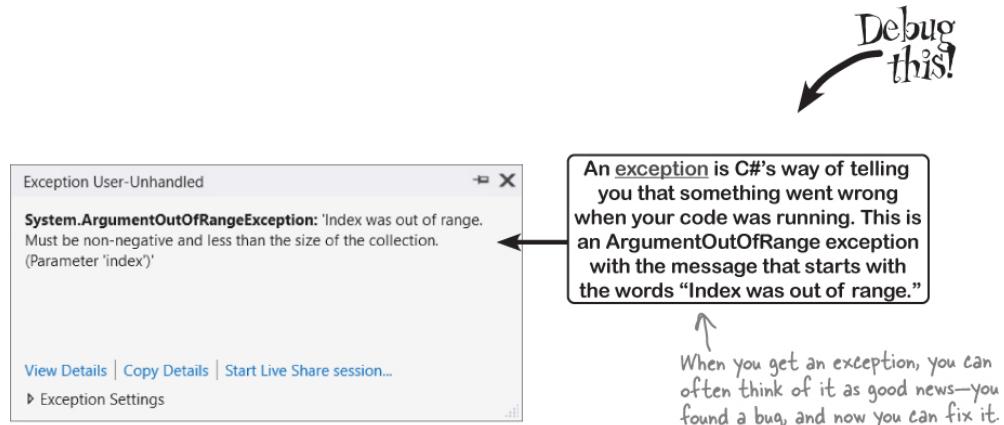
You've heard the word "bug" before. You've probably said something like this to your friends at some point in the past: "That game is really buggy, it has so many glitches." But have you ever really stopped to think about what causes bugs? Every bug has an explanation—everything in your program happens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

Understanding a bug is the first step in fixing it.

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

1. Restart your game a few times.

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



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

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

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

TextBlock lastTextBlockClicked;
bool findingMatch = false;

Exception User-Unhandled

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

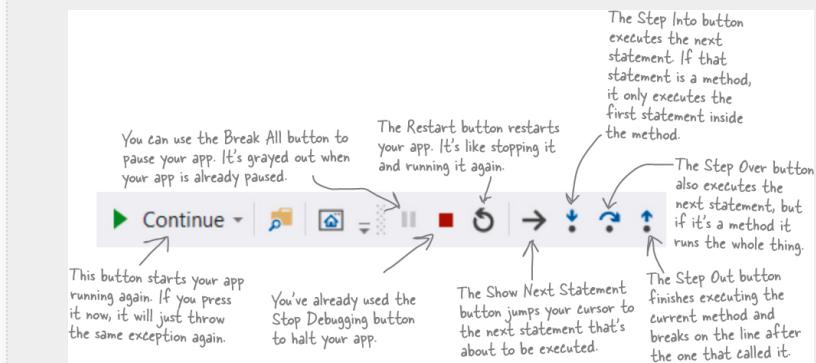
View Details | Copy Details | Start Live Share session...
▶ Exception Settings

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



ANATOMY OF THE DEBUGGER

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



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

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

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

You've just placed a **breakpoint** on the line.

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



SHARPEN YOUR PENCIL

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

1. How many times does your app halt on the breakpoint before the exception? _____
2. A Locals window appears when you're debugging your app. What do you think it does? (If you don't see the Locals window, choose **Debug >> Windows >> Locals (Ctrl D, L)** from the menu.)



SHARPEN YOUR PENCIL SOLUTION

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

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

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

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

▶ 🐾 animalEmoji	Count = 16
-----------------	------------

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

▶ 🐾 animalEmoji	Count = 15
-----------------	------------

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

evidence is that **this all started after we added a new TextBlock to the window.**

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



BEHIND THE SCENES

foreach is a kind of **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 int in a List of ints
```

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

```
The number is 2
The number is 5
The number is 9
The number is 11 } The foreach loop runs the same code over and over again for
each element in the collection, setting the variable to the next
element each time. So in this case, it sets oneNumber to the
next number in the List, and uses it to print a line of text.
```

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

NOTE



Sleuth it out

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. But what caused it to run out of emoji to add?

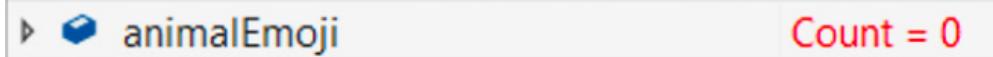
Your program worked before you made the most recent change. Then you added a TextBlock... and then it stopped working. Right inside of a loop that iterates through all of the TextBlocks. A clue... how very, very interesting.

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. And remove the breakpoint by toggling it again or choosing **Delete All Breakpoints (Ctrl-Shift-F9)** from the Debug menu.

```

foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
    if (textBlock.Name != "timeTextBlock") ← Add this if statement inside
    {                                         the foreach loop so that it
        textBlock.Visibility = Visibility.Visible; skips the TextBlock with
        int index = random.Next(animalEmoji.Count); the name timeTextBlock.
        string nextEmoji = animalEmoji[index];
        textBlock.Text = nextEmoji; ← This isn't the only way to fix the bug. One thing
        animalEmoji.RemoveAt(index);       you'll learn as you write more code is that there
    }                                         are many, many, MANY ways to solve any problem...
}                                         and this bug is no exception (no pun intended).

```

Add the rest of the code and finish the game

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

```

    animalEmoji.RemoveAt(index);
}
}

timer.Start();
tenthsOfSecondsElapsed = 0;
matchesFound = 0; } Add these three lines of code to the
                                         very end of the SetUpGame method to
}                                         start the timer and reset the fields.

```

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

```

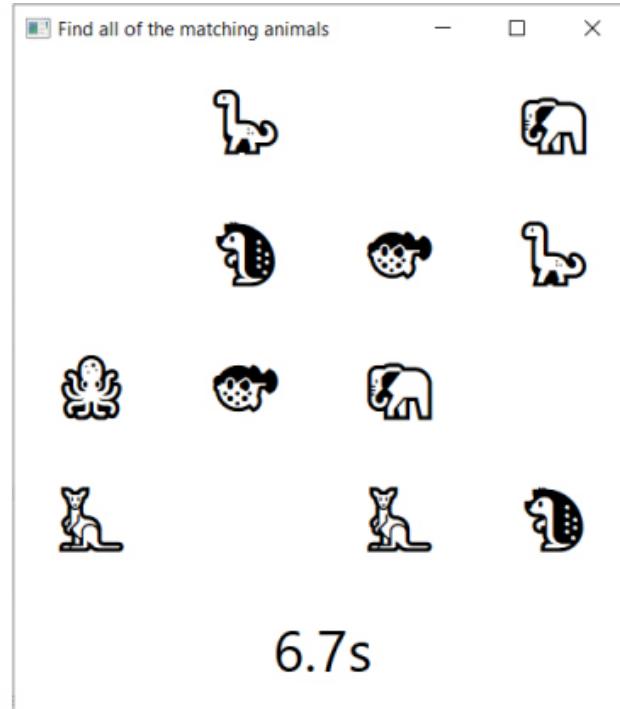
else if (textBlock.Text == lastTextBlockClicked.Text)
{
    matchesFound++; ← Add this line of code to increase
    textBlock.Visibility = Visibility.Hidden; matchesFound by one every time the
    findingMatch = false;                     player successfully finds a match.
}

```

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?

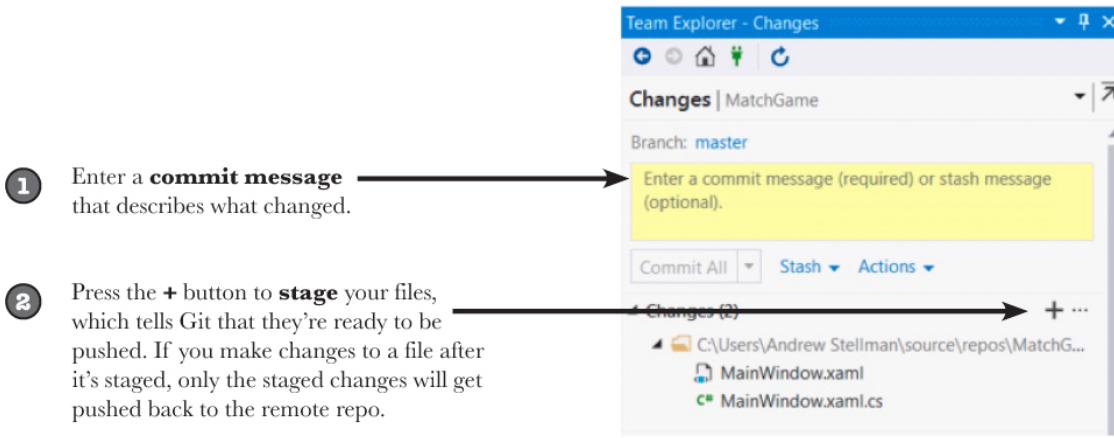


NOTE

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.



- ➊ Enter a **commit message** that describes what changed.
- ➋ Press the **+** button to **stage** your files, which tells Git that they're ready to be pushed. If you make changes to a file after it's staged, only the staged changes will get pushed back to the remote repo.
- ➌ Choose **Commit Staged and Sync** from the commit dropdown (it's right under the commit message box). It may take a few seconds to sync, and then you'll see a success message in Team Explorer.



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



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

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

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

Even better if's...

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

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



MINI SHARPEN YOUR PENCIL

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

NOTE

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



BULLET POINTS

- The IDE tracks the number of times a method is **referenced** elsewhere in the C# or XAML code.
- An **event handler** is a method that your application calls when a specific event like a mouse click, keypress, or window resize happens.
- The IDE makes it easy to **add and manage** your event handler methods.
- The IDE's **Error List window** shows any errors that prevent your code from building.
- **Timers** execute Tick event handler methods over and over again on a specified interval.
- **foreach** is a kind of loop that iterates through a collection of items.
- When your program throws an **exception**, gather evidence and try to figure out what's causing it.
- Exceptions are easier to fix when they're **reproducible**.
- Visual Studio makes it really easy to use **source control** to easily back up your code and keep track of all changes that you've made.
- You can commit your code to a **remote Git repository**. We use Github for the repository with the source code for all of the projects in this book.



Part I. Unity Lab 1: Start Exploring Unity

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

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

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

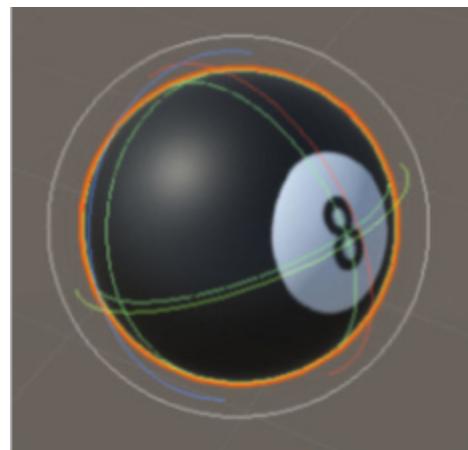
In this first lab, you'll get up and running with Unity. You'll get oriented with the Unity editor, and you'll start creating and manipulating 3D shapes.

Unity is a powerful tool for game design

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

A cross-platform game engine

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

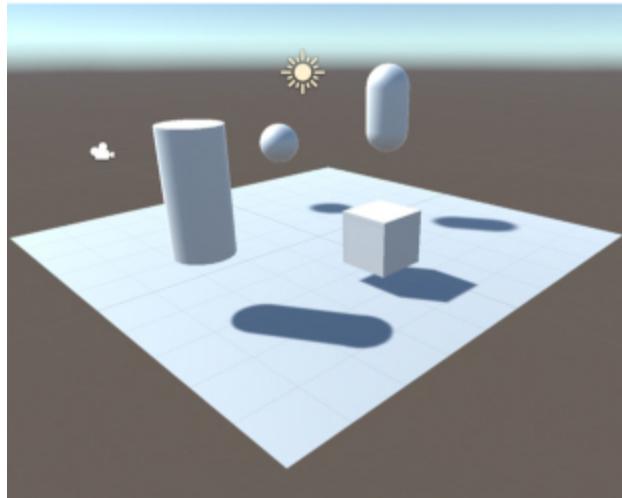


A powerful 2D and 3D scene editor

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

NOTE

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



An ecosystem for game creation

Unity is more than “just” an enormously powerful tool for creating games. It also features an ecosystem to help you build and learn. The Learn Unity page (<https://unity.com/learn>) has

valuable resources to help you learn, and the Unity forums (<https://forum.unity.com/>) help you connect with other game designers and ask questions. The Unity Asset Store (<https://assetstore.unity.com/>) lets you download free and paid assets like characters, shapes, and effects, that you can use in your Unity projects.

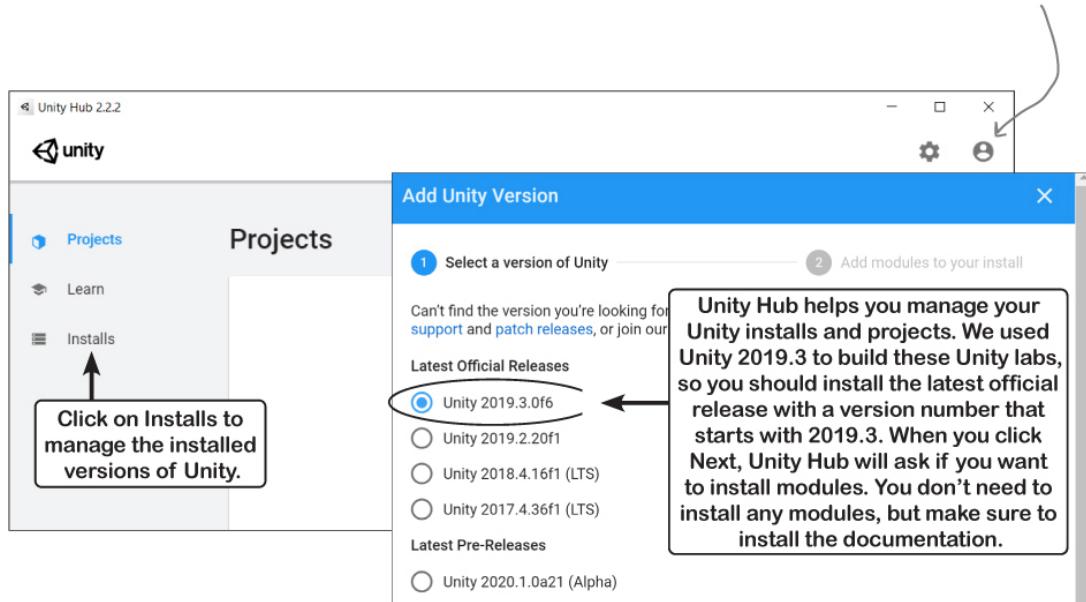
These Unity Labs will focus on using Unity as a tool to explore—and get lots of practice with!—the C# tools and ideas that you've learned throughout the book.

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

Download Unity Hub

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

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 2019.3*** – that's the same version we used to take the screenshots in these labs. Once it's installed, make sure that it's set as the preferred version.

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

You can learn more about installing Unity Hub on Windows, MacOS, and Linux [here](#):

<https://docs.unity3d.com/2019.3/Documentation/Manual/GettingStartedInstallingHub.html>

NOTE

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



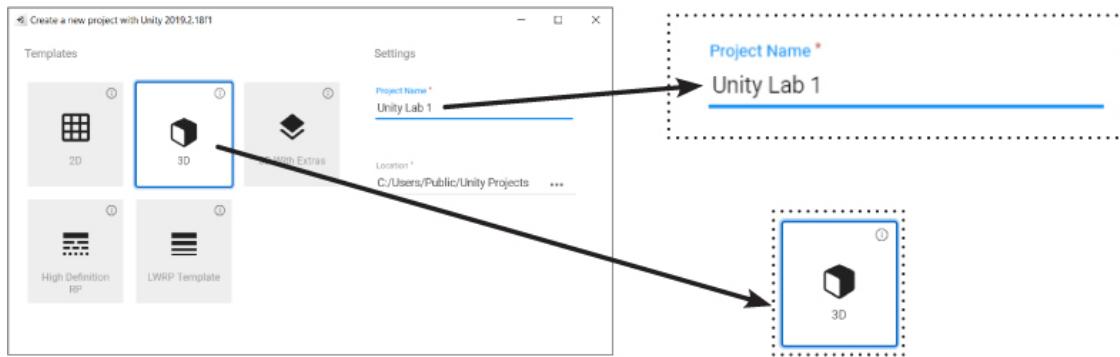
WATCH IT!

Unity Hub may look a little different.

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

Use Unity Hub to create a new project

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

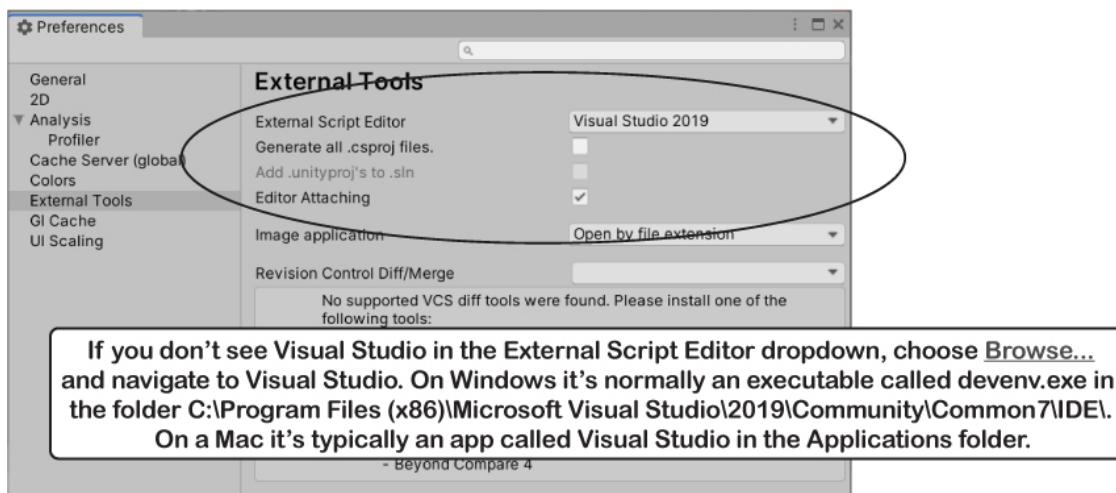


Click Create Project to create the new folder with the Unity project. When you create a new project, Unity generates a lot of files (just like Visual Studio does when it creates new projects for you). It could take Unity a minute or two to create all of the files for your new project.

Make Visual Studio your Unity script editor

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

and **choose Visual Studio** from the External Script Editor window. Make sure the **Editor Attaching box is checked**, too—this will let you debug your Unity code in the IDE.



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

NOTE

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

Take control of the Unity layout

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

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



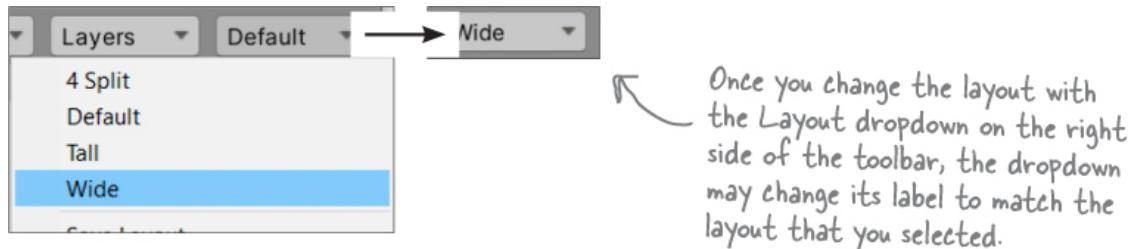
Try docking it inside or next to other panels, then drag it to the middle of the editor to make it a floating window.

NOTE

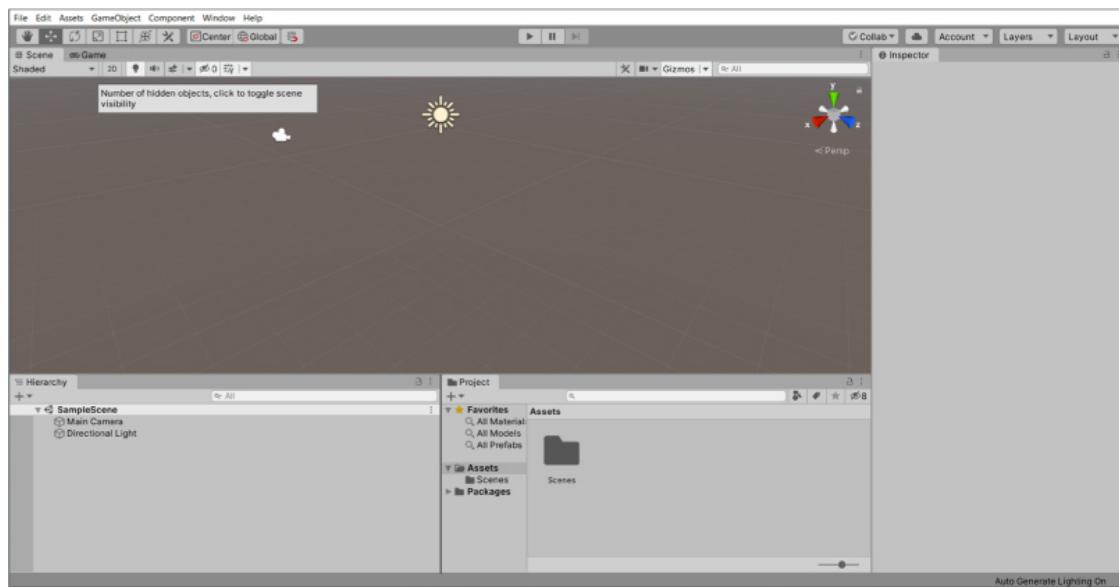
The **Scene view** is your main interactive view of the world that you're creating. You use it to position 3D shapes, cameras, lights, and all of the other objects in your game.

Choose the Wide layout to match our screenshot

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



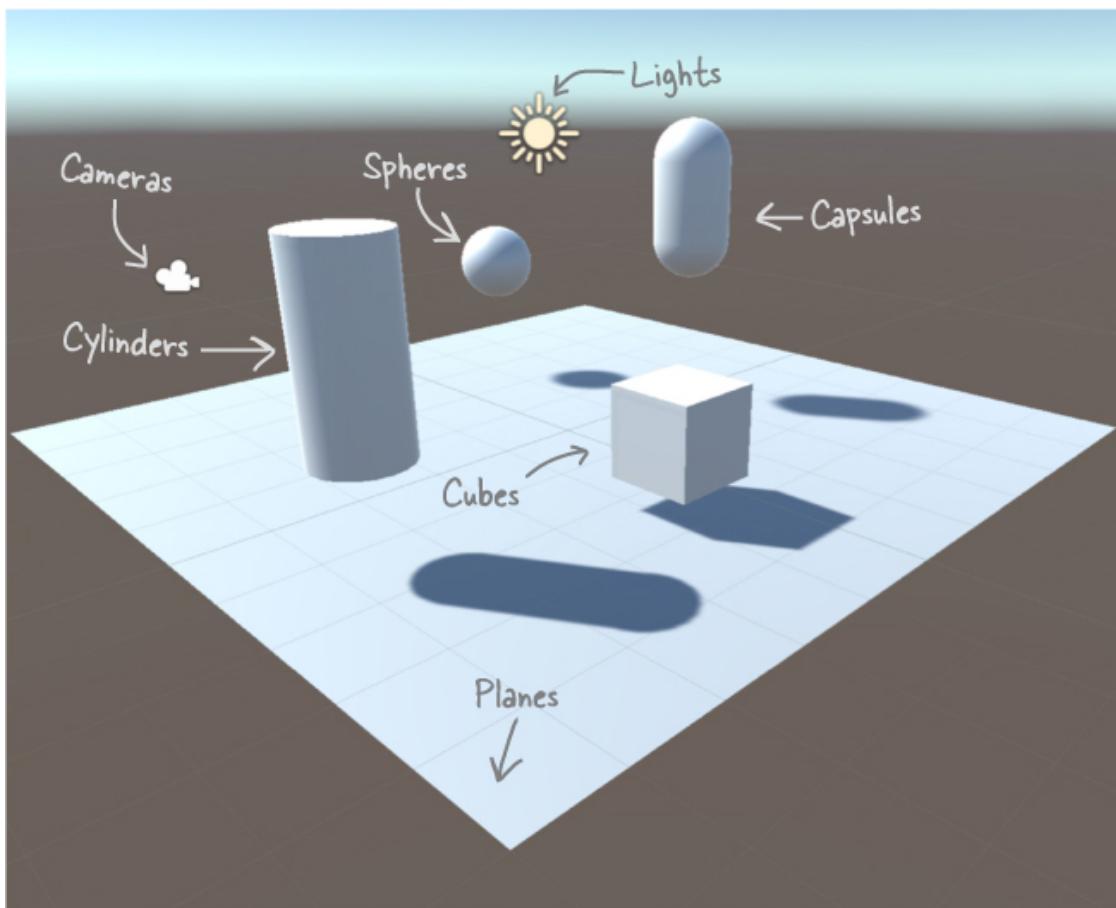
Here's what your Unity editor should look like in the Wide layout.



Unity games are made with GameObjects

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

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



NOTE

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

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

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

ren-der, verb.

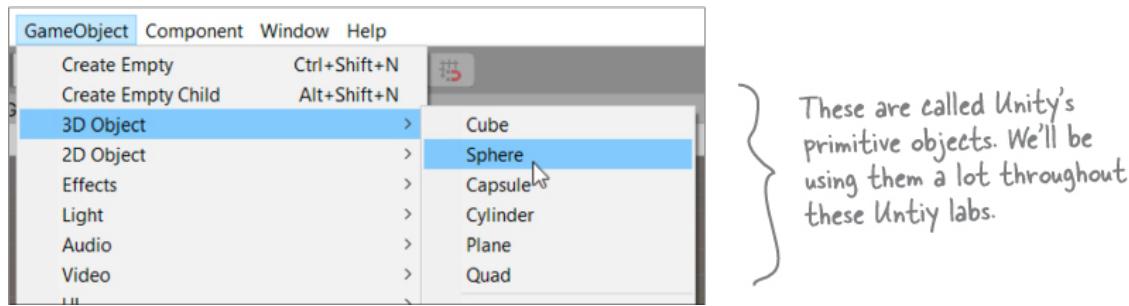
to represent or depict artistically.

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

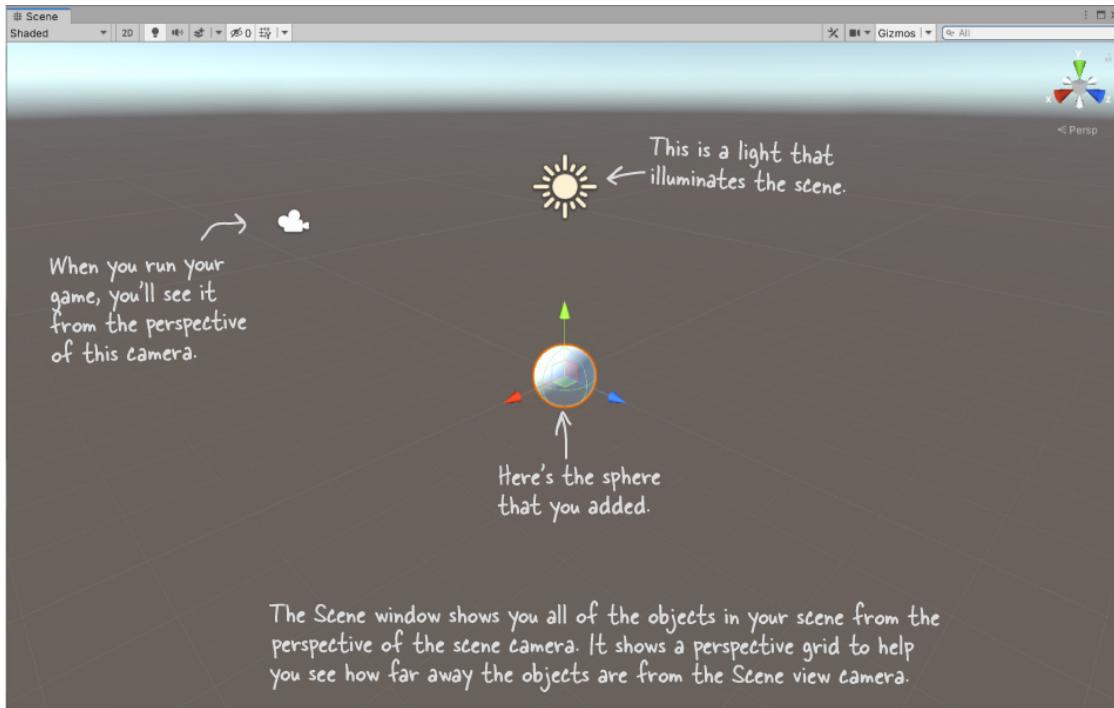
Your scene is a 3D environment

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:

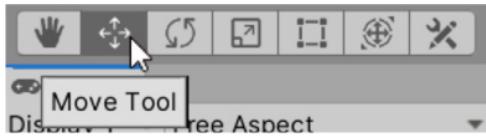


A sphere will appear in your Scene window. Everything you see in the Scene window is shown from the perspective of the **Scene view camera**, which “looks” at the scene and captures what it sees.



Use the Move Gizmo to move your GameObjects

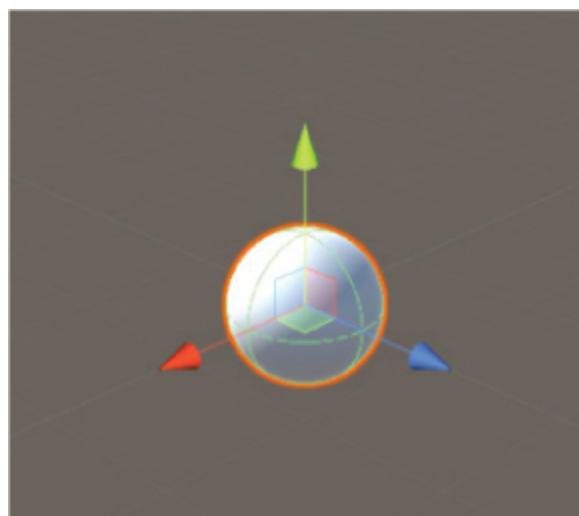
The toolbar at the top of the Unity Editor lets you choose Transform Tools. If the Move Tool isn't selected, press its button to select it.



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

The Move Tool lets you use the **Move Gizmo** to move GameObjects around the 3D space. You should see red, green,

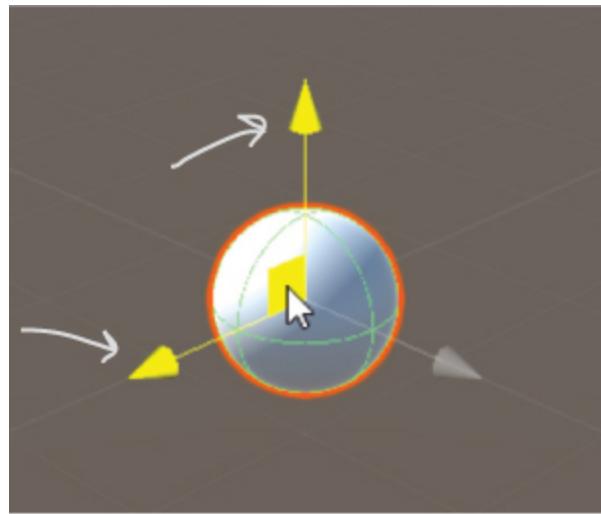
and blue arrows and a cube appear in the middle of the window. This is the Move Gizmo, which you can use to move the selected object around the scene.



NOTE

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

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



NOTE

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

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

NOTE

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

The Inspector shows your GameObject's components

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

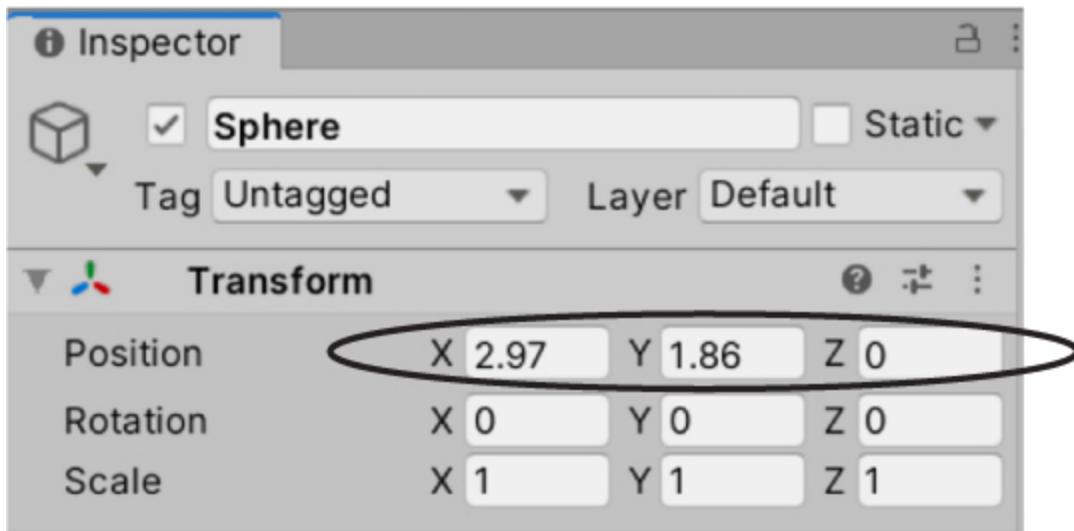
NOTE

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

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.



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

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

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



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

NOTE

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

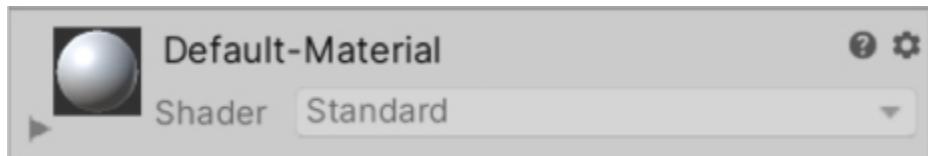
Save your scene often! Use File >> Save or Ctrl-S to save the scene right now.

Add a material to your sphere GameObject

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

1. Select the sphere.

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



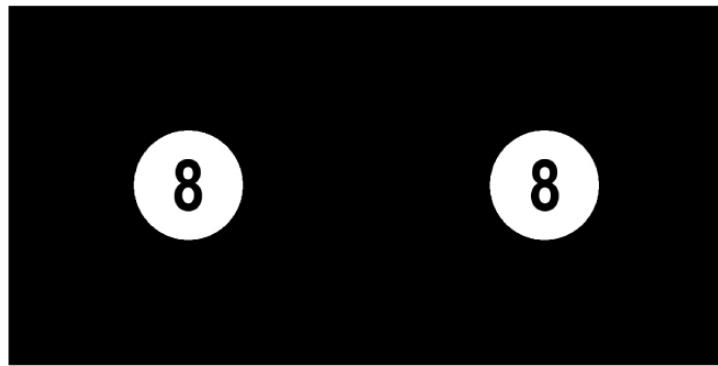
We'll make your sphere more interesting by adding a **texture**—that's just a simple image file that's wrapped around a 3D shape, almost like you printed the picture on a rubber sheet and stretched it around your object.

2. Go to our Billiard Ball Textures page on GitHub.

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

3. Download the texture for the 8 ball.

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



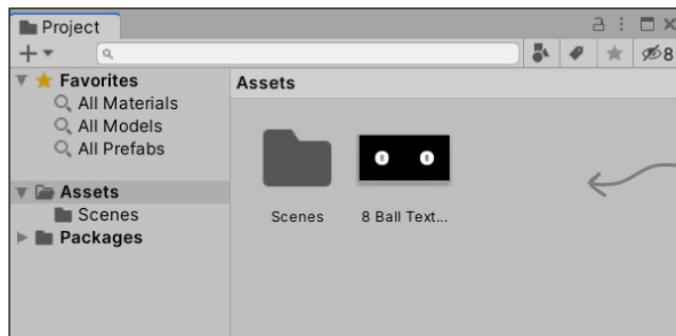
We designed this image file so that it looks like an 8 ball when Unity "wraps" it around a sphere.

Download the file into a folder on your computer.

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

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

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

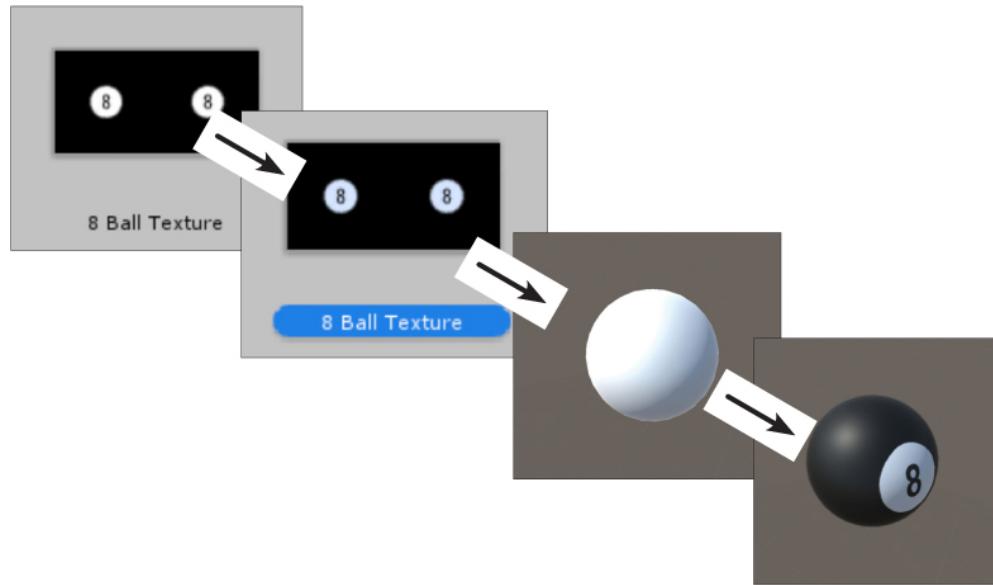


You right-clicked inside the Assets folder in the Project window to import the new asset, so Unity imported the texture into that folder.

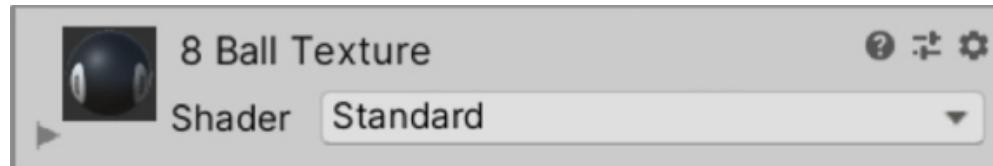
5. Add the texture to your sphere.

Now we just need to take that texture and “wrap” it around your sphere. Click on 8 Ball Texture in the

Project window to select it. Once it's selected, drag it onto your sphere.



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





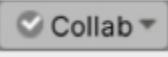
Unity is a great way to really “get” C#.

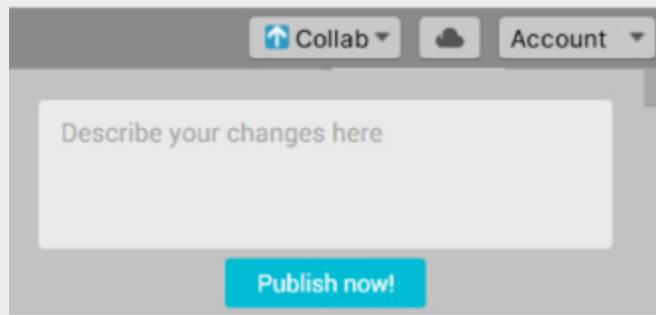
Programming is a skill, and the more practice that you get writing C# code, the better your coding skills will get. That’s why we designed the Unity labs throughout this book to specifically **help you practice your C# skills** and reinforce the C# tools and concepts that you learn in each chapter. As

you write more C# code, you'll get better at it, and that's a really effective way to become a great C# developer. And neuroscience tells us that we learn more effectively when we experiment, so we designed these Unity labs with lots of options for experimentations, and suggestions for how you can get creative and keep going with each lab.

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

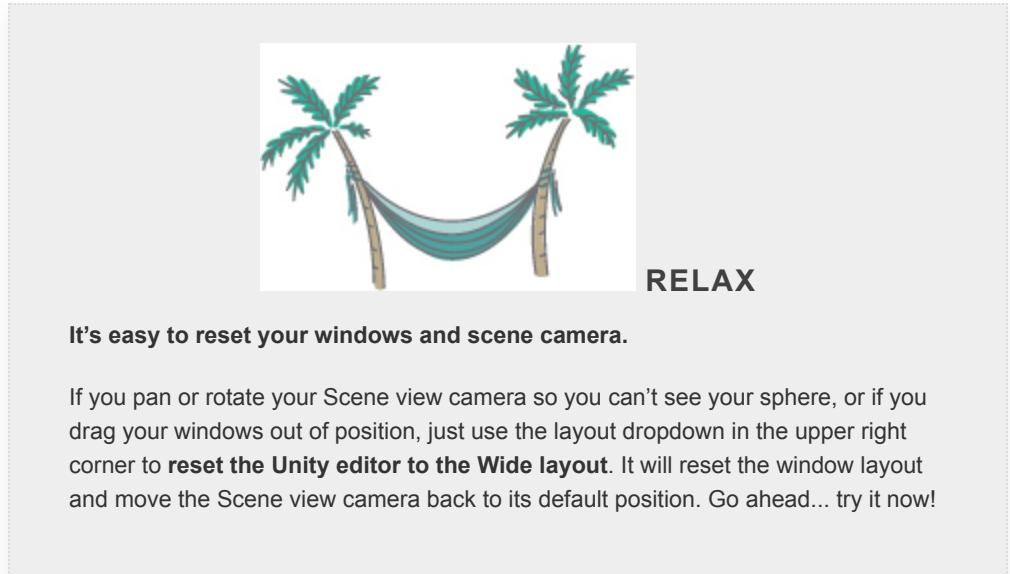
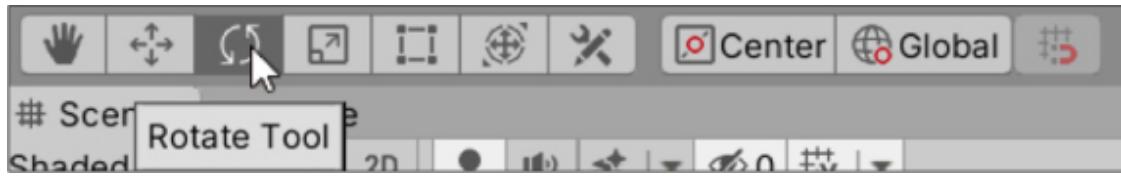
You probably noticed that we haven't talked about adding your Unity projects to source control. That's because we've been using Visual Studio to take care of source control for us, but it doesn't have a way of knowing about exactly which files are part of your Unity project. Luckily, there's an easy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

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

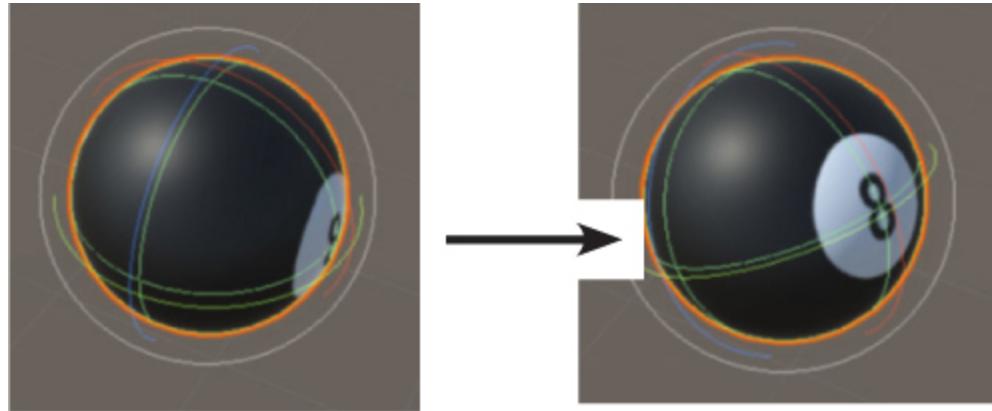


Rotate your sphere

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

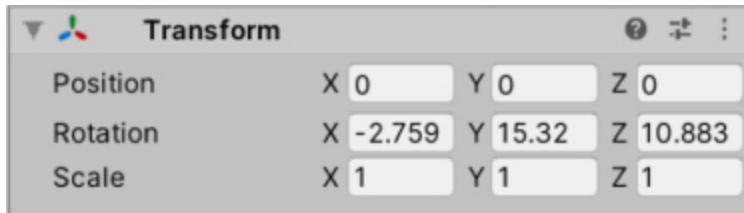


1. **Click on the sphere.** Unity will display a wireframe sphere Rotate Gizmo with red, blue, and green circles. Click the red circle and drag it to rotate the sphere around the X axis.

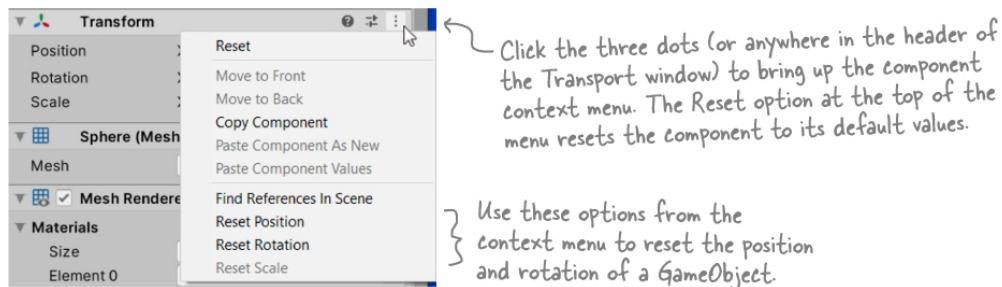


2. **Click and drag the green and blue circles to rotate on the Y and Z axes.** The outer white circle

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



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

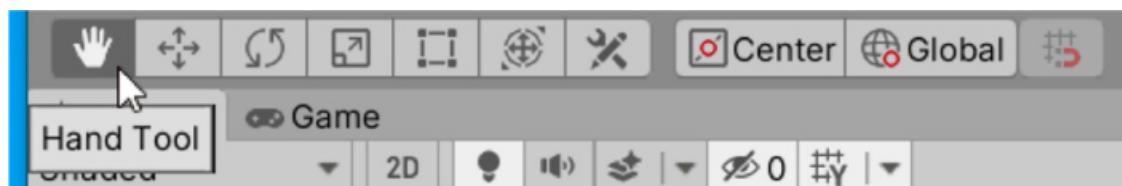


Use File >> Save or Ctrl-S to save the scene right now. Save early, save often!

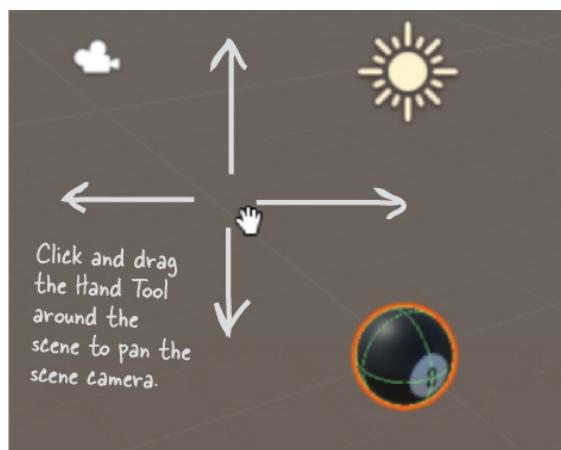
Move the Scene view camera with the Hand Tool and Scene Gizmo

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

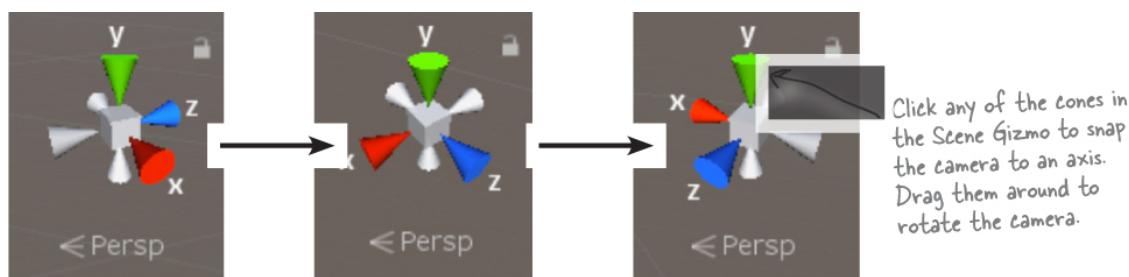
Press Q to select the **Hand Tool**, or choose it from the toolbar. Your cursor will change to a hand.



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



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



THERE ARE NO DUMB QUESTIONS

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

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

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

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

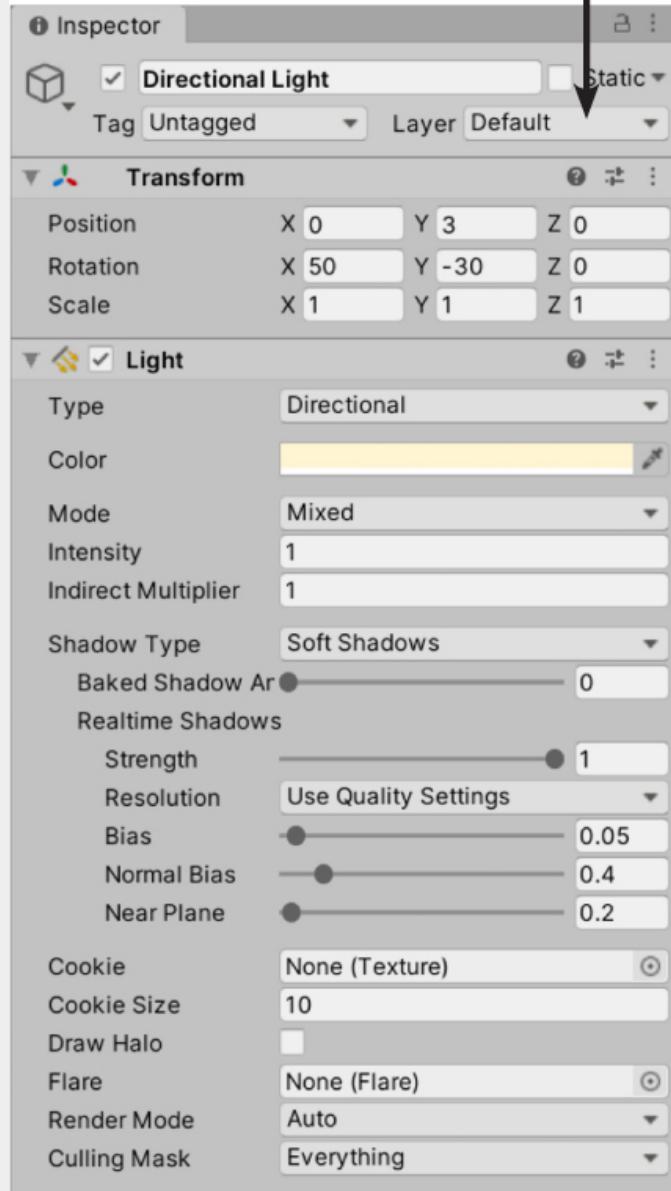
Q: If I add a Light component to any GameObject, does it become a light?

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

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

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

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



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

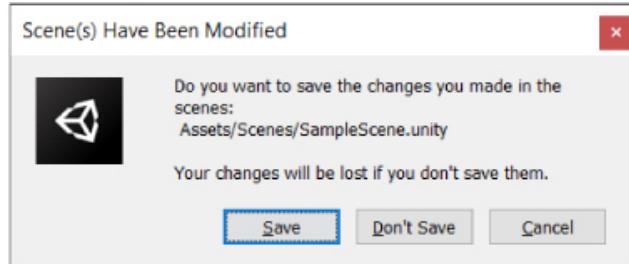
Get creative!

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

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

- Can you create an interesting 3D scene out of shapes, textures, and lights?

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.



NOTE

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



BULLET POINTS

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

Chapter 2. Dive Into C#: Statements, Classes, and Code



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

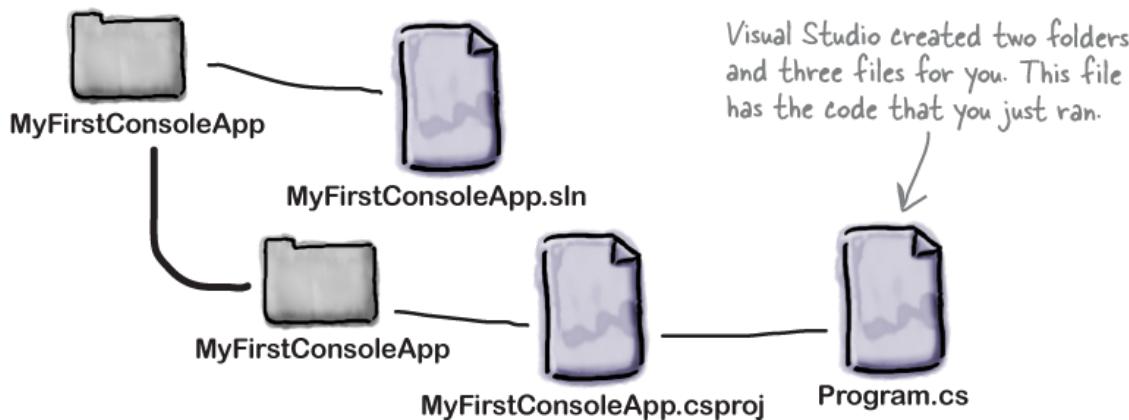
You can get a lot of work done using the IDE, but there's only so far it can take you. Visual Studio is one of the most advanced software development tools ever made, but a **powerful IDE** is

only the beginning. It's time to **dig in to C# code**: how it's structured, how it works, and how you can take control of it... because there's no limit to what you can get your apps to do.

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

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

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



Let's take a closer look at the `Program.cs` file that it created.

Open it up in Visual Studio:

```
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 }
```

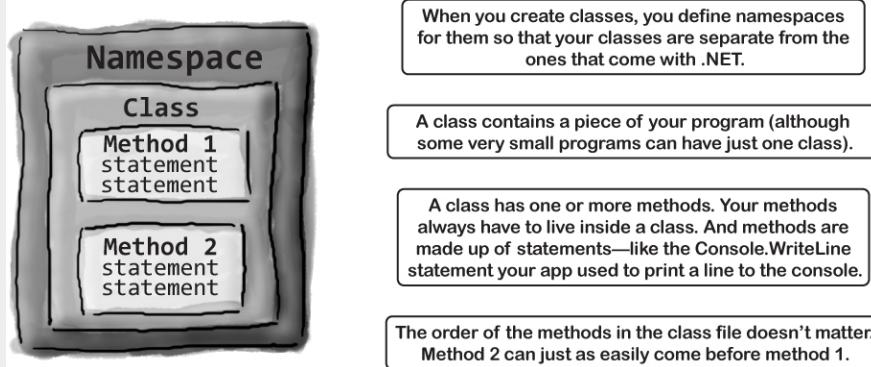
This is a **method** called **Main**. When a **Console App** starts, it looks for a class with a method called **Main**. and starts by executing the first statement in that method. It's called the **entry point** because that's where C# "enters" the program.

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



ANATOMY OF A C# PROGRAM

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



A statement performs one single action

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

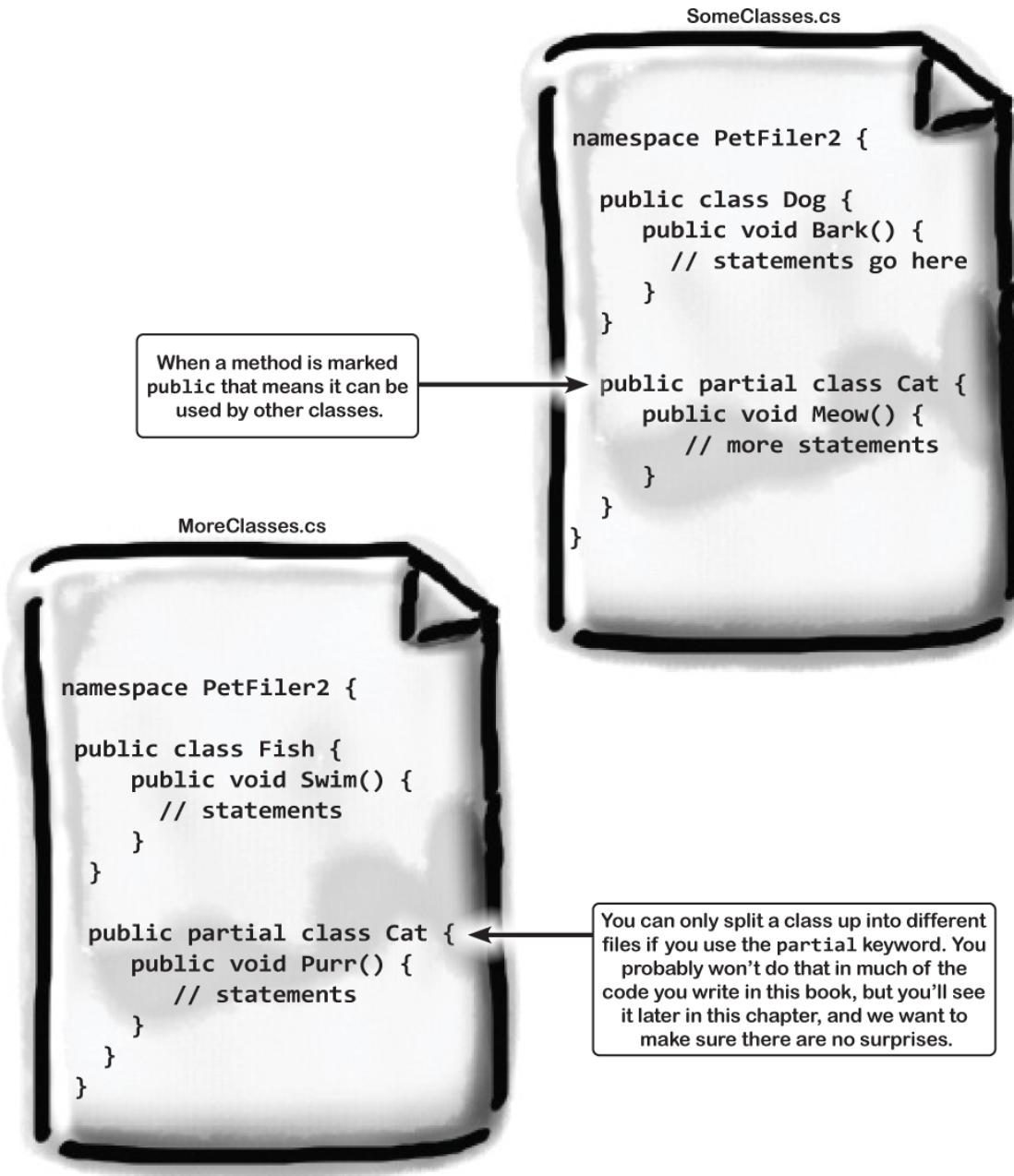
THERE ARE NO DUMB QUESTIONS

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

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

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

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



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.



The IDE helps you build your code right.

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

Over the years, Microsoft (and, let’s be fair, a lot of other companies, and a lot of individual developers) figured out

many helpful things like error highlighting, IntelliSense, WYSIWYG click-and-drag window UI editing, automatic code generation, and many other features.

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

THERE ARE NO DUMB QUESTIONS

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

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

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

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

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

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

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

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

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

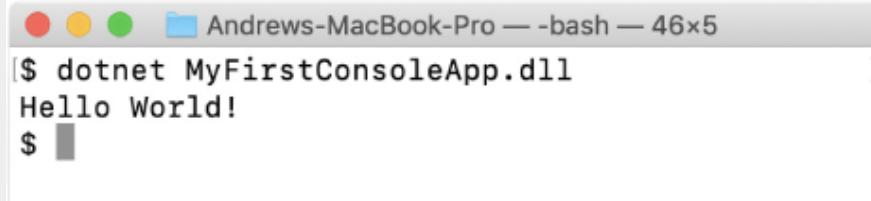
Q: So my console app really runs on Linux and MacOS?

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

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

<https://dotnet.microsoft.com/download>

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



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

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

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

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

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

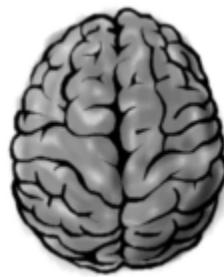
<https://docs.microsoft.com/en-us/dotnet/core/rid-catalog>

Statements are the building blocks for your apps

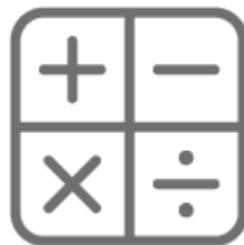
Your app is made up of classes, and those classes contain methods, and those methods contain statements. So if we want to build apps that do a lot of things, we'll need a few **different kinds of statements** to make them work. You've already seen one kind of statement:

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

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



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



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



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



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

Your programs use variables to work with data

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



Declare your variables

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

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

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.

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.

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

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

Variables vary

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

```
int myHeight = 63;
```

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

```
myHeight = 12;
```

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

You need to assign values to variables before you use them

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

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

Go ahead, try it right now. You'll get an error, and the IDE will refuse to build your code. That's because it checks each variable to make sure that you've assigned it a value before you use it.

The easiest way to make sure you don't forget to assign your variables values is to combine the statement that declares a variable with a statement that assigns its value:

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

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

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

NOTE

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

A few useful types

Every variable has a type that tells C# what kind of data it can hold. We'll go into a lot of detail about the many different types in C# in Chapter 4. In the meantime, we'll concentrate on the

three most popular types. `int` holds integers (or whole numbers), `string` holds text, and `bool` holds **Boolean** true/false values.

var-i-a-ble, noun.

an element or feature likely to change.

*Predicting the weather would be a whole lot easier if meteorologists didn't have to take so many **variables** into account.*

Generate a new method to work with variables

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

Do this!

1. **Add a method to your new 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:

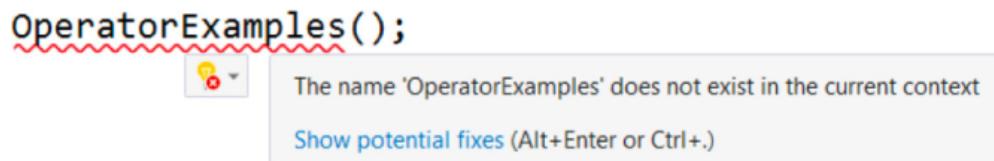
```
Console.WriteLine("Hello World!");
```

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:



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

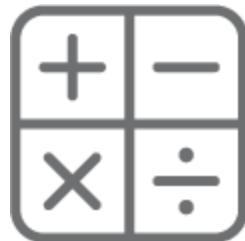
3. Generate the OperatorExamples method.

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



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

Add code that uses operators to your method



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

```

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

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

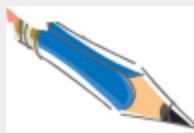
    // Declare two more int variables to hold numbers and
    // use the + and * operators to add and multiply values
    int height = 2 + 4;
    int area = width * height;
    Console.WriteLine(area);

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

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

```

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



MINI SHARPEN YOUR PENCIL

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

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

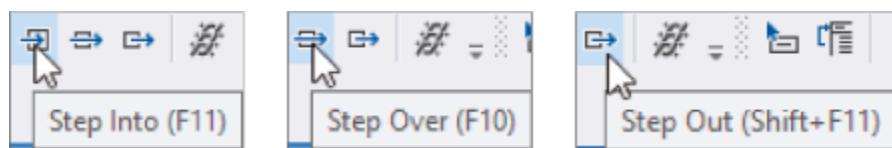
Line 1: _____

Line 2: _____

Line 3: _____

Use the debugger to watch your variables change

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

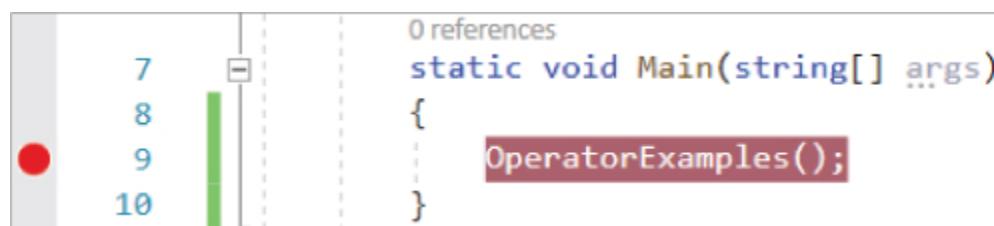


If you end up in a state you don't expect, just use the Restart button (↻) to restart the debugger.

Debug this!

1. Add a breakpoint and run your program.

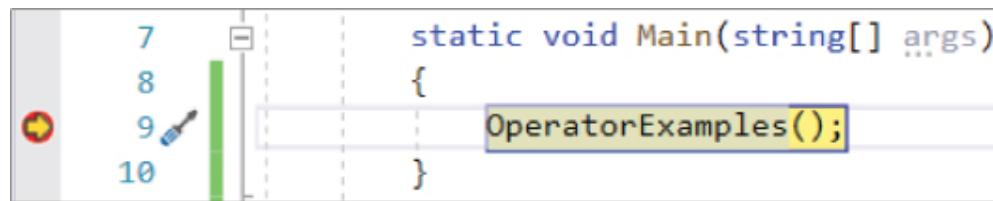
Place your mouse cursor on the method call that you added to your program's Main method and **choose Toggle Breakpoint (F9)** from the Debug menu. The line should now look like this:



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

2. Step into the method.

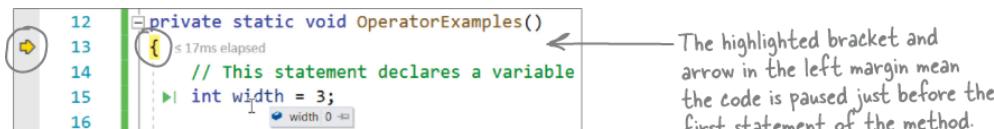
Your debugger is stopped at the breakpoint on the statement that calls the OperatorExamples method.



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

3. Examine the value of the width variable.

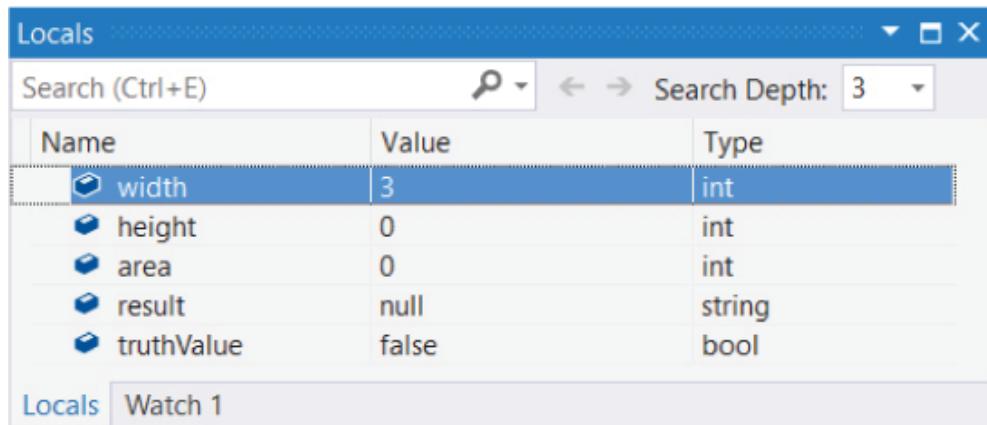
When you're **stepping through your code**, the debugger pauses after each statement that it executes. This gives you the opportunity to examine the values of your variables. Hover over the `width` variable.



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

4. The locals window shows values of your variables.

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

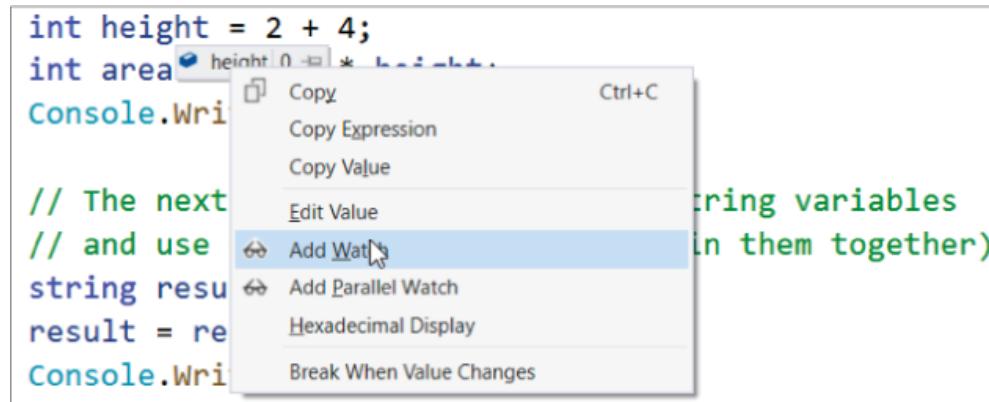


The screenshot shows the Locals window in Visual Studio. It has a header bar with tabs for 'Locals' and 'Watch 1'. Below the header is a search bar labeled 'Search (Ctrl+E)' and a 'Search Depth' dropdown set to 3. The main area is a table with columns 'Name', 'Value', and 'Type'. There are five rows of data:

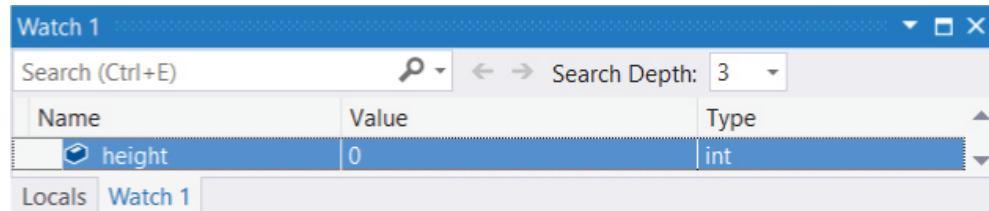
Name	Value	Type
width	3	int
height	0	int
area	0	int
result	null	string
truthValue	false	bool

5. Add a watch for the height variable.

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



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



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

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.

Press Alt-Tab before and after the `Console.WriteLine` statements to switch back and forth to the Debug Console to see the output.

Use operators to work with variables

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

Equality Operators

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

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

Relational Operators

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

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

Logical Operators

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

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



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

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

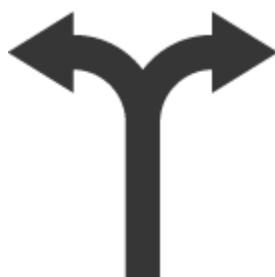
Use operators to compare two int variables

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

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

These are the ones you'll use most often.

if statements make decisions



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

```
int someValue = 10;  
string message = "";  
  
if (someValue == 24)  
{  
    message = "Yes, it's 24!"  
}
```

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

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

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

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

```
if (someValue == 24) 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.";
}
```

Loops perform an action over and over



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

while loops keep looping statements while a condition is true

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

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

do/while loops run the statements then check the condition

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

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

for loops run a statement after each loop

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

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

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

NOTE

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



FOR LOOPS UP CLOSE

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

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

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

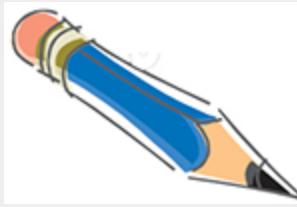
A for loop has three sections: an initializer, a condition, an iterator, and a body:

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

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

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

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



SHARPEN YOUR PENCIL

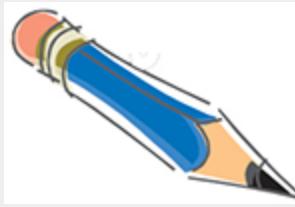
Here are a few loops. Write down if each loop will repeat forever or eventually end. If it's going to end, how many times will it loop? Also, answer the questions in the comments in loops #2 and #3.

```
// Loop #1           // Loop #2           // Loop #3
int count = 5;       int j = 2;         int p = 2;
while (count > 0) {   for (int i = 1; i < 100;   for (int q = 2; q < 32;
    count = count * 3;     i = i * 2)           q = q * 2)
    count = count * -1;   {                     {
}                           j = j - 1;         while (p < q)
}                           while (j < 25)       {
}                           {                     // How many times will
}                           // How many times will
// the next statement
// be executed?
j = j + 5;           }
}                           }
}                           }
}                           }
}                           }
}                           }

// Loop #4           // Loop #5
int i = 0;             while (true) { int i = 1;
int count = 2;           int i = 1;
while (i == 0) {         count = count * 3;
    count = count * 3;   count = count * -1;
    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.

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



SHARPEN YOUR PENCIL SOLUTION

NOTE

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

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           // Loop #2           // Loop #3
int count = 5;       int j = 2;         int p = 2;
while (count > 0) {   for (int i = 1; i < 100;   for (int q = 2; q < 32;
    count = count * 3;     i = i * 2)           q = q * 2)
    count = count * -1;   {                     {
}                           j = j - 1;           while (p < q)
Loop #1 executes once      while (j < 25)           {
                           {                     // How many times will
                           // the next statement
                           // be executed?
                           j = j + 5;           p = p * 2;
                           }                     }
                           }                     q = p - q;
}                           Loop #2 executes 7 times.   Loop #3 executes 8 times.
                           The statement j = j + 5 is   The statement p = p * 2
                           executed 6 times.        executes 3 times.
}
Loop #4 runs forever.   // Loop #5
                        while (true) { int i = 1;}
                        Loop #5 is also an infinite loop.
```

NOTE

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

Use code snippets help to write loops

Do this!

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

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

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

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.

Type `area < 50` – the IDE will replace `true` with the text.

Press Enter to finish the snippet. Then add three statements between the brackets:

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

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:

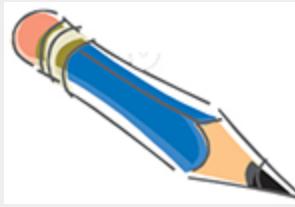
```
do
{
}
```

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

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

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

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



SHARPEN YOUR PENCIL

Let's get some practice working with conditionals and loops. Update the Main method in your console app so it matches the new 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)
{
    OperatorExamples();
    TryAnIf();
    TrySomeLoops();
    TryAnIfElse();
}

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

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

private static void TrySomeLoops()
{
    int count = 0;
    while (count < 10)
    {
        count = count + 1;
    }

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

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

What does the TryAnIf method write to the console?
.....

What does the TryAnIfElse method write to the console?
.....

What does the TrySomeLoops method write to the console?
.....

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

Some useful things to keep in mind about C# code

- **Don't forget that all your statements need to end in a semicolon.**

```
name = "Joe";
```

- **Add comments to your code by starting a line with two slashes.**

```
// this text is ignored
```

- Use /* and */ to start and end comments that can include line breaks.

```
/* this comment  
 * spans multiple lines */
```

- **Variables are always declared with a *name* and a *type*.**

```
int weight;  
// the name is weight and the type  
is int
```

- **Most of the time, extra whitespace is fine.**

```
So this:           int          j =  
1234            ;
```

```
Is exactly the same as this: int j  
= 1234;
```

- **If/else, while, do, and for are all about testing conditions.**

Every loop we've seen so far keeps running as long as a condition is true.



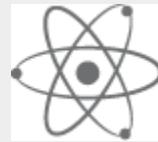
Then your loop runs forever.

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

keep running until you kill the program or turn the computer off!

NOTE

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



BRAIN POWER

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



GAME DESIGN... AND BEYOND MECHANICS

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

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



Definitely! Every program has its own kind of mechanics.

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

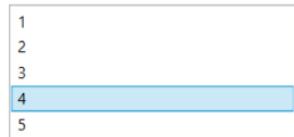
Here's an example. The mechanics of a game determine how hard or easy it is to play. Make Pac Man faster or the ghosts slower and the game gets easier. That doesn't necessarily make it better or worse—just different. And guess what? The same exact idea applies to how you design your classes! You can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

Controls drive the mechanics of your user interfaces

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

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

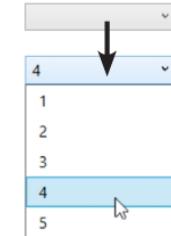
- ★ **Radio buttons** let you restrict the user's choice. You can use them for numbers if you want, and you can choose how you want to lay them out.



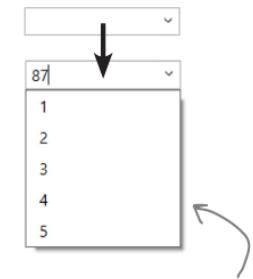
- ★ The other controls on this page can be used for other types of data, but **sliders** are used exclusively to choose a number. Phone numbers are just numbers, too. So *technically* you could use a slider to choose a phone number. Do you think that's a good choice?

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.

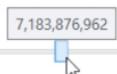
We can borrow the idea of mechanics from video games to understand our options, so we can make great choices for any of our own apps—not just games.



- ★ 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.

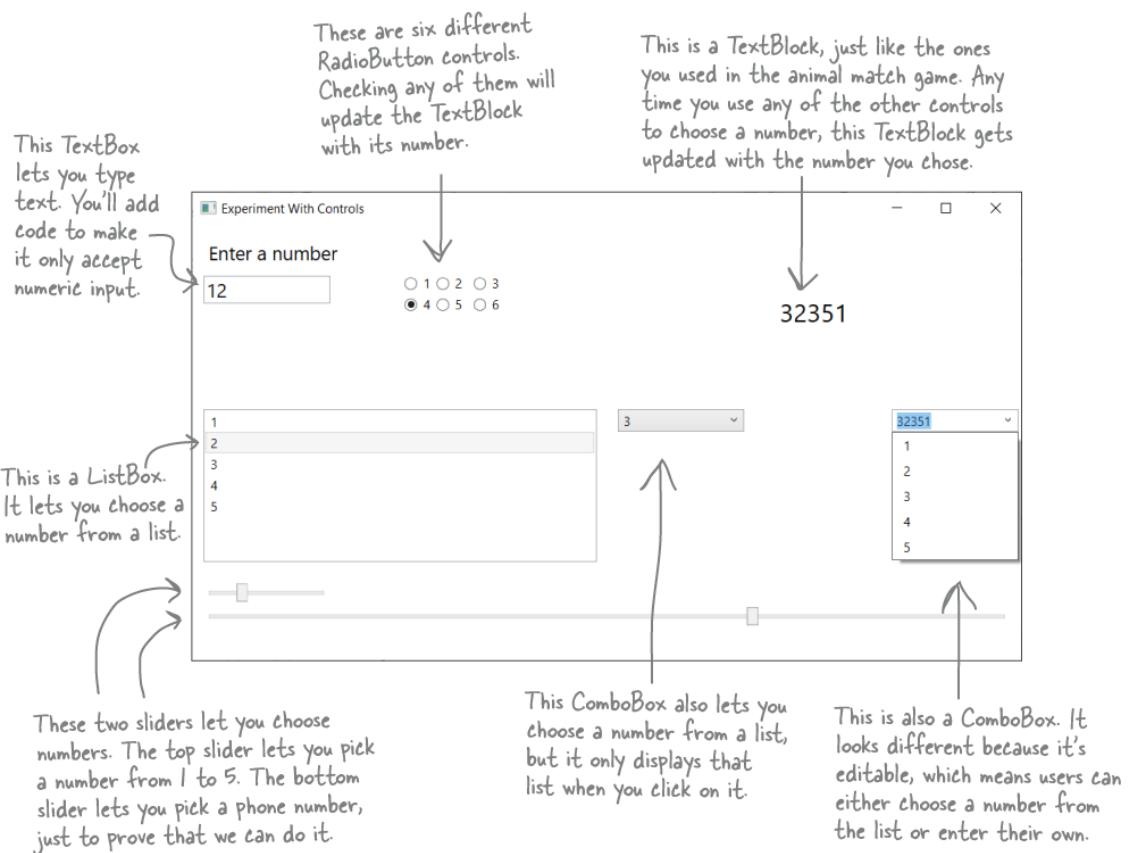


Editable check boxes let the user either choose from a list of items or type in their own value.



Create a WPF app to experiment with controls

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





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.



EXERCISE

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

Create a new WPF project

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



Name your project **ExperimentWithControls**.

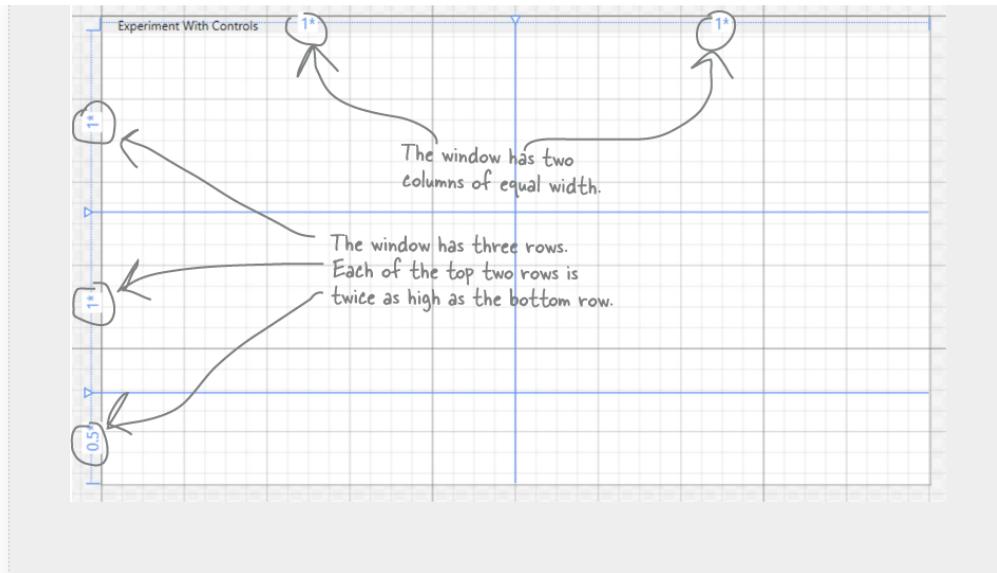
Set the window title

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

Add the rows and columns

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

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



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.

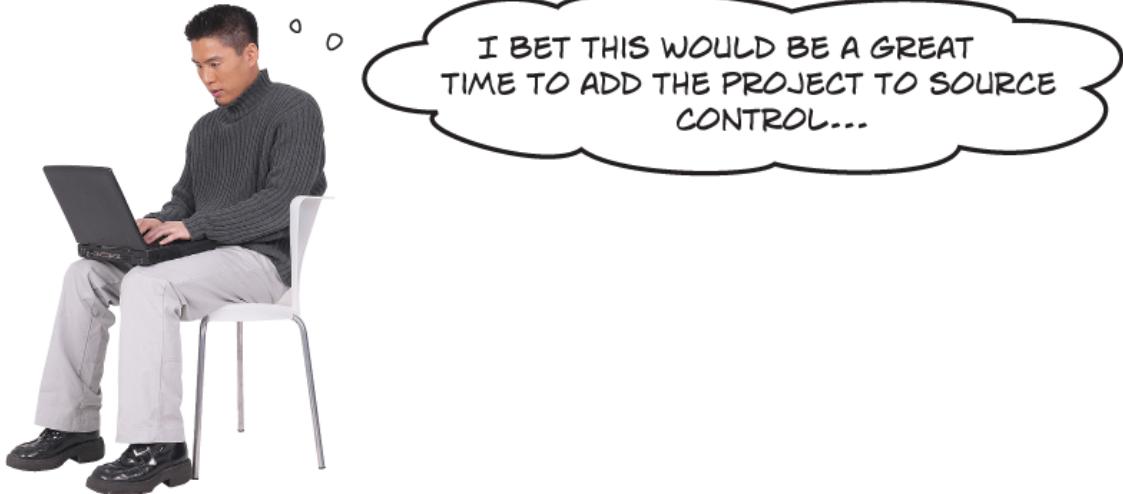
```

<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.)



“Save early, save often.”

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



Add a TextBox control to your app

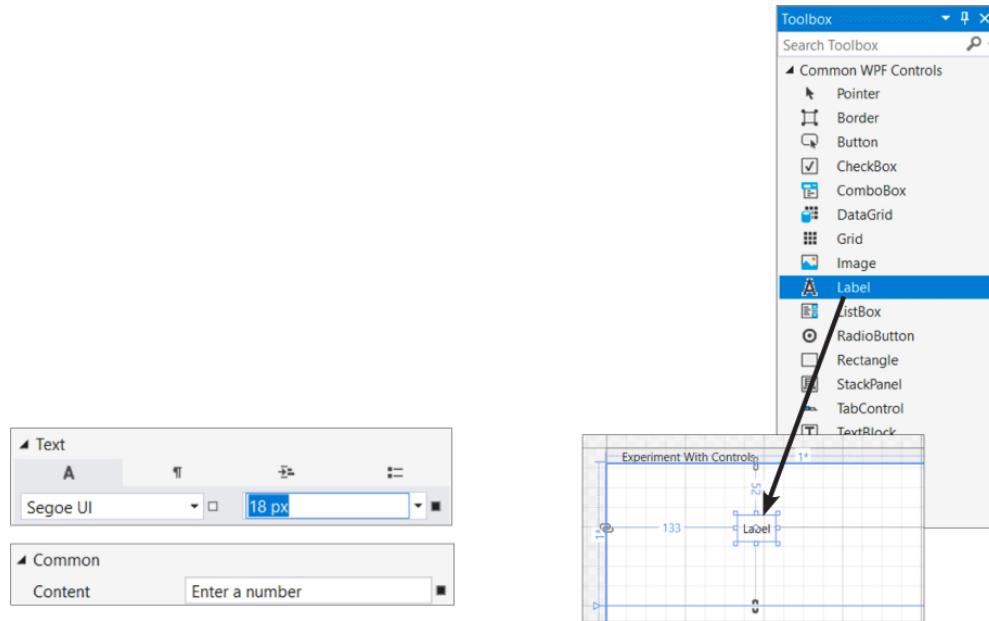
A **TextBox control** gives your user a box to type text into, so let's add one to your app. But we don't just want a TextBox sitting there without a label, so we'll use a **Label control** (which is a lot like a TextBlock, except it's specifically used to add labels to other controls).

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

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

- 2. Set the text size and content of the Label.**

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

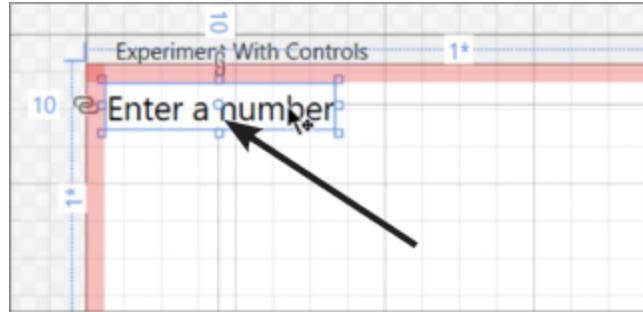


3. Drag the Label to the upper left corner of the cell.

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

The XAML for your window should now contain Label control:

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



MINI EXERCISE

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





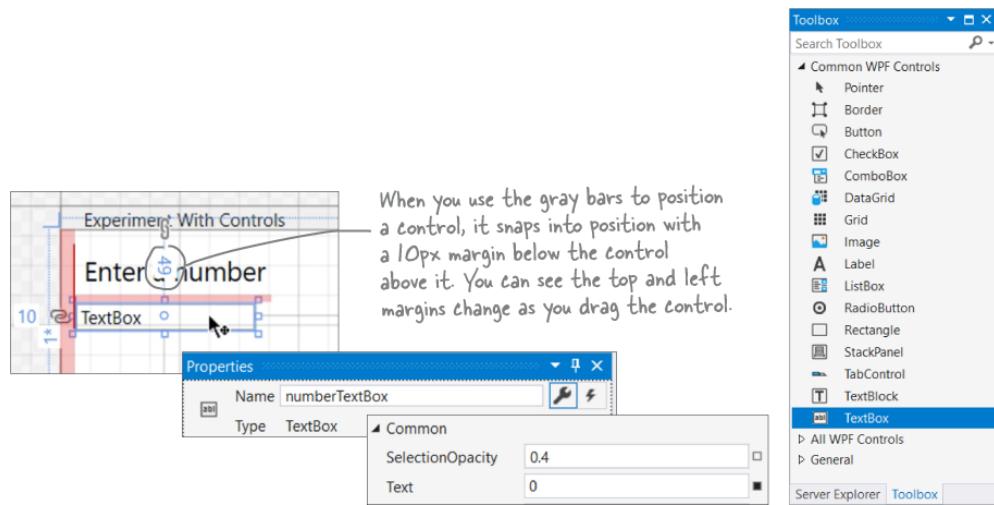
MINI EXERCISE SOLUTION

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

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

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

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



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 okay if your properties are in a different order or if there are line breaks.

Add C# code to update the TextBlock

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

NOTE

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

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

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

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

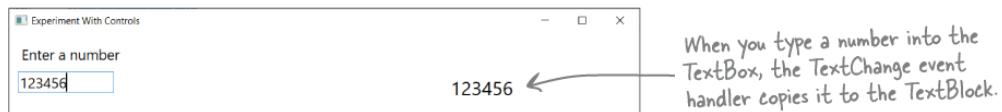
6. Add code to the new TextChanged event handler.

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

```
private void numberTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    number.Text = numberTextBox.Text; ← This line of code sets the text in the TextBlock so it's
}                                         the same as the text in the TextBox, and it gets called
                                                any time the user changes the text in the TextBox.
```

7. Run your app and try out the TextBox.

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



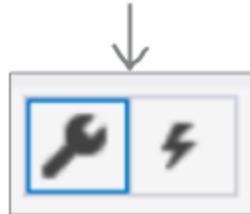
But something's wrong—you can enter any text into the TextBox, not just numbers!



Add an event handler that only allows number input

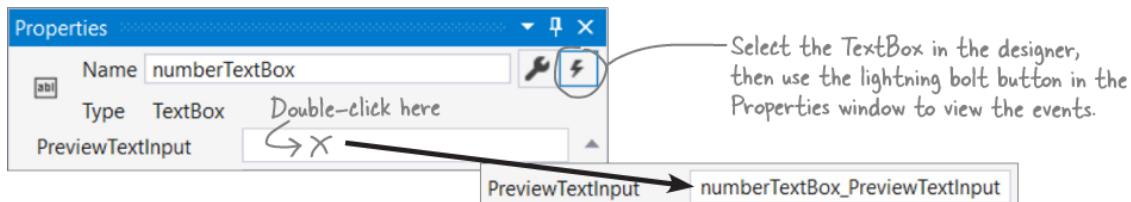
When you added the `MouseDown` event to your `TextBlock` in Chapter 1, you used the buttons in the upper right corner of the Properties window to switch between properties and events. Now you'll do the same thing, except this time you'll use the **PreviewTextInput** event to only accept input that's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

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



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

Do this!



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

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

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

Here's how this event handler works:

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

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

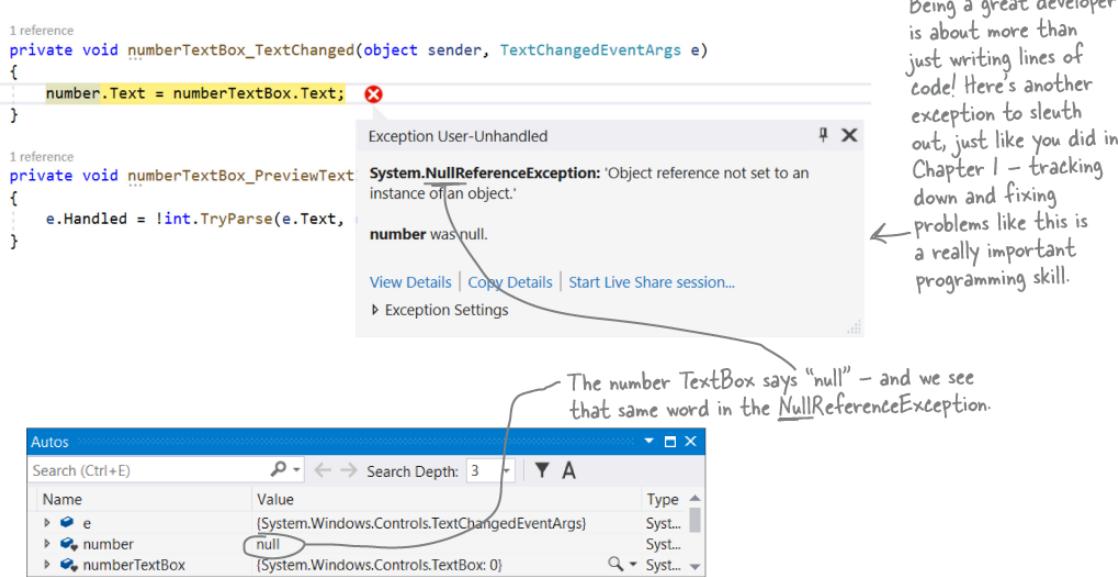
```
<TextBox x:Name="numberTextBox" FontSize="18"
Margin="10,49,0,0" Text="0" Width="120"
    HorizontalAlignment="Left"
    TextWrapping="Wrap" VerticalAlignment="Top"
    TextChanged="numberTextBox_TextChanged"

    PreviewTextInput="numberTextBox_PreviewTextInput"/>
```

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

PreviewTextInput event is hooked up to a method called numberTextBox_PreviewTextInput.

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



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

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



SLEUTH IT OUT

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

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

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

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

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" ... />← Select the TextBlock tag in the
<TextBox x:Name="numberTextBox" ... />           XAML editor move it above the
                                                       TextBox so it gets initialized first.
```

NOTE

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.

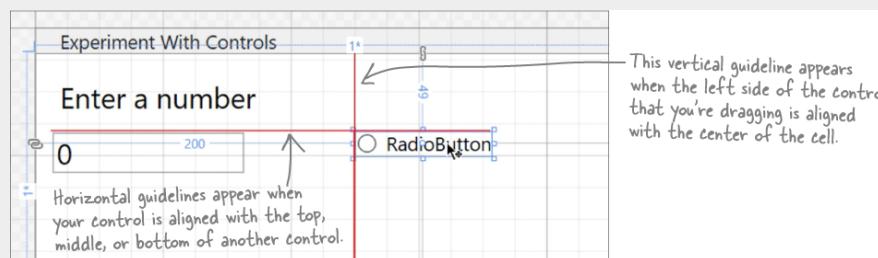


EXERCISE

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

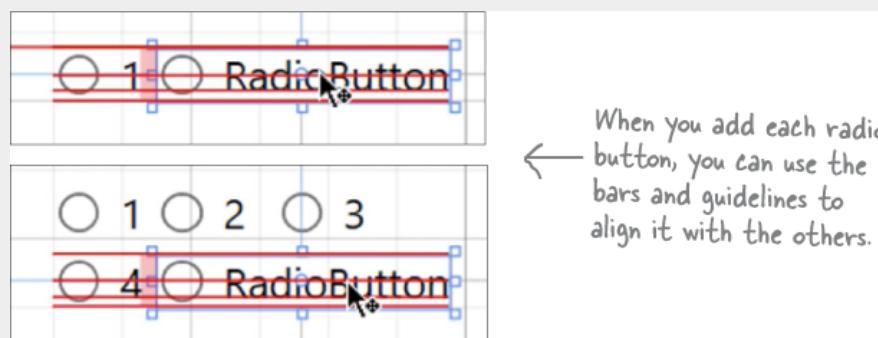
Add radio buttons to the upper left cell next to the TextBox

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



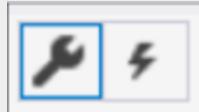
Expand the Common section of the Properties window and set the Content of the RadioButton control to 1.

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



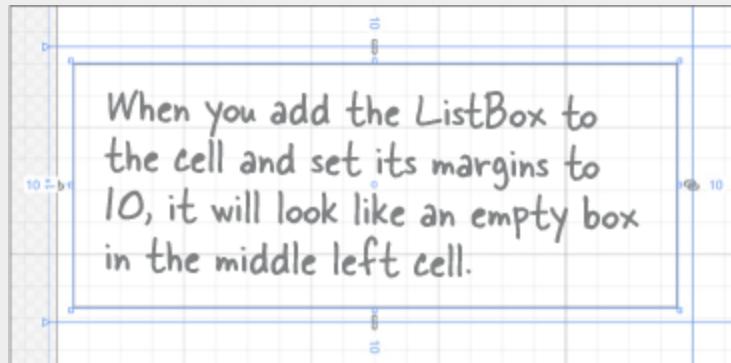
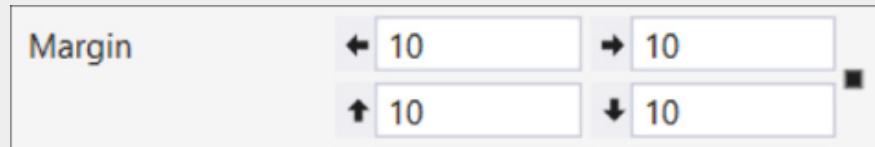
NOTE

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



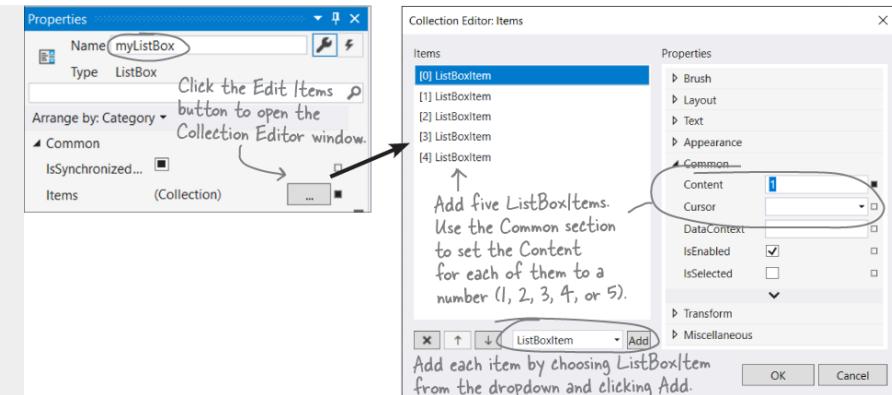
Add a list box to the middle left cell of the grid

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

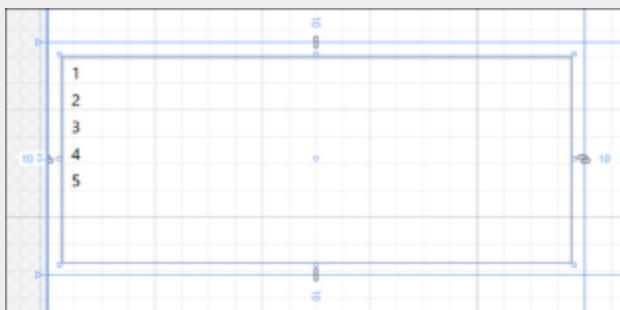


Name your ListBox myListBox and add ListBoxItems to it

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



Your ListBox should now look like this:



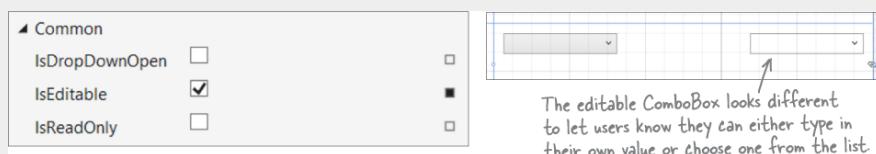
Add two different ComboBoxes to the middle right cell in the grid

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



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

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





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 them just above the closing `</Grid>` tag. And just like with any other XAML you've seen so far, it's okay if the properties for a tag are in a different order in your code, or if you have different line breaks.

```
<RadioButton Content="1" Margin="200,49,0,0"
            HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="2" Margin="230,49,0,0"
            HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="3" Margin="265,49,0,0"
            HorizontalAlignment="Left" VerticalAlignment="Top"/> ← The IDE added the
<RadioButton Content="4" Margin="200,69,0,0"
            HorizontalAlignment="Left" VerticalAlignment="Top"/> margin and alignment
<RadioButton Content="5" Margin="230,69,0,0"
            HorizontalAlignment="Left" VerticalAlignment="Top"/> properties to each
<RadioButton Content="6" Margin="265,69,0,0"
            HorizontalAlignment="Left" VerticalAlignment="Top"/> RadioButton control
                                         when you dragged it
                                         into place.

<ListBox x:Name="myListBox" Grid.Row="1" Margin="10,10,10,10">
    <ListBoxItem Content="1"/>
    <ListBoxItem Content="2"/>
    <ListBoxItem Content="3"/>
    <ListBoxItem Content="4"/>
    <ListBoxItem Content="5"/> } 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 give
                                         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 → When you run your program, it should look like this. You can use all of the controls, but only the TextBox actually updates the value in the upper right.
```

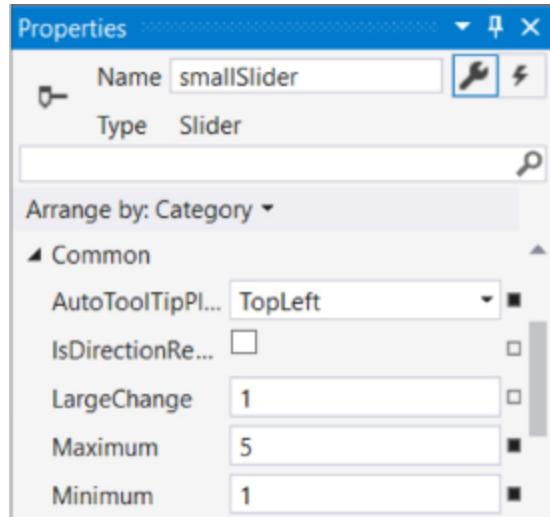
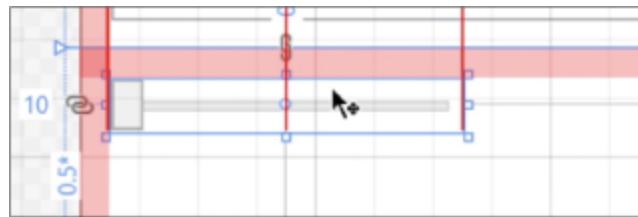


Add sliders to the bottom row of the grid

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

1. Add a slider to your app.

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



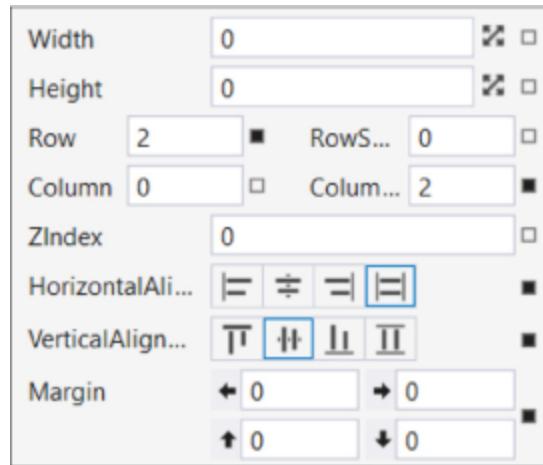
Use the Common section of the Properties window to set AutoToolTipPlacement to **TopLeft**, Maximum to **5**, Minimum to **1**, and give it the name **smallSlider**. Then double-click on the slider to add this event handler:

```
private void smallSlider_ValueChanged(  
    object sender, RoutedPropertyChangedEventArgs<double> e)  
{  
    number.Text = smallSlider.Value.ToString("0");  
}  
The value of the Slider control is fractional number with  
a decimal point. This "0" converts it to a whole number.
```

2. Add a ridiculous slider to choose phone numbers.

There's an old saying: "*Just because an idea is terrible and also maybe stupid, that doesn't mean you shouldn't do it.*" So let's do something that's just a bit stupid: add a slider to select phone numbers.

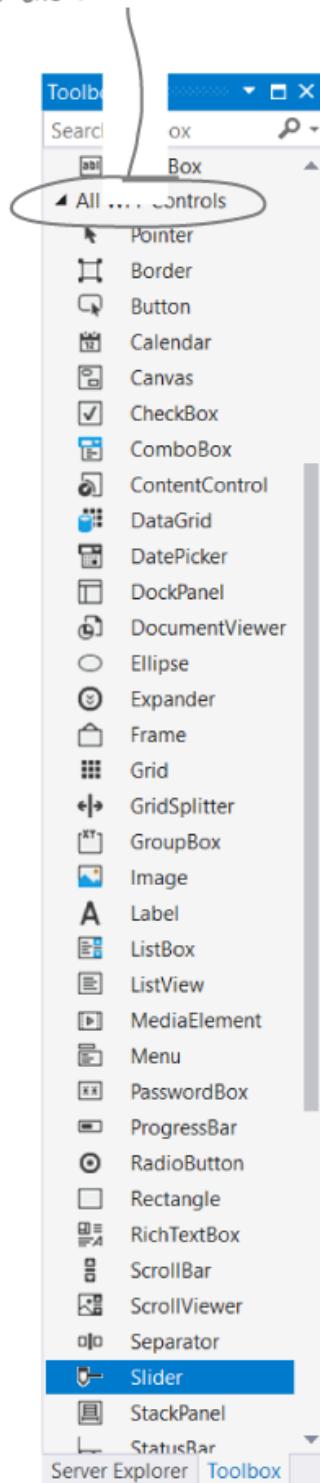
Drag another slider into the bottom row. Use the Layout section of the Properties window to **reset its width**, set its ColumnSpan to **2**, set all of its margins to **10**, and set its vertical alignment to **center** and horizontal alignment to **stretch**. Then use the Common section to set AutoToolTipPlacement to **TopLeft**, Minimum **1111111111**, Maximum **9999999999**, and value **7183876962**. Give it the name **bigSlider**. Then double-click on it and add this ValueChanged event handler:



```
private void bigSlider_ValueChanged(  
    object sender, RoutedPropertyChangedEventArgs<double> e)  
{  
    number.Text = bigSlider.Value.ToString("000-000-0000");  
}
```

The zeroes and hyphens causes the
method to format any 10-digit
number as a US phone number.
you are here >

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



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

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

1. Add a Checked event handler to the first RadioButton control.

Double-click on the first RadioButton. The IDE will add a new event handler method called RadioButton_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();  
}
```



Ready Bake Code

This statement uses the `is` keyword, which you'll learn about in Chapter 7. For now, just carefully enter it exactly like it appears on the page (and do the same for the other event handler method, too).

2. Make the other RadioButtons use the same event handler.

Look closely at the XAML for the RadioButton that you just modified. The IDE added this property:

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



If you switch the Properties window to the Events view, you can select any of the RadioButton controls and make sure they all have the Checked event hooked up to the RadioButton_Checked event handler.

3. Make the ListBox update the TextBlock in the upper right cell.

When you did the exercise, you named your `ListBox` `myListBox`. Now you'll add an event handler that fires any time the user selects an item and uses the name to get the number that the user selected.

Double-click inside the empty space in the `ListBox` below the items to make the IDE will add an event handler method for the `SelectionChanged` event. Add this statement to it:

NOTE

Make sure you click on the empty space below the list items. If you click on an item, it will add an event handler for that item and not for the entire `ListBox`. You can also use the Properties window to add a `SelectionChanged` event

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

4. Make the read-only combo box update the TextBlock.

Double-click on the read-only ComboBox to make Visual Studio add an event handler for the SelectionChanged event, which is raised any time a new item is selected in the ComboBox. Here's the code—it's really similar to the code for the ListBox:

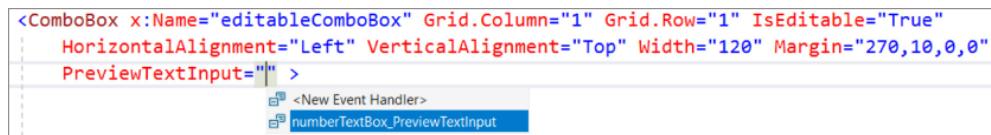
```
private void  
readOnlyComboBox_SelectionChanged(  
    object sender,  
    SelectionChangedEventArgs e)  
{  
    if  
(readOnlyComboBox.SelectedItem is  
    ListBoxItem listBoxItem)  
        number.Text =  
        listBoxItem.Content.ToString();  
}
```

5. Make the editable combo box update the TextBlock.

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

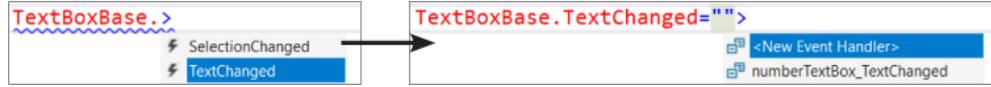
numbers, just like we did with the TextBox. In fact, you even **reuse the same event handler** that you already added for the TextBox.

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



The previous event handlers used the list items to update the TextBlock. But users can enter any text they want into an editable ComboBox, so this time you'll **add a different kind of event handler**.

Edit the XAML again. This time, **type TextBoxBase**.—as soon as you type the period, the autocomplete will give suggestions. Choose **TextBoxBase.TextChanged** and type an equals sign. Now choose < New Event Handler> from the dropdown.



The IDE will add a new event handler to the code-behind. Here's the code for it:

```
private void  
editableComboBox_TextChanged(object sender, TextChangedEventArgs e)  
{  
    if (sender is ComboBox  
comboBox)  
        number.Text =  
comboBox.Text;  
}
```

Now run your program. All of the controls should work. Great job!



Controls give you the flexibility to make things easy for your users.

When you're building the UI for an app, there are so many choices that you make: what controls to use, where to put each one, what to do with their input. Picking one control instead of another gives your users an *implicit* message about how to use your app. For example, when you see a set of radio buttons, you know that you need to pick from a small set of choices, while an editable combo box tells you that there your choices are nearly unlimited. So don't think of UI design as a matter of making "right" or "wrong" choices. Instead, think of it as your way to make things as easy as possible for your users.



BULLET POINTS

- C# programs are organized into **classes**, classes contain **methods**, and methods contain **statements**.
- Each class belongs to a **namespace**. Some namespaces (like System.Collections.Generic) contain .NET classes.
- Classes can contain **fields**, which live outside of methods. Different methods can access the same field.
- When a method is marked **public** that means it can be called from other classes.
- **.NET Core console apps** are cross-platform programs that don't have a graphical user interface.
- The IDE **builds** your code to turn it into a **binary**, which is a file that can be executed.
- If you have a cross-platform .NET Core console app, you can use the **dotnet** command line program to **build binaries** for different operating systems.
- The **Console.WriteLine** method writes a string to the console output.
- Variables need to be **declared** before they can be used. You can set a variable's value at the same time.
- The Visual Studio debugger lets you **pause your app** and inspect the value of variables.
- Controls **raise events** for lots of different things that change: mouse clicks, selection changes, text entry. Sometimes people say events are **triggered** or **fired**, which is the same as saying that they're raised.
- **Event handlers** are methods that are called when an event is raised to respond to—or **handle**—the event.
- TextBox controls can use the **PreviewTextInput** event to accept or reject text input.
- A **slider** is a great way to get number input, but a terrible way to choose a phone number.

Part II. 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.**

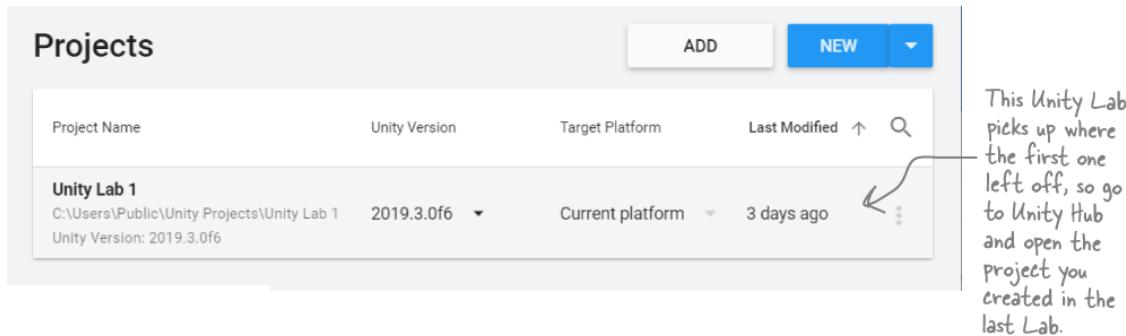
In the last Unity Lab you learned how to navigate around Unity and your 3D space, and started to create and explore GameObjects. Now it's time to write some code to take control of your GameObjects. The whole goal of that Lab was to get you oriented in the Unity editor (and give you an easy way to remind 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, and. And you'll start using the Visual Studio debugger with Unity to sleuth out problems in your games.

C# scripts add behavior to your GameObjects

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

This Unity lab will introduce tools that you'll use to work with C# and Unity. You're going to build a simple “game” that's really just a little bit of visual eye candy: you'll make your 8 ball fly around the scene. Start by going to Unity Hub and **opening the same project** that you created in the first Unity Lab.



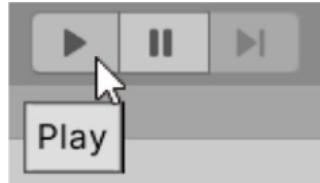
Here's what you'll do next:

1. **Attach a C# script to your GameObject.** You'll add a Script component to your Sphere GameObject. When

you add it, Unity will create a simple class for you. You'll modify that class so that it drives the 8 ball sphere's behavior.

2. Use Visual Studio to edit the script. Remember how you set the Unity editor preferences to make Visual Studio the script editor? That means you can just double-click on the script in the Unity editor and it will open up in Visual Studio.

3. Play your game in Unity. There's a Play button at the top of the screen. When you press it, it starts executing all of the scripts attached to the GameObjects in your scene. You'll use that button to run the script that you added to the Sphere.



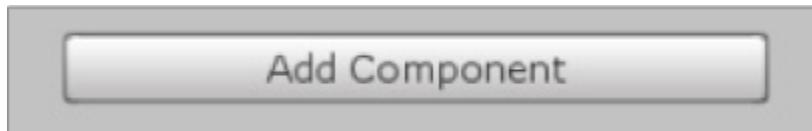
← The Play button does not save your game! So make sure you save early and save often. A lot of people get in the habit of saving the scene every time they run the game.

4. Use Unity and Visual Studio together to debug your script. You've already seen how valuable the Visual Studio debugger is when you're trying to track down problems in your C# code. That's why Unity and Visual Studio work together seamlessly to let add breakpoints, use the Locals window, and work with the other familiar tools in the Visual Studio debugger while your game is running.

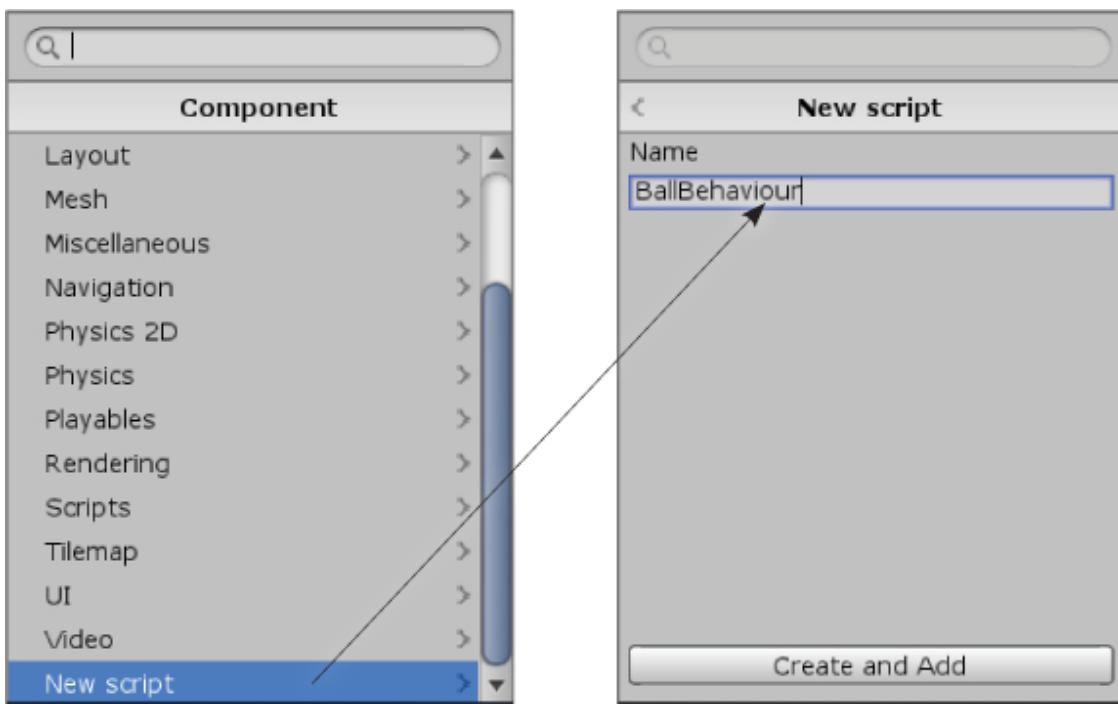
Add a C# script to your GameObject

Unity is more than an amazing platform for building 2D and 3D games. Many people use it for artistic work, data visualization, augmented reality, and more. But it's especially valuable to you, as a C# learner, because you can write code to control everything that you see in a Unity game. That makes Unity ***a great tool for learning and exploring C#.***

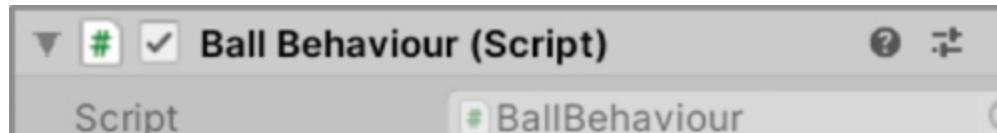
Let's start using C# and Unity right now. Make sure the Sphere GameObject is selected, then **click the Add Component button** at the bottom of the Inspector window.



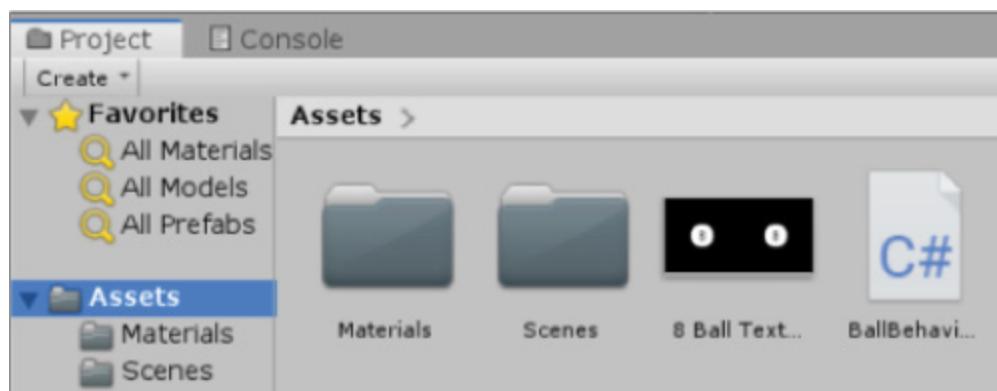
When you click it, Unity pops up a window with all of the different kinds of components that you can add—and there are ***a lot*** of them. **Choose New Script** to add a new C# Script to your Sphere GameObject. You'll be prompted for a name. **Name your script BallBehaviour.**



Click the Create and Add button to add the script. Once you do, you'll see a component called *Ball Behaviour (Script)* in the Inspector window.



You'll also see the C# script in the Project window.





Watch it!

Unity code uses British spelling.

*If you're American (like us), or if you're used to the U.S. spelling of the word **behavior**, you'll need to be careful when you work with Unity scripts because the class names often feature the British spelling **behaviour**.*

NOTE

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.

NOTE

Did you notice a folder called Materials appeared in the Project window as soon as you dragged the 8 ball texture onto your sphere?

Write C# code to rotate your sphere

At the beginning of this Lab you told Unity to use Visual Studio as its external script editor. So go ahead and **double-click on your new C# script**. When you do, ***Unity will open your script in Visual Studio***. Your C# script contains a class called BallBehaviour with two empty methods called Start and Update.

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

public class BallBehaviour : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

NOTE

The Hierarchy window shows you a list of every GameObject in the current scene. When Unity created your project, it added a scene called SampleScene with a camera and a light. You added a sphere to it, so your Hierarchy window will show all of those things.

Here's a line of code that will rotate your sphere. Add it to your Update method:

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

Now go back to the Unity editor and click the play button in the toolbar to start your game:





YOUR CODE UP CLOSE

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

public class BallBehaviour : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        // Update is called once per frame
        void Update()
        {
            transform.Rotate(Vector3.up, 180 * Time.deltaTime);
        }
    }
}
```

When Unity created the file with the C# script, it added using lines at the top so it can use code in the UnityEngine namespace and other commonly used namespaces.

A frame is a fundamental concept of animation. Unity draws one still frame, then very quickly draws the next one very quickly, and your eye interprets these frames as movement. Unity calls the Update method before for every GameObject before each frame so it can move, rotate, or make any other changes that it needs to make. A faster computer will run at a higher frame rate—or number of frames per second (FPS)—than a slower one.

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

Inside your Update method, multiplying any value by Time.deltaTime turns it into that value per second.

Different computers will run your game at different frame rates. If it's running at 30 FPS, we want one rotation every 60 frames. If it's running at 120 FPS, it should rotate once every 240 frames. Your game's frame rate may even change if it needs to run more or less complex code.

That's where the special Time.deltaTime value comes in handy. Every time the Unity engine calls a GameObject's Update method—once per frame—it sets deltaTime to the fraction of a second since the last frame. Since we want our ball to do a full rotation every two seconds, or 180 degrees per second, all we need to do is multiply it by Time.deltaTime to make sure that it rotates exactly as much as it needs to for that frame.

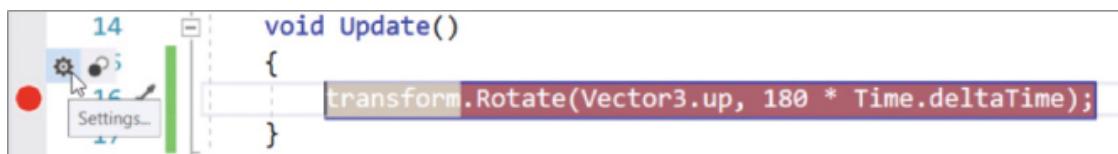
Add a breakpoint and debug your game

You can **attach Visual Studio to Unity** to debug your game in Visual Studio. Your code is running inside the Unity editor, and when Visual Studio is attached to another process running on your computer, attaching to that process lets you use the debugger to place breakpoints and watch variables *on the running program*. Let's explore how that works.

Find the debug button in the Visual Studio toolbar—it now says “Attach to Unity” (▶ **Attach to Unity** ▾). When you’re editing a Unity project, **Visual Studio will play the game in Unity when you start the debugger**.

Start the debugger—press the debug button or choose Start Debugging (F5) from the Debug menu, just like before. Since your game is already running, Visual Studio will attach to the game currently in progress.

Next, **add a breakpoint** on the line that you added to the Update method.

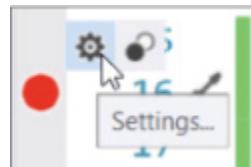


Since Visual Studio is attached to Unity, when you add the breakpoint it **breaks immediately**.

NOTE

Congratulations, you're now debugging a game!

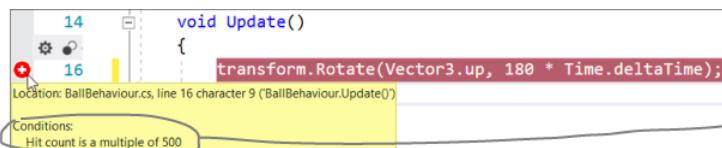
Next, you'll set a **Hit Count condition** to make your breakpoint stop only after it's been run 500 times, then skip another 500 frames before stopping. Hover over the red breakpoint dot in the left margin of the code editor. You'll see a gear for settings, and dots to enable and disable the breakpoint.



Click on the Settings gear to bring up the breakpoint settings. **Check the Conditions box**, choose **Hit Count**, and make the breakpoint trigger when the hit count **is a multiple of 500**.



When you close the settings window, a white plus appears inside the breakpoint dot (⊕) to show that it has a condition.



When your breakpoint has conditions, hover over it to see a tooltip window that describes those conditions.

Now the breakpoint will only pause the game every 500 times the Update method is run—or every 500 frames. So if your game is running at 60FPS, that means when you press Continue, the game will run for little over 8 seconds before it breaks again. So **press Continue, then switch back to Unity** and watch the ball spin until the breakpoint breaks.



Watch it!

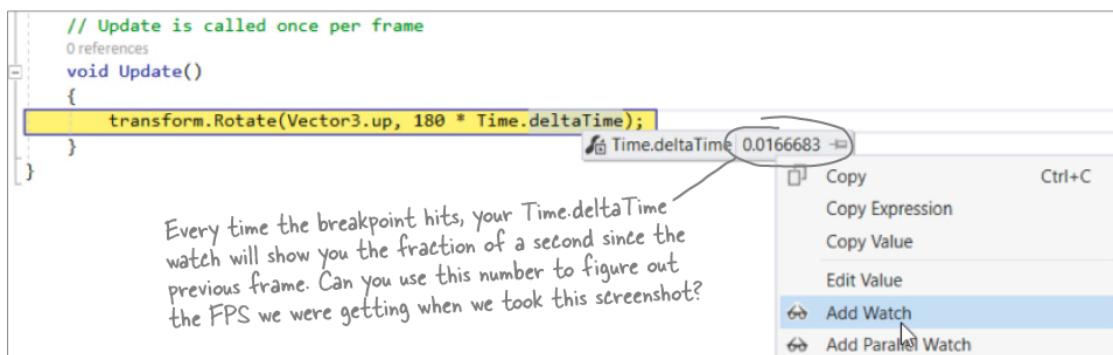
Having trouble getting the Visual Studio debugger to attach to Unity?

*Unity and Visual Studio are built to work together! But you need to do a little configuration to make sure it works. First, make sure you followed the instructions at the start of the first Unity Lab to **set the external script editor** in Unity, and make sure the **Editor Attaching checkbox is checked**. If you still don't see “Attach to Unity and Play” when Unity opens the script in Visual Studio, go to the Individual Tools section in the Visual Studio installer and make sure Visual Studio Tools for Unity is installed (Windows only). And a few users have found that unchecking Editor Attaching, restarting Unity, then re-checking it and restarting again can help fix it.*

Use the debugger to understand time.deltaTime

You're going to be using Time.deltaTime in many of the Unity Labs projects. Let's take advantage of your breakpoint and use the debugger to really understand what's going on with time.deltaTime.

While your game is paused on the breakpoint in Visual Studio, **hover over Time.deltaTime** to see the fraction of a second that elapsed since the previous frame (you'll need to put your mouse cursor over deltaTime). Then **add a watch for time.deltaTime** by clicking on the value and choosing Add Watch from the pop-up menu.



Click Continue (use the button or choose Continue (F5) from the Debug menu) in Visual Studio to resume your game. The ball will start rotating again, and after

another 500 frames the breakpoint will trigger again. You can keep running the game for 500 frames at a time. Keep your eye on the Watch window each time it breaks.

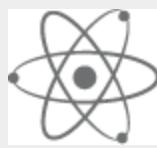


Press Stop debugging (F5) to stop your program. Then start debugging again—since your game is still running, the breakpoint will continue to work. Once you're done debugging, **toggle your breakpoint again to delete it** so the IDE will still keep track of it but not break when it's hit. **Stop debugging** one more time to detach from Unity.

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

NOTE

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



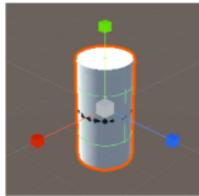
BRAIN POWER

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

Add a cylinder to show where the Y axis is

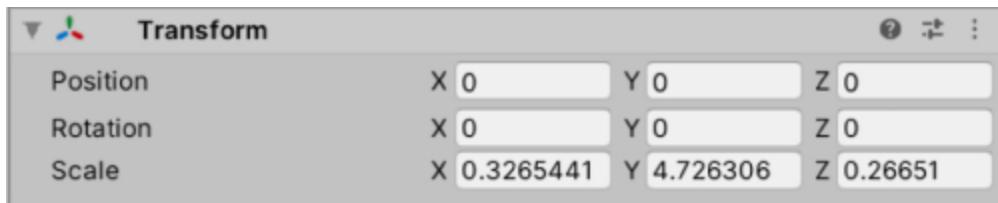
Your sphere is rotating around the Y axis at the very center of the scene. Let's add a very tall and very skinny cylinder to make it visible. **Choose 3D Object >> Cylinder** from the GameObject menu to create a new cylinder. Make sure it's selected in the Hierarchy window, then check the Inspector window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

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



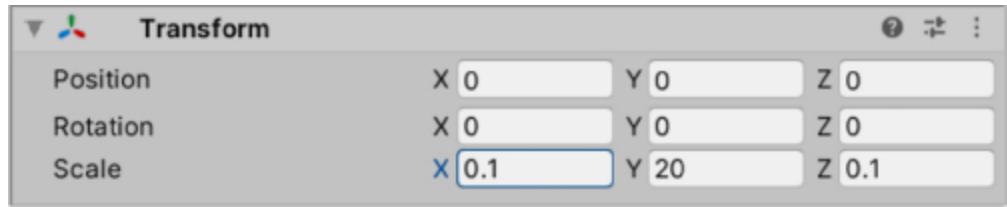
The Scale Gizmo looks a lot like the Move Gizmo, except that it has cubes instead of cones at the end of each axis. Your new cylinder is sitting on top of the sphere—you might see just a little of the sphere showing through the middle of the cylinder. When you make the cylinder narrower by changing its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

Click and drag the green cube up to elongate your cylinder along the Y axis. Then click on the red cube and drag it towards the cylinder to make it very narrow along the X axis, and do the same with the blue cube to make it very narrow along the Z axis. Watch the Transform panel in the Inspector as you change the cylinder's scale—the Y scale will get larger, and the X and Z will get much smaller.

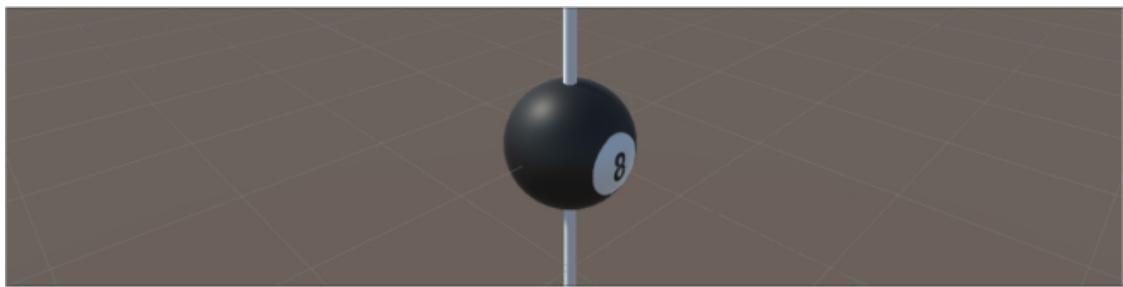


Click on the X label in the scale row in the Transform window and drag up and down. Make sure you click the actual X label to the left of the input box with the number. When you click the label it turns blue, and a blue box appears around the X value. As you drag your mouse up and down, the number in the box goes up and down, and the Scene view updates the scale in as you change it. Look closely as you drag—the scale can be positive and negative.

Now **select the number inside the X box and type .1 –** the cylinder gets very skinny. Press tab and type 20, then press tab again and type .1, and press enter.



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



Add fields to your class for the rotation angle and speed

In [Chapter 1](#) you learned how C# classes can have **fields** that store values methods can use. Let's modify your code to use fields. Add these four lines just under the class declaration, **immediately after the first curly brace {:**

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

```
public float ZRotation = 0;  
public float DegreesPerSecond = 180;
```

NOTE

These are like the fields that you added to the animal match game in [Chapter 1](#). They're variables that keep track of their values—each time `Update` is called it reuses the same field over and over again.

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

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

NOTE

You'll learn a lot more about fields and the `new` keyword in the next chapter.

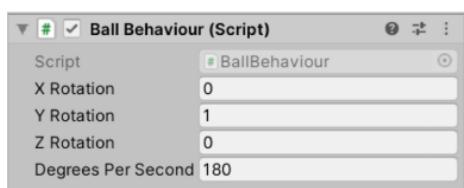
The `DegreesPerSecond` field contains the number of degrees to rotate per second, which you'll multiply by `Time.deltaTime` just like before. **Modify your `Update` method to use the fields.** This new code creates a `Vector3` variable called `axis` and passes it to the `transform.Rotate` method:

```

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

```

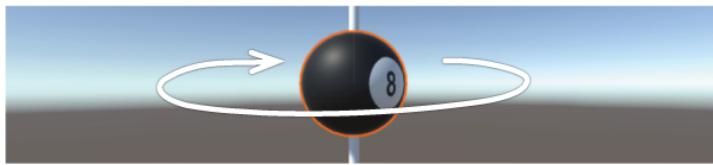
Select the Sphere in the Hierarchy window. Your fields now show up in the Script component. When the Script component renders fields, it adds spaces between the capital letters to make them easier to read.



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

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

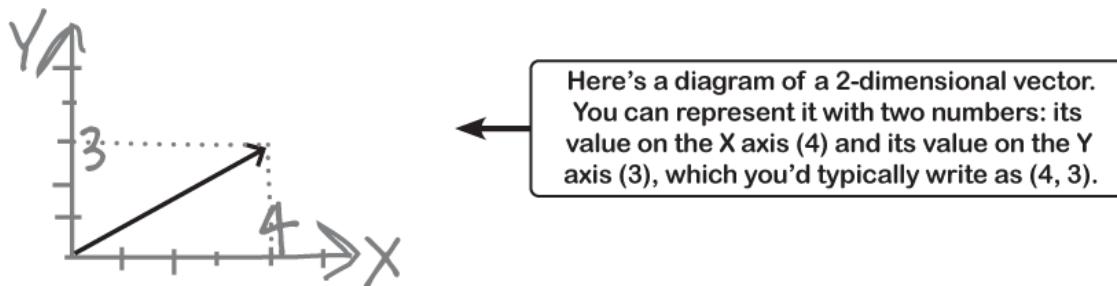
While the game is stopped, use the Unity editor to change the X Rotation to field to 1 and the Y Rotation field to 0. Start your game—the ball will rotate away from you. Click the X Rotation label and drag it up and down to change the value while the game is running,. As soon as the number turns negative,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



When you use the Unity editor to set the YRotation field to 1 and then start your game, the ball rotates clockwise around the Y axis.

Use Debug.DrawRay to explore how 3D vectors work

A **vector** is a value with a **length** (or magnitude) and a **direction**. If you ever learned about vectors in a math class, you probably saw lots of diagrams like this one of a 2D vector:



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

Use Unity to visualize vectors in 3D

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

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

So what does this line do, exactly?

- **It has a type:** `vector3`. Every variable declaration starts with a type. Instead of using `string`, `int`, or `bool`, you're declaring it with the type `Vector3`. This is a type that Unity uses for 3D vectors.
- **It has a variable name:** `axis`. It has a type: `Vector3`. And it has a type, just like the similar variable declarations that you saw in [Chapter 1](#)—except instead of `string`, `int`, or `bool` you're declaring it with the type `vector3`, Unity's 3D vector type.
- **It uses the new keyword to create a `vector3`.** It uses the `XRotation`, `YRotation`, and `ZRotation` fields to create a vector with those values.

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

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

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

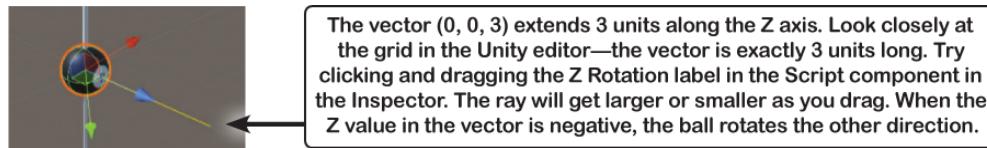
Run the game to see the ray in the Scene view

Now run your game again. You won't see anything different in the Game view because the `Debug.DrawRay` is a tool for debugging that doesn't affect gameplay at all. So the Scene tab to **switch to the scene view**. You may also need to **reset the Wide layout** by choosing Wide from the layout dropdown.



Now you're back in the familiar Scene view. Then do these things to get a real sense of how 3D vectors work:

- Set the X rotation to 0, Y rotation to 0, and Z rotation to 3. The ball was mid-rotation, so use the context menu  in the Transform component to reset its position. You should now see a yellow ray coming directly out of the Z axis and the ball rotating around it.



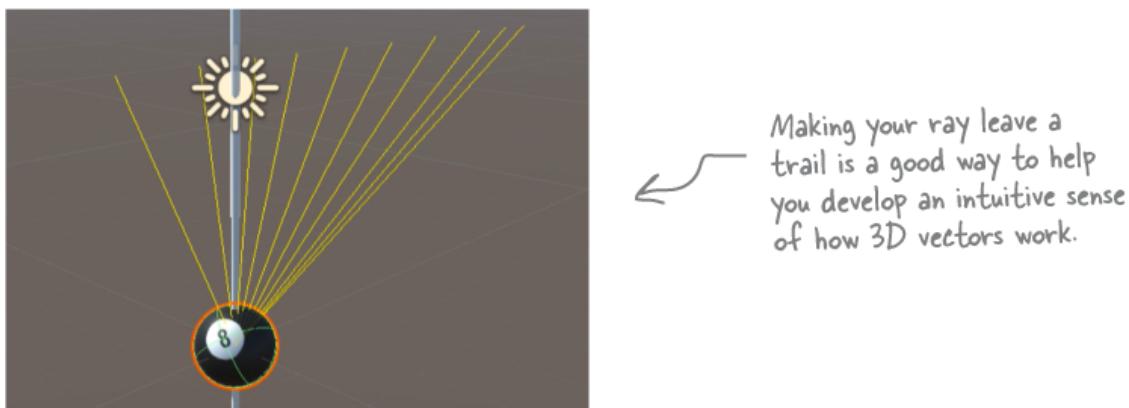
- Set the Z Rotation back to 3. Experiment with dragging the X Rotation and Y Rotation values to see what they do to the ray. Make sure to reset the Transform component each time you change them.
- Use the Hand Tool and the Scene Gizmo to get a better view. Click the X cone on the Scene Gizmo to set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

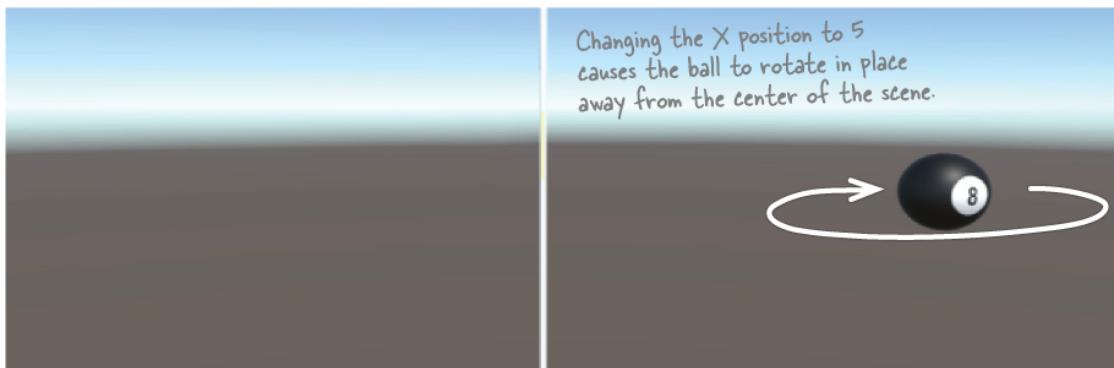
```
Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
```

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.



Rotate your ball around a point in the scene

Your code calls `transform.Rotate` method to rotate your ball around its center, which changes its X, Y, and Z rotation values. **Select Sphere in the Hierarchy window and change its X position to 5** in the Transform component. Then **use the context menu to reset the Script component** to reset its fields so it rotates around the Y axis again. When you run the game, the ball will be at position (5, 0, 0) and rotating around its own Y axis.



Let's modify the `Update` method to use a different kind of rotation. Now we'll make the ball rotate around the center point of the scene, coordinate [0, 0, 0] using the **`transform.RotateAround` method**, which rotates a `GameObject` around a point in the scene. (This is *different* from the `transform.Rotate` you used earlier, which rotates it a `GameObject` around its center.) Its first parameter is the point to rotate around. We'll use `Vector3.zero` for that parameter, which is a shortcut for writing `new Vector3(0, 0, 0)`.

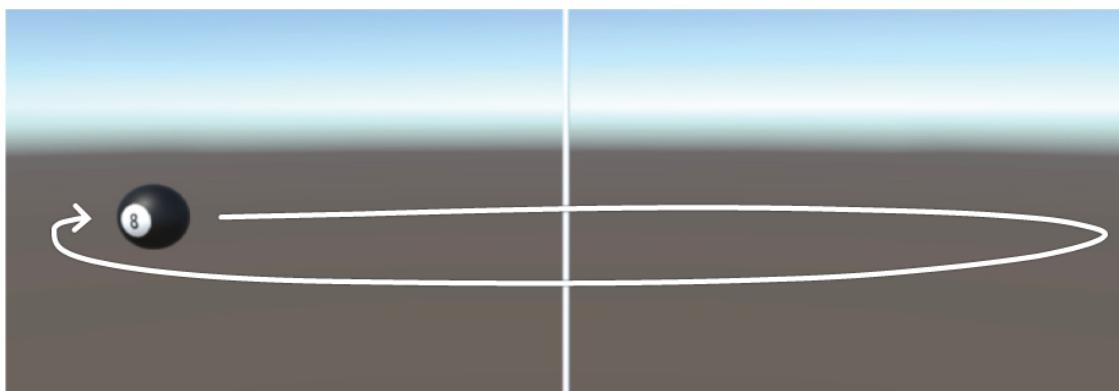
NOTE

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

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:



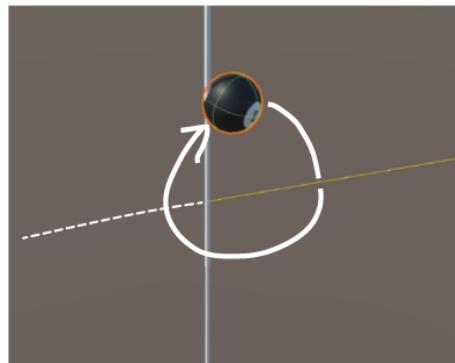
Use Unity to take a closer look at rotation and vectors

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

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

- **Switch back to the Scene view** so you can see the yellow ray that `Debug.DrawRay` renders in your `BallBehaviour.Update` method.
- Use the Hierarchy window to **select the Sphere**. You should see its components in the Inspector window.
- Change the **X Rotation, Y Rotation, and Z Rotation values** in the Script component to `10` so you see the vector rendered as a long ray. Use the Hand Tool (Q) to rotate the Scene View until you can clearly see the ray.

- Use the Transform component's context menu (⋮) to **reset the Transform component**. Since the center of the sphere is now at the zero point in the scene (0, 0, 0), it will rotate around its own center.
- Then **change the X position** in the Transform component to 2. The ball should now be rotating around the vector. You'll see the ball cast a shadow on the Y axis cylinder as it flies by.



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

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

NOTE

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

Get creative!

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

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

NOTE

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#.

NOTE

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 makes it break after the statement has executed a certain number of times.
- A **field** is a variable that lives inside of a class outside of its methods, and it retains its value between method calls.
- When you add fields to the class in your Unity script, the Script component displays **input boxes that let you modify those fields**. It inserts spaces between capital letters in the field names to make them easier to read.
- You can create 3D vectors using `new Vector3`. (You'll learn more about the `new` keyword in [Chapter 3](#).)
- The **Debug.DrawRay** method draws a vector in the Scene view (but not the Game view). You can use vectors as a debugging tool, but also as a learning tool.
- The **transform.RotateAround** method rotates a GameObject around a point in the scene.

Chapter 3. Objects... Get Oriented!

Making code make sense

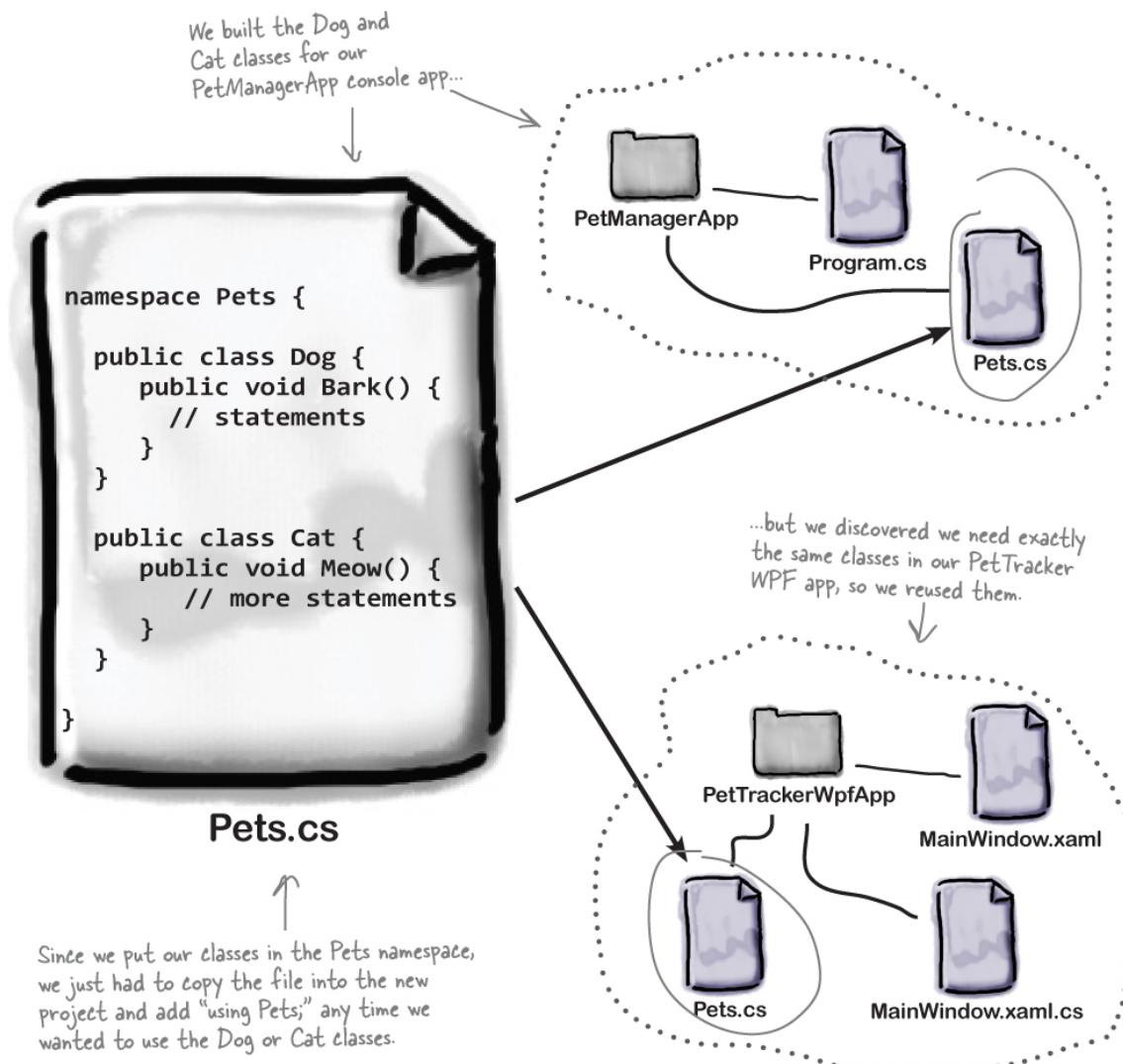


Every program you write solves a problem.

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

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 return a value

In the next project, we'll use methods that have a **return value**, or 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 the `return` statement.

Here's an example of a method that has a return type—it returns an `int`:

```
int TimesTwo(int factor1, int factor2) {  
    int result = factor1 * factor2;  
    return result; }  
The return type is int, so the method must return an int value  
The return statement passes the value back to the statement that called the method.
```

NOTE

A method with a return type uses the `return` keyword to send a value back to the statement that called it.

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 `TimesTwo` method and stores the result in a variable called `area`.

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

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

Do this!

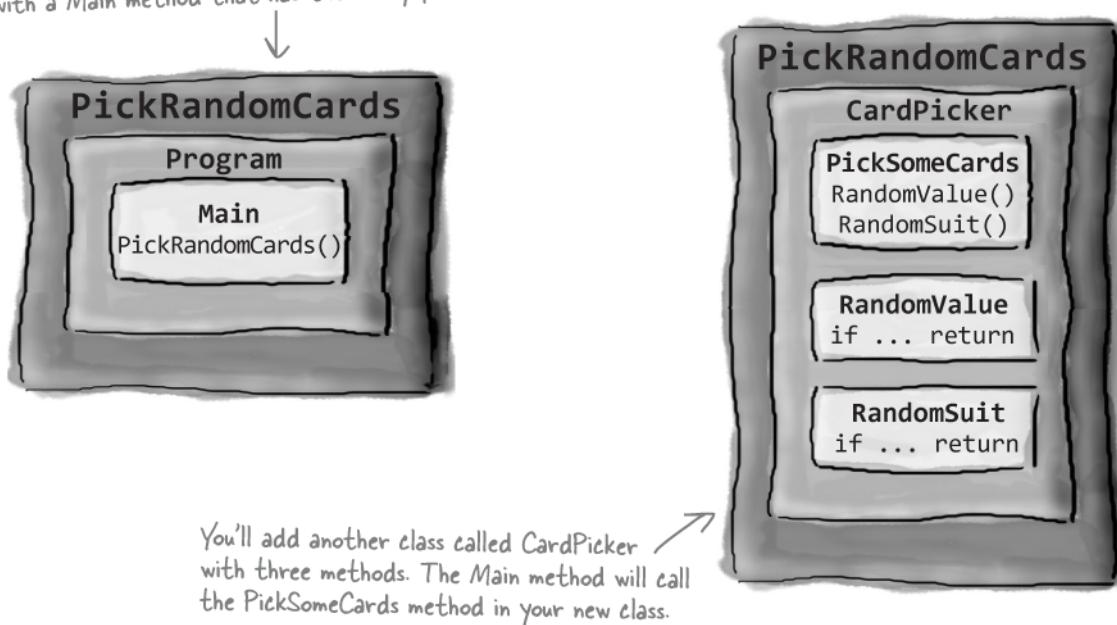
Since you're about to create methods that return values, right now is a perfect time to use the debugger to *really dig into how the return statement works*.

- 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.



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.

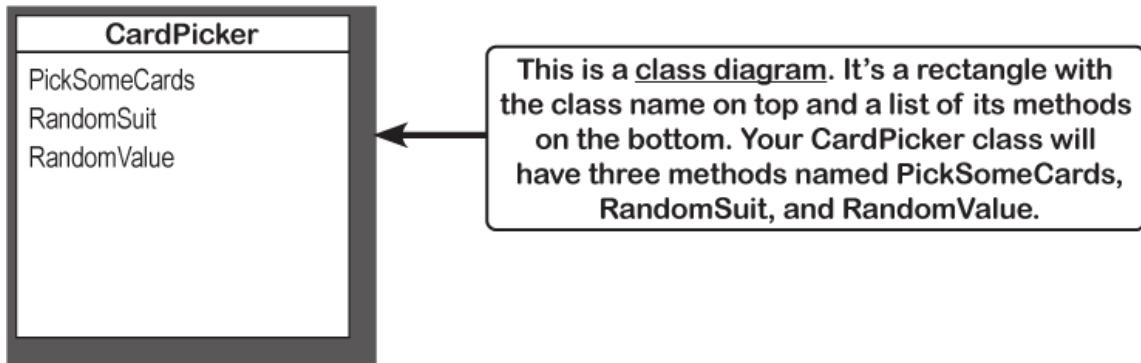


After your array is generated, you'll use a `foreach` loop to write it to the console.

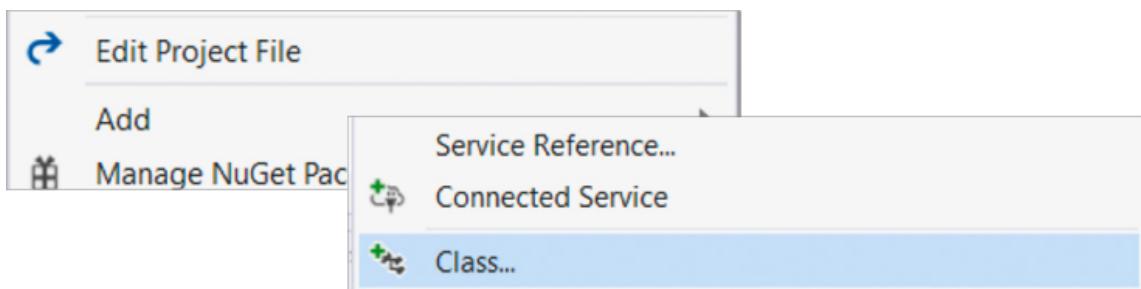
Create your `PickRandomCards` console app

Do this!

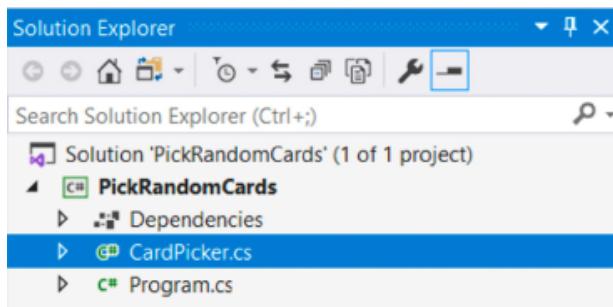
Let's use what you've learned so far to create a program that picks a number of random cards. Open Visual Studio and **create a new Console App project called `PickRandomCards`**. Your program will include a class called `CardPicker`. Here's a class diagram that shows its name and methods:



Right-click on the PickRandomCards project in the Solution Explorer and **choose Add >> Class...** 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 of it. **Add a new method called `PickSomeCards`.** Here's what your class should look like:

```

class CardPicker
{
    public static string[] PickSomeCards(int numberOfCards)
    {
        Make sure you include the
    } public and static keywords.
    We'll talk more about
    them later in the chapter.
}

```

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

Finish your PickSomeCards method

Now do this!

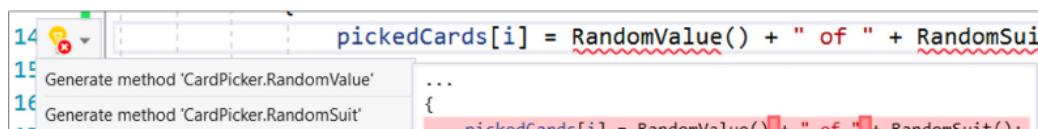
- Your PickSomeCards method needs a return statement, so let's add one.** Go ahead and fill in the rest of the method—and now that it uses a return statement to return a string array value, the error goes away:

```

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.
}

```

- 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 okay 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 that statement 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 methods in a program. Let's imagine that we'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();

```

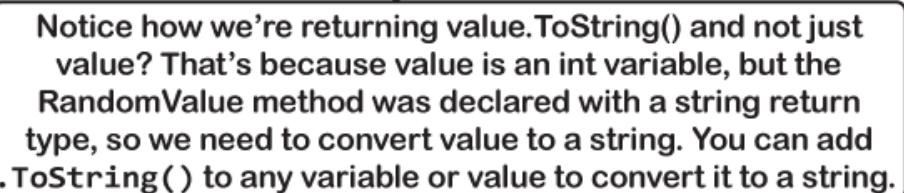
Now add code to your `RandomSuit` method that takes advantage of return statements to stop executing the method as soon as it finds a match. The random number generator's `Next` method can take two parameters: `random.Next(1, 5)` returns a number that's at least 1 but less than 5 (in other words, a random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

NOTE

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

Your finished CardPicker class

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

```

class CardPicker
{
    static Random random = new Random(); ← 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 ← 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.
        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";
    }
}

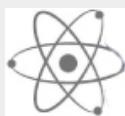
```



RELAX

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

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



BRAIN POWER

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



EXERCISE

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

Useful method #1: Console.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. But the line of input you get from the user is a string, so you'll need a way to convert it to an int. You'll use the int.TryParse method for that:

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

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

Put it all together

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

- Use Console.Write to ask the user for the number of cards to pick.
- Use Console.ReadLine to read a line of input into a string variable called line.
- Use int.TryParse to try to convert it to an int variable called numberOfCards.
- If the user input **could be converted** to an int value, use your CardPicker class to pick the number of cards that the user specified: CardPicker.PickSomeCards(numberOfCards) – use a string[] variable to save the results, then use a foreach loop to call Console.WriteLine on each card in the array. Flip back to Chapter 1 to see an example of a foreach loop—you'll use it to loop through every element of the array. Here's the first line of the loop: foreach (string card in CardPicker.PickSomeCards(numberOfCards))

- If the user input **could not be converted**, use Console.WriteLine to write a message to the user indicating that the number was not valid.

NOTE

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.Write("Enter the number of cards to pick: ");
    string line = Console.ReadLine();
    if (int.TryParse(line, out int numberOfCards))
    {
        foreach (string card in CardPicker.PickSomeCards(numberOfCards))
        {
            Console.WriteLine(card); ↗
        }
    }           This foreach loop executes Console.WriteLine(card) for
    else         each element in the array returned by PickSomeCards.
    {
        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.

NOTE

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:

```
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.
```

NOTE

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 just getting started on her next one.



Ana's started working on some **prototypes**. In one exciting part of the game, the player needs to escape a failed heist attempt and evade the cops. She's been working on the code for the enemies that the player has to avoid. She'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, attacking the player if the player was found, and capturing the player if the enemy gets too close.



IN MY NEXT GAME,
THE PLAYER IS PULLING
OFF THE HEIST OF THE
CENTURY.

O

O

```
SearchForPlayer();
```



```
if (SpottedPlayer()) {  
    CommunicatePlayerLocation();  
}
```



```
CapturePlayer();
```



```
StopSearching();
```

Ana's game is evolving...

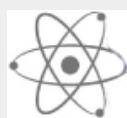
The bank heist idea is pretty good, but Ana's not 100% sure that's the direction she wants to go. She's also thinking about a nautical game where the player has to evade pirates. Or maybe it's a zombie survival game set on a creepy farm. In all three of those ideas, she thinks the enemies will have different graphics, but their behavior can be driven by the same methods.





... so how can Ana make things easier for herself?

Ana's not sure which direction the game should go, so she wants to make a few different prototypes—and she wants them all to have the same code for their enemies, with the SearchForPlayer, StopSearching, SpottedPlayer, CommunicatePlayerLocation,, and CapturePlayer methods. She's got her work cut out for her.



BRAIN POWER

Can you think of a good way for Ana to use the same methods for enemies in different prototypes?



I PUT ALL OF THE ENEMY BEHAVIOR METHODS INTO A SINGLE ENEMY CLASS. CAN I **REUSE** THE CLASS IN EACH OF MY THREE DIFFERENT GAME PROTOTYPES?

Enemy
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer



GAME DESIGN... AND BEYOND

Prototypes

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

- The first prototype is often a **paper prototype**, where you lay out the core elements of the game on paper. For example, you can learn a lot about your game by 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 a lot the most 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 simple 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. And it's easy to get started building them—all you need is some paper and a pen or pencil. Start by choosing your favorite classic game. Platform games work especially well, so we chose one of the **most popular, most recognizable** classic video games ever made... but you can choose any game you'd like!

Here's what to do next:

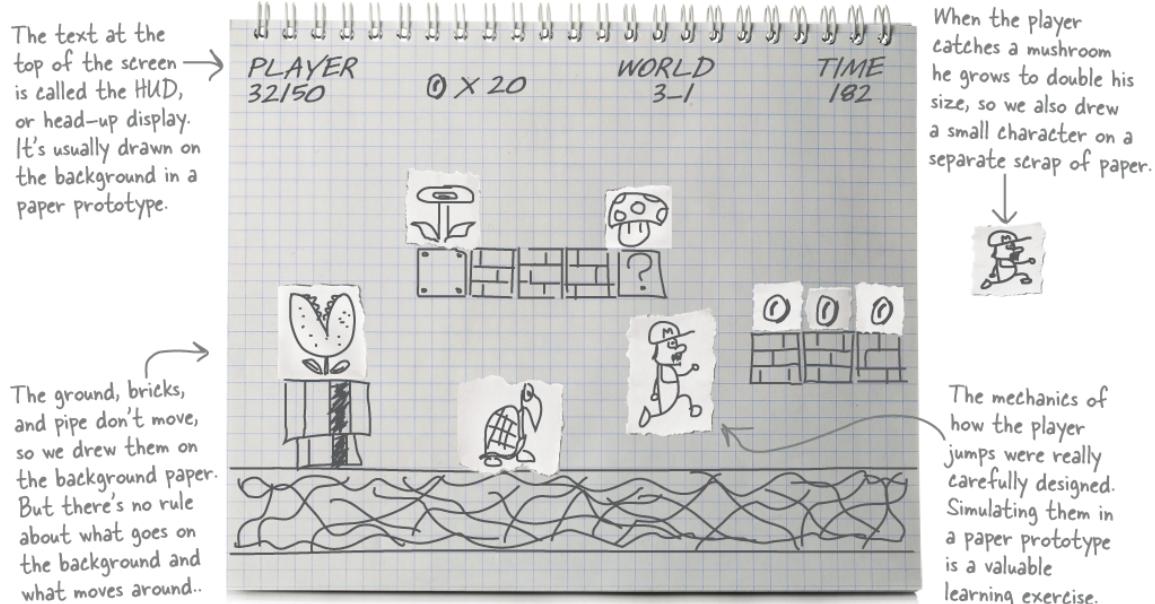
Draw this!

1. **Draw the background on a piece of paper.** Start your prototype by creating the background. In our prototype, the ground, bricks, and

pipe don't move, so we drew them on the paper. We also added the score, time, and other text at the top.

2. Tear small scraps of paper and draw the moving parts. In our prototype, we drew the characters, the piranha plant, 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! Move the player around the page. Try to simulate player movement. Drag the player around the page. Make the non-player characters move. 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 okay!)





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

NOTE

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 a user interface, building a paper prototype is a great way to get started. But sometimes you need to create a few paper prototypes before you start to 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. And **prototyping is a really valuable skill for any kind of developer**, not just a game developer.



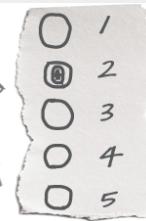
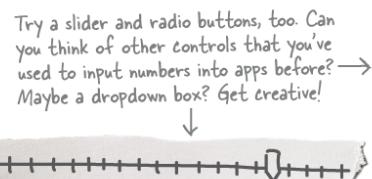
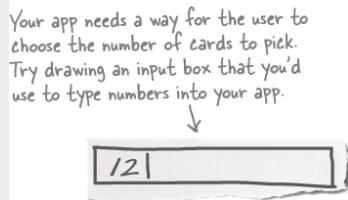
SHARPEN YOUR PENCIL

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.

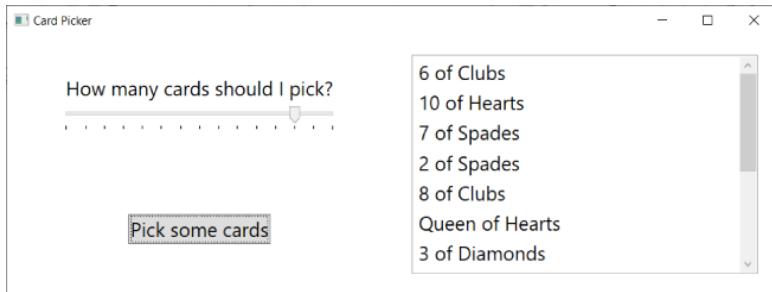


Next, draw a bunch of different types of controls on more small scraps of paper. Drag them around the window and experiment with ways to fit them together. What design do you think works best? There's no single right answer—there are lots of ways to design any app.



Up next: build a WPF version of your card picking app

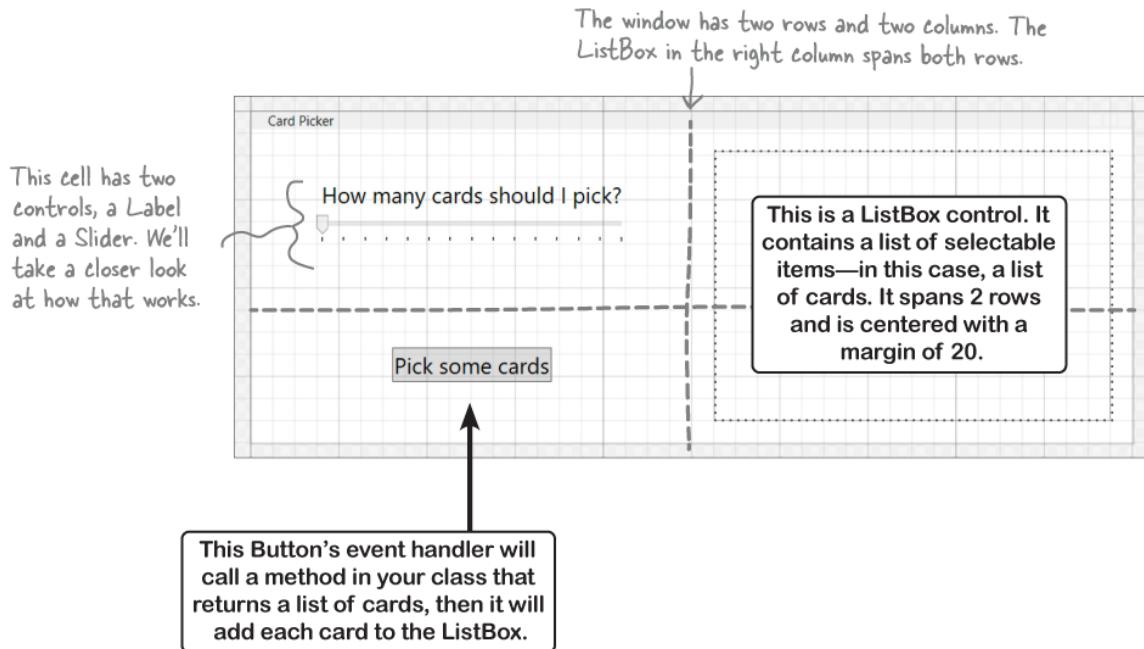
In the next project, you'll build a WPF app called PickACardUI. Here's what it will look like:



We decided to go with a slider to choose the number of cards. But that doesn't mean it's the only way to design this app! Did you come up with a different design with your paper prototype? That's okay! There are many ways to design any app, and there's almost never a single right (or wrong) answer.

Your PickACardUI app will let you use a Slider 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:



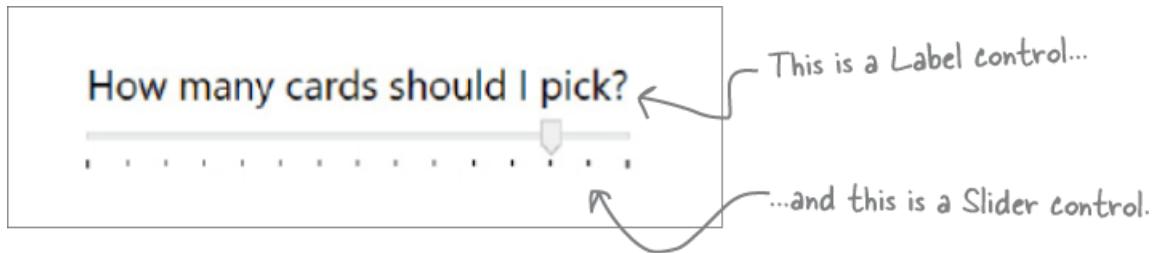
NOTE

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!

 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 match game. Before you start writing code, let's take a closer look at two controls in the upper left cell of the grid:

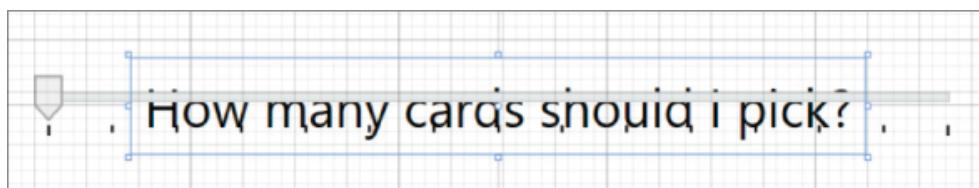


So how stack them on top of each other like that? We **could** try putting them in the same cell in the grid:

```
<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:

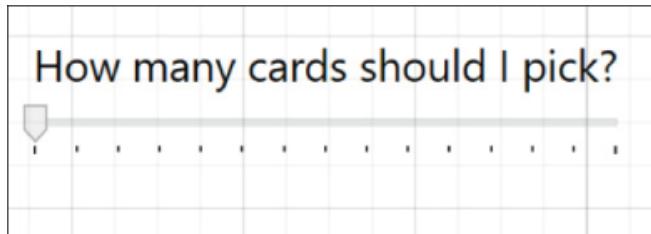


That's where a **StackPanel** control comes in handy. A StackPanel is a container control—like a Grid, its job is to contain other controls and make sure they go in the right place in the window. But while the Grid lets you arrange controls in rows and columns, a StackPanel lets you arrange controls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



NOTE

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 easy 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!

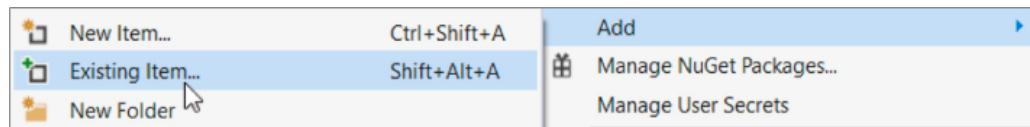
1. Create a new WPF app called PickACardUI.

You'll follow exactly the same steps that you used to create your animal match game in [Chapter 1](#):

- Open Visual Studio and create a new project.
- Select **WPF App (.NET Framework)**.
- 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.

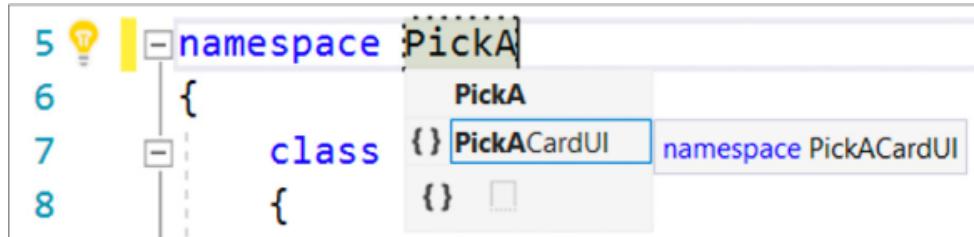
Right-click on the project name and choose **Add >> Existing Item...** from the menu.



Navigate to the folder with your console app and add CardPicker.cs to add it to your project. Your project should now have a copy of the CardPicker.cs file from your console app project.

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 auto-suggest**:



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.

- 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

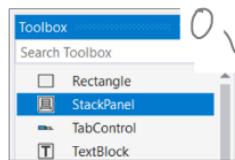
```

</Grid.ColumnDefinitions>

<StackPanel/>

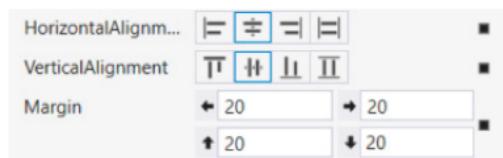
</Grid>
</Window>

```



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.

- 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 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.

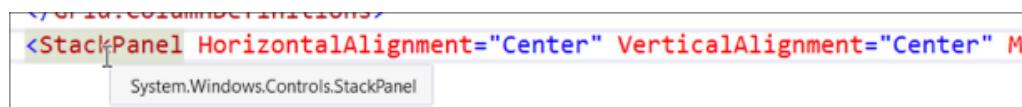
Lay out your Card Picker desktop app's window

Lay out your new card picker app's window 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 other 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 easy to add controls to it as it is to set its properties. **Click on the StackPanel to select it.**



While 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 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:

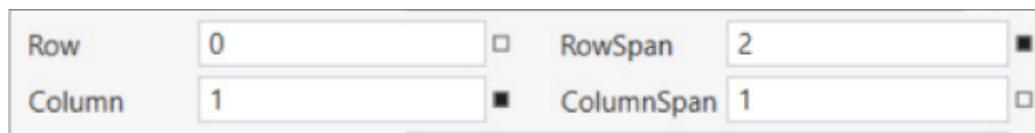
```
<Button Grid.Row="1" Content="Pick some  
cards" FontSize="20"  
HorizontalAlignment="Center"  
VerticalAlignment="Center" />
```

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.

- 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 the Row is set to 1 and Column is set to 0. 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 okay if this margin 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” – so let’s resize it, just like you did with the animal match 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**—or 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:

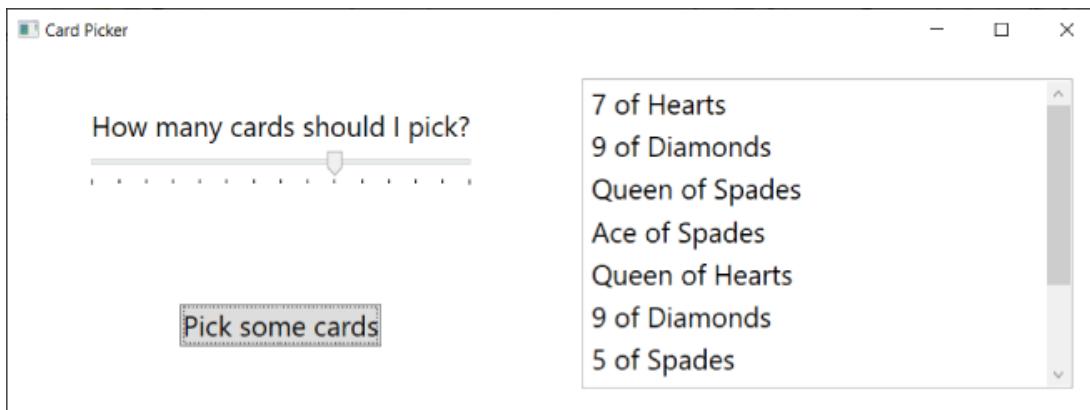
```
private void Button_Click(object sender,
    RoutedEventArgs e)
{
    string[] pickedCards =
        CardPicker.PickSomeCards((int)numberOfCards.Value);

    listOfCards.Items.Clear();
    foreach (string card in pickedCards)
    {
        listOfCards.Items.Add(card);
    }
}
```

NOTE

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

Now run your app. Use the slider to choose the number of random cards to pick, then press the button to add them to the ListBox. **Nice work!**

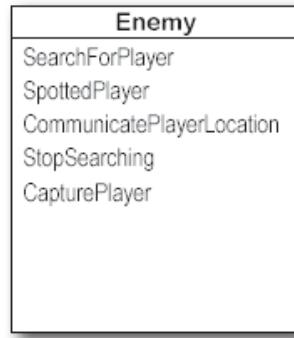


BULLET POINTS

- 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: `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 on 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 cops, pirates, or zombies, she could use the same methods from her `Enemy` class to make them work. Her game is starting to shape up. Ana's early prototypes only had a single enemy. That worked fine with her `Enemy` class.

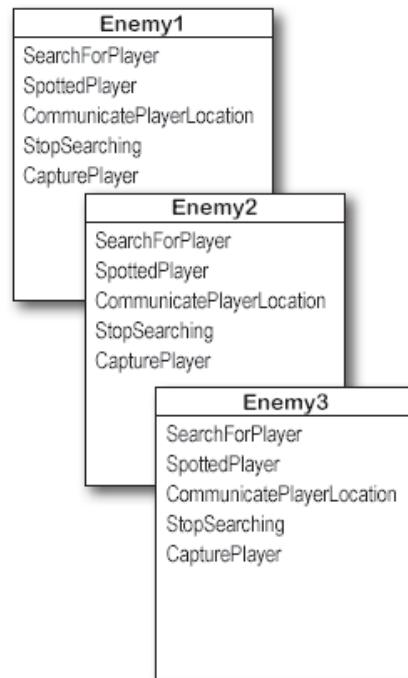


... but what if she wants more than one enemy?

And that's great... until Anna wants more than one enemy! 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?



NOTE

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

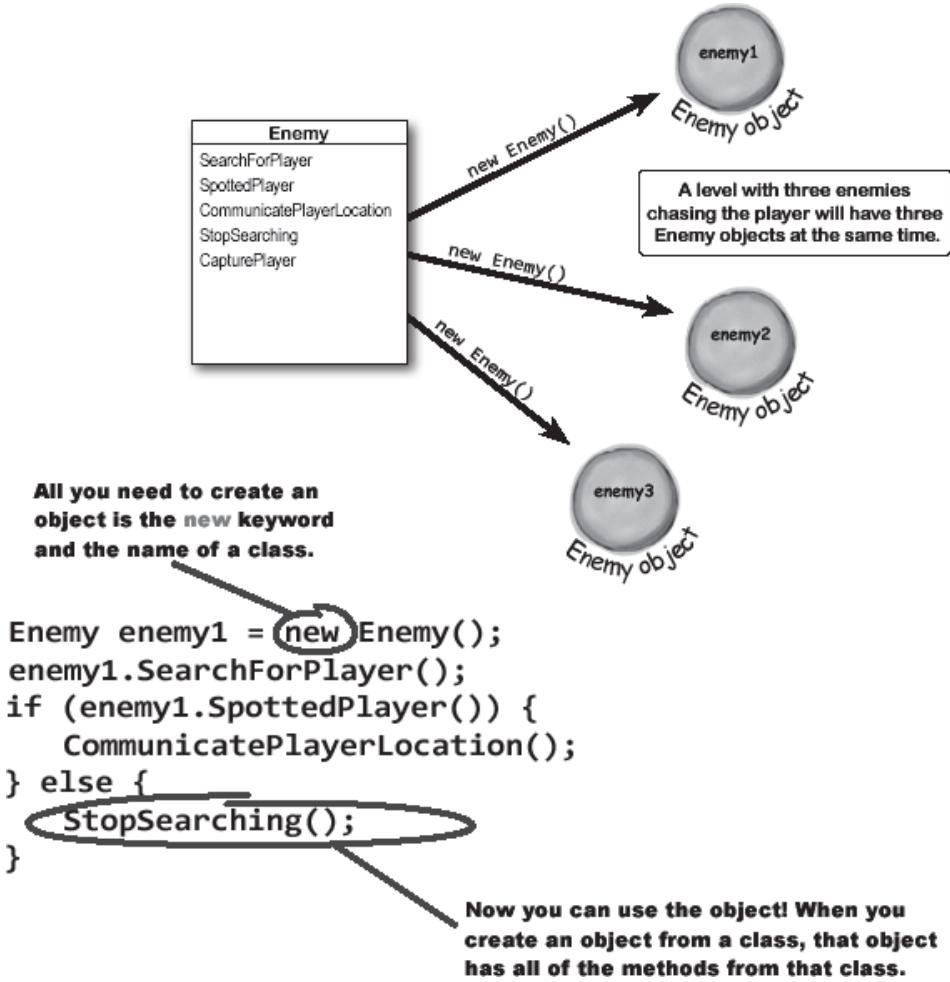


Maintaining three copies of the same code is really messy.

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 enemies 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, but she can also 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.

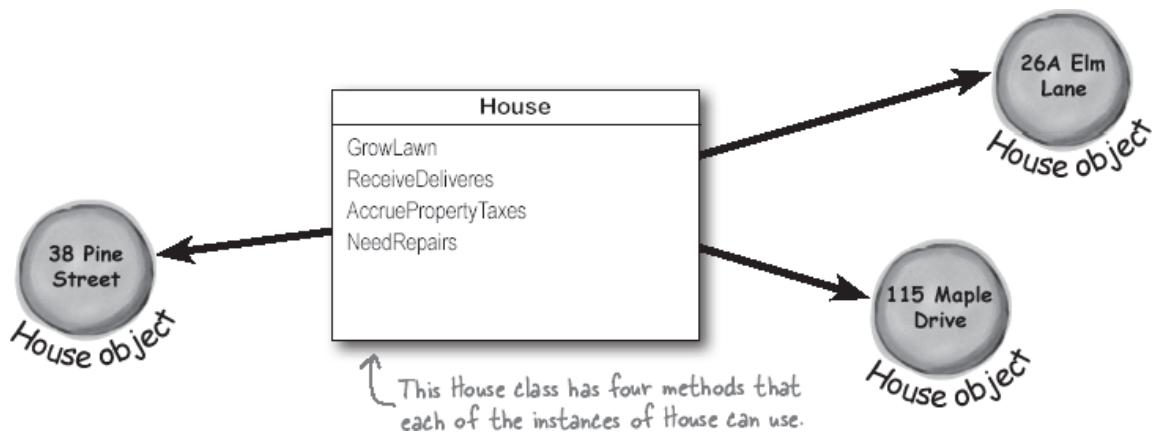


NOTE

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, 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. You the class as the variable type to declare the variable, so instead of int or bool, you'll use a class like House or Enemy.

in-stance, noun.

an example or one occurrence of something. *The IDE search-and-replace feature finds every instance of a word and changes it to another.*

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

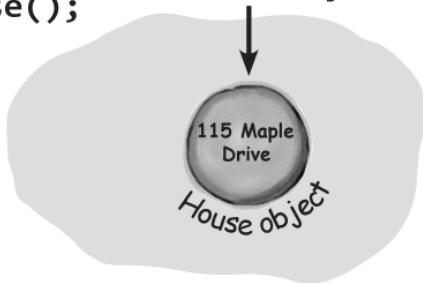


Your program executes a new statement.

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

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

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



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

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

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

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

You're creating 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

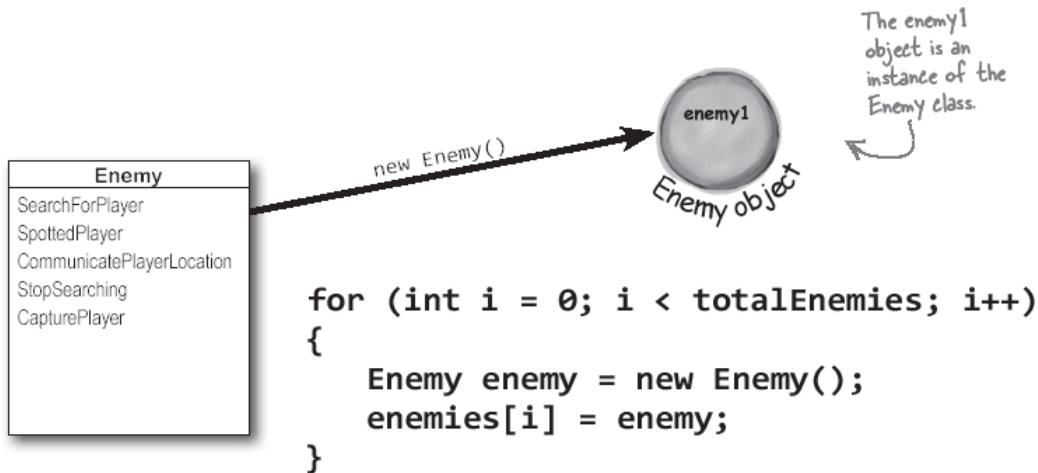
Ana used objects to reuse the code in the Enemy class without all that messy copying that would leave duplicate code all over her project. Here's how did it.

1. Ana created a Level class that stored the enemies in an **Enemy array** called `enemies`, just like you used string array to store cards or animal emoji.

```
public class Level {  
    Enemy[] enemies = new Enemy[3];
```

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

2. She used a loop that called `new` statements to create new instances of the Enemy class for the level and add them to a array of enemies.



3. She called methods of each `Enemy` instance during every frame update to implement the enemy behavior.



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



NOTE

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



WAIT A MINUTE! YOU DIDN'T GIVE ME
**NEARLY ENOUGH INFORMATION TO
BUILD ANA'S GAME.**

That's right, we didn't. Some game prototypes are really simple, while others are much more complicated—but complicated programs **follow the same patterns** as simple ones. Ana's game program is an example of how someone would use objects in real life. And this doesn't just apply to game development! No matter what kind of program you're building, you'll use objects in exactly the same way that Ana did in her game. Ana's example is just the starting point for getting this concept into your brain. We'll give you **lots more examples** over the rest of the chapter—and this concept is so important that we'll even 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 this.

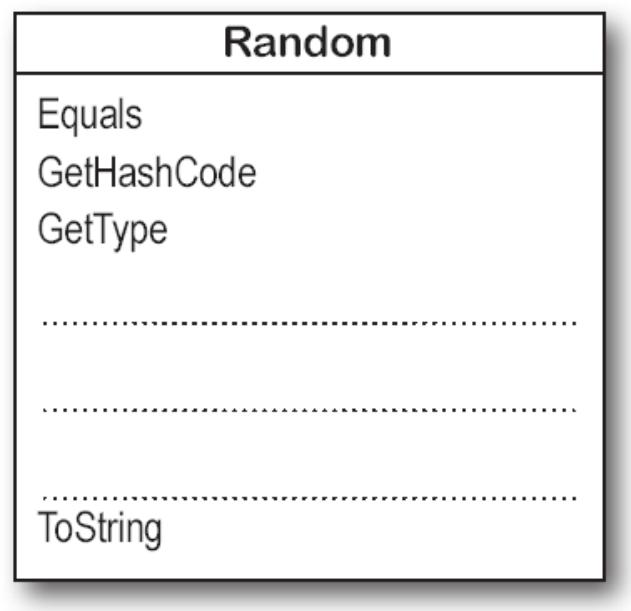
115 Maple Drive
House object



SHARPEN YOUR PENCIL

Now that you've got a better idea of how objects work, it's a great time to go back to your CardPicker class and get to know the Random class that you're using.

1. Put your cursor inside of any of the methods, press enter to start a new statement, then type `random.` – as soon as you type the period, Visual Studio will pop up an IntelliSense window that shows its methods. Each method is marked with a cube icon (). We filled in some of the methods. Finish filling in the class diagram for the Random class.



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 = double[20]; ..... ← We filled in part  
..... double value = ..... ← of the code, plus  
..... ..... ← the ending curly  
..... brace. Your job  
..... is to finish those  
..... statements and  
..... then write the  
..... rest of the code.  
..... }
```



SHARPEN YOUR PENCIL SOLUTION

Now that you've got a better idea of how objects work, it's a great time to go back to your CardPicker class and get to know the Random class that you're using.

1. Put your cursor inside of any of the methods, press enter to start a new statement, then type `random.` – as soon as you type the period, Visual Studio will pop up an IntelliSense window that shows its methods. Each method is marked with a cube icon (⟨⟩). We filled in some of the methods. Finish filling in the class diagram for the Random class.

The screenshot shows the Visual Studio IDE with the code editor open. A tooltip window is displayed over the word `random.` in the code. The tooltip lists several methods of the `Random` class:

random.	Equals
	GetHashCode
	GetType
	Next
	NextBytes
	NextDouble
	ToString

Annotations explain the tooltip:

- A callout points to the `Next` method with the text: "Here's the IntelliSense window that Visual Studio popped up when you typed "random." inside one of your CardPicker methods."
- A callout points to the `NextDouble` method with the text: "When you select the NextDouble in the IntelliSense window, it shows documentation for the method."
- A callout points to the `int Random.Next()` entry with the text: "int Random.Next() (+ 2 overloads) Returns a non-negative random integer."
- A callout points to the `double Random.NextDouble()` entry with the text: "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 = 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. And we just saw how you used the `static` keyword to declare a field in your CardPicker class:

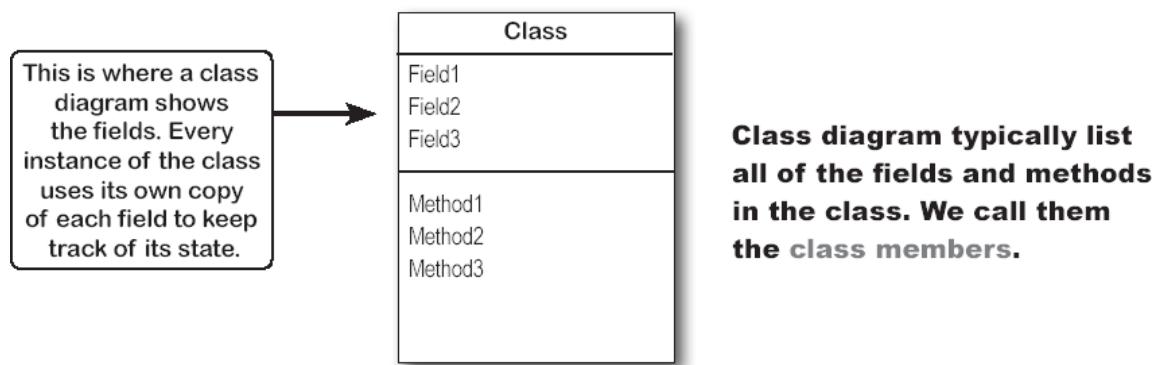
```
static Random random = new Random();
```

But 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.

NOTE

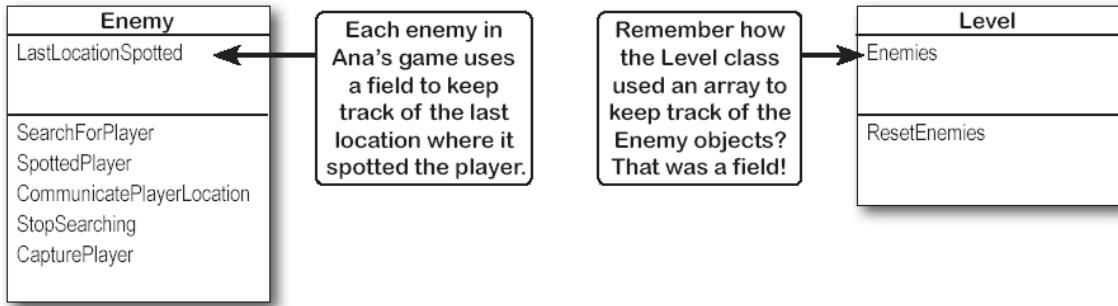
Sometimes people think the word “instantiate” sounds a little weird, but it makes sense when you think about what it means.

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



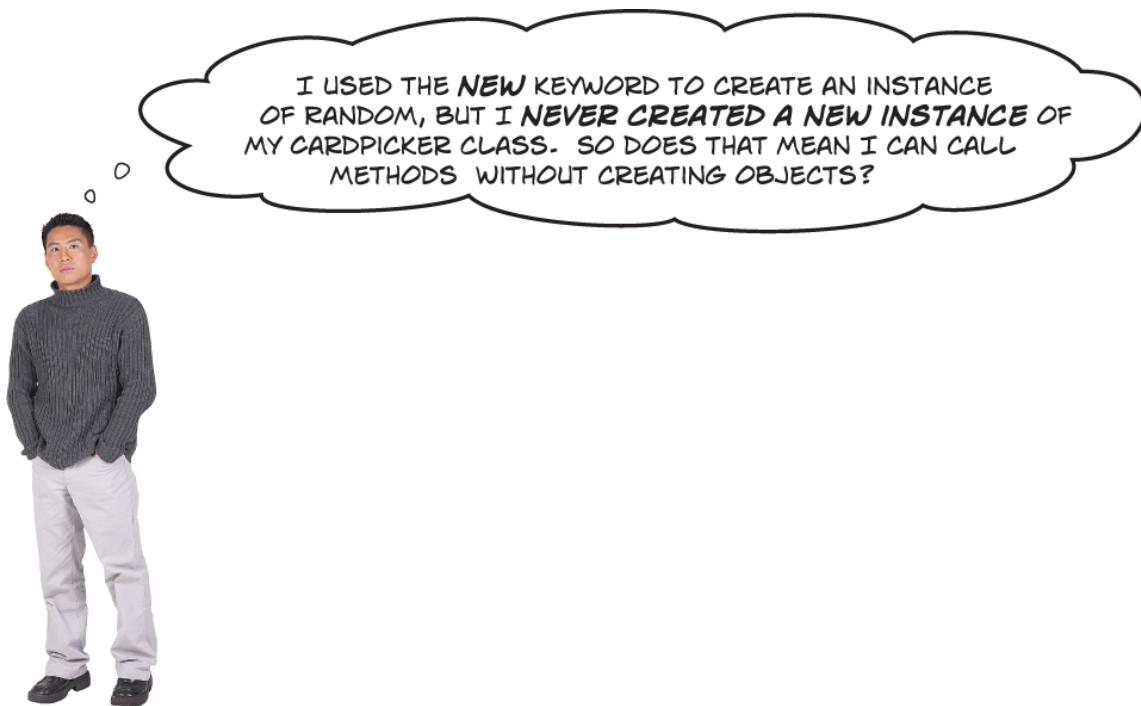
Methods are what an object does. Fields are what the object knows.

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.



NOTE

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



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 call 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.

NOTE

When a field is static, there's only one copy of it shared by all instances



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?

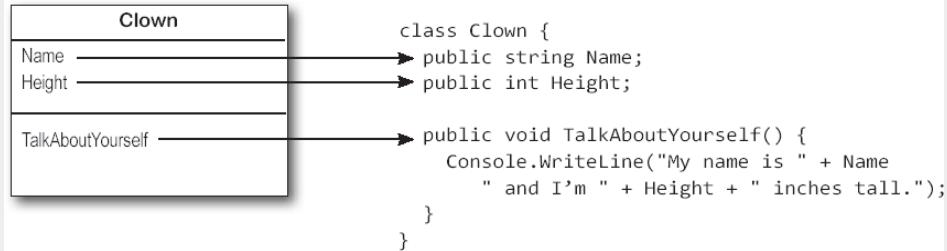
A: Because if you have an object that's keeping track of certain data—like Ana's instances of her Enemy class that each kept track of different enemies in her game—then you can use each instance's methods to work with that data. So when Ana's game calls the StopSearching method on the enemy2 instance, it only causes that one enemy to stop searching for the player. It didn't affect the enemy1 or enemy3 objects, and they could 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.



SHARPEN YOUR PENCIL

Here's a .NET Console App that writes several lines to the console. It includes a class called Clown that has two fields, Name and Height, and a method called TalkAboutYourself. Your job is to read the code and write down the lines that are printed to the console.

Here's the class diagram and code for the Clown class:



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.

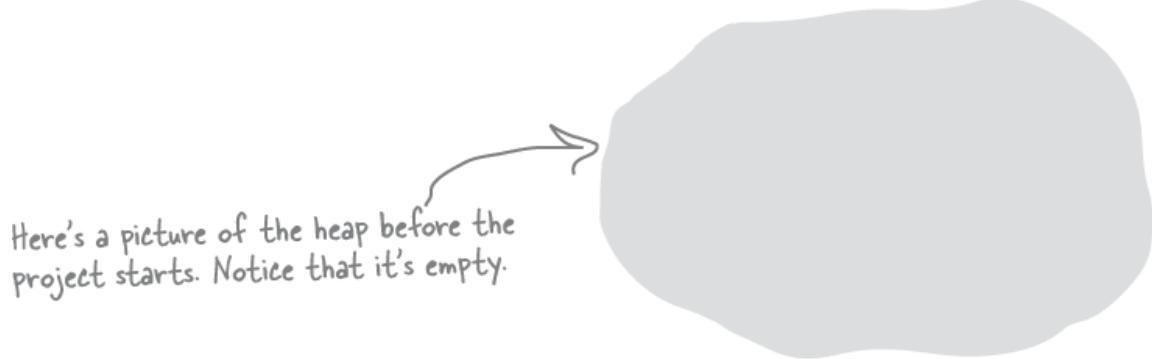
    Clown clown3 = new Clown();
    clown3.Name = anotherClown.Name;
    clown3.Height = oneClown.Height - 3;
    clown3.TalkAboutYourself();       // My name is _____ and I'm ____ inches tall.

    anotherClown.Height *= 2;         // Another call to TalkAboutYourself()
    anotherClown.TalkAboutYourself(); // My name is _____ and I'm ____ inches tall.
}
```

Remember, 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.



NOTE

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

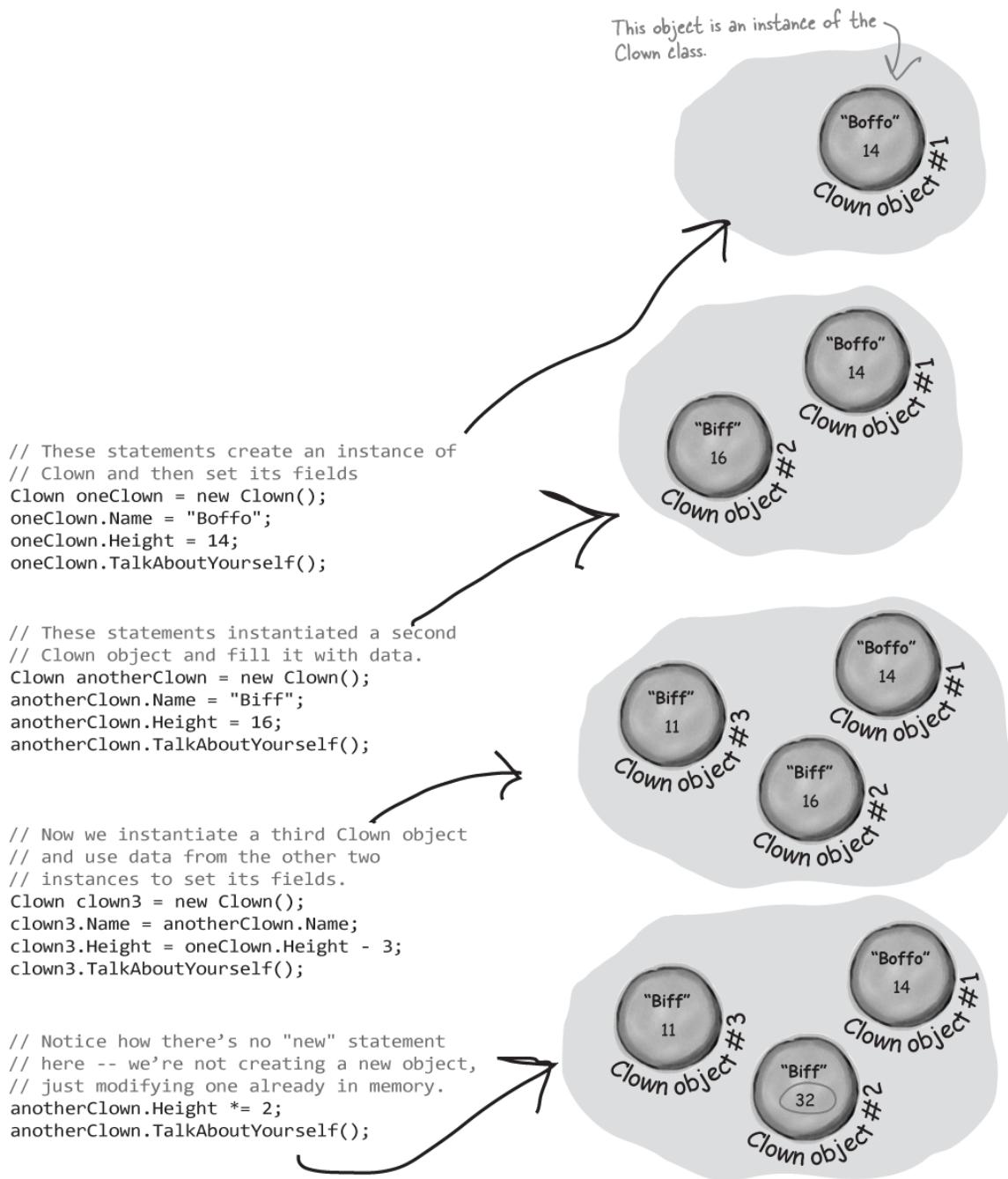
Let's take a closer look at the program in the “Sharpen your Pencil” exercise, starting with the first line of the Main method. It's actually **two statements** combined into one:

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.



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? Or 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 that code. 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` 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. And 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:

<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.

But 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 lucky for

us, 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 code with the manual that tells us what the code is supposed to do. Adding comments can definitely help us understand what it's supposed to do.

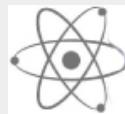
```
/* This code runs every 3 minutes to check the temperature
 * If it exceeds 160C we need to vent the cooling system
 */
int t = m.chkTemp();
if (t > 160) {
    // Get the controller system for the turbines
    T tb = new T();

    // Close throttle valve on turbine #2
    tb.clsTrpV(2);

    // Fill and vent the isolation cooling system
    ics.Fill();
    ics.Vent();

    // Initiate the air system check
    m.airsyschk();
}
```

Adding extra line breaks to your code in some places can make it easier to read.



BRAIN POWER

Code comments are a good start. But 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. And it turns out that `m` was a class that controlled 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 the 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 okay.

```
/// <summary>
/// If the nougat temperature exceeds 160C it's too hot.
/// </summary>
public bool IsNougatTooHot() {
    int temp = CandyBarMaker.CheckNougatTemperature();
    if (temp > 160) {
        return true;
    } else {
        return false;
    }
}
```

When we renamed 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

The IDE makes it easy to 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 also 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!

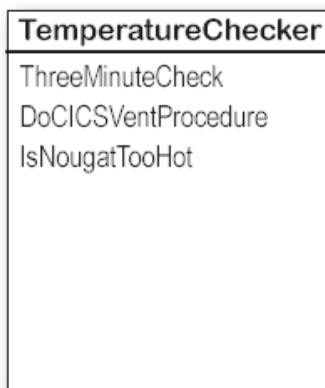
So now we have the `IsNougatTooHot` and `DoCICSVentProcedure` methods, we can ***rewrite the original confusing code as a single method***—and we can give it a name that makes clear exactly what it does.

```
/// <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();  
}  
}
```

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.**

We bundled these new methods into a class called TemperatureChecker. Here's its class diagram.



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.



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 easy to understand. 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. But no matter how well we plan our code, we almost never get things exactly right the first time.

That's why ***advanced developers constantly refactor their code.***

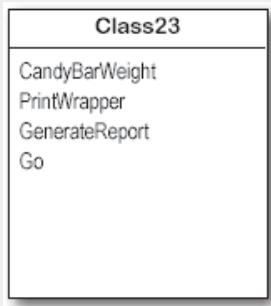
They'll move code into methods and give them names that make sense.

They'll rename variables. Any time they see code that isn't 100% obvious, they'll take a few minutes to refactor it. They know it's worth taking the time to do it now, because it will make it easier to add more code in an hour (or a day, a month, or a year!).



SHARPEN YOUR PENCIL

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



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

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



These two classes are part of a system that a pizza parlor uses to track the pizzas that are out for delivery.

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

CashRegister
MakeSale
NoSale
PumpGas
Refund
TotalCashInRegister
GetTransactionList
AddCash
RemoveCash

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

.....

.....

.....

.....



SHARPEN YOUR PENCIL SOLUTION

Here's how we corrected the classes. We show just one possible way to fix the problems—but there are plenty of other ways you could design these classes depending on how they'll be used.

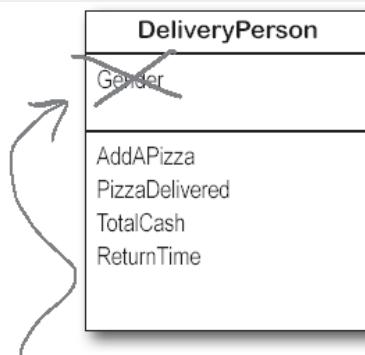


This class is part of the candy manufacturing system from earlier.

The class name doesn't describe what the class does. A programmer who sees a line of code that calls Class23.Go will have no idea what that line does. We'd also rename the method to something that's more descriptive—we chose MakeTheCandy, but it could be anything.

These two classes are part of a system that a pizza parlor uses to track the pizzas that are out for delivery.

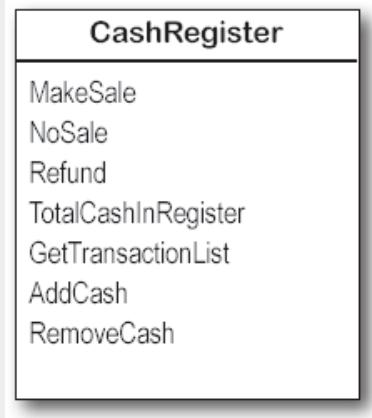
It looks like the DeliveryGuy class and the DeliveryGirl class both do the same thing—they track a delivery person who's out delivering pizzas to customers. A better design would replace them with a single class that adds a field for gender.



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

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

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



CODE TIP: A FEW IDEAS FOR DESIGNING INTUITIVE CLASSES

We're about to jump back into writing code. You'll be writing code for the rest of this chapter, and a LOT of code throughout the book. And 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.



RELAX

It's okay if you get stuck when you're writing code. In fact, it's a good thing!

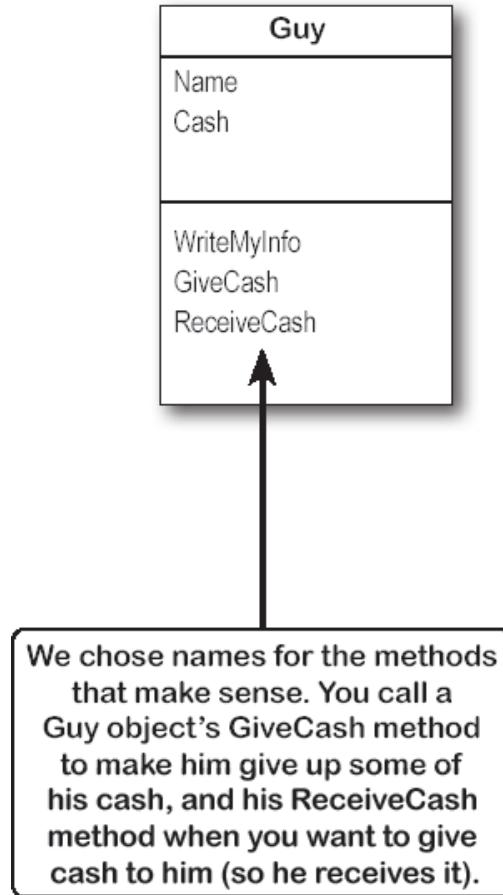
Writing code is all about solving problems—and some of them can be tricky! But if you keep a few simple things in mind, it'll make the code exercises go more smoothly:

- 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!

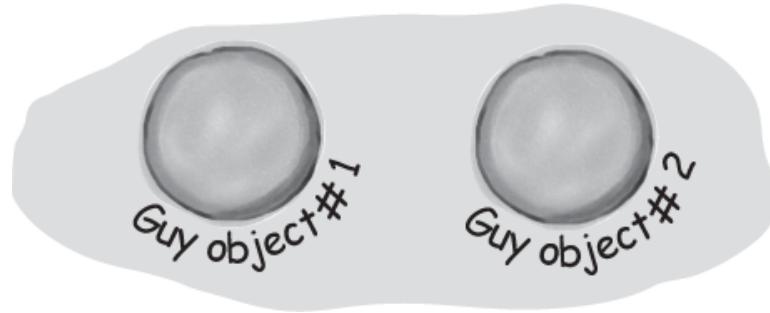
Build a class to work with some guys

Joe and Bob lend each other money all the time. Let's create a class to keep track of them. We'll start with an overview of what we'll build.



1. We'll create two instances of a "Guy" class.

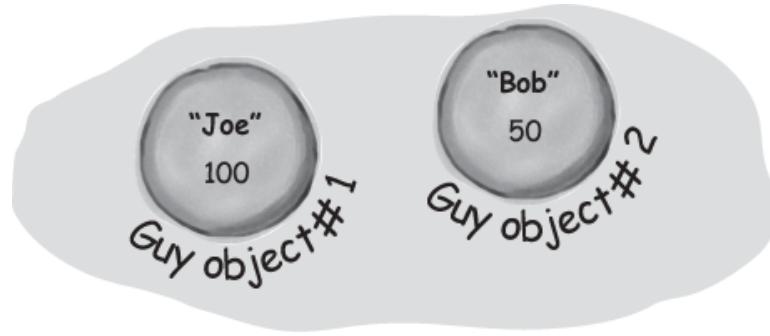
We'll use two Guy variables called `joe` and `bob` to keep track of each of our instances. Here's what the heap will look like after they're created.



2. We'll set each Guy object's cash and name fields.

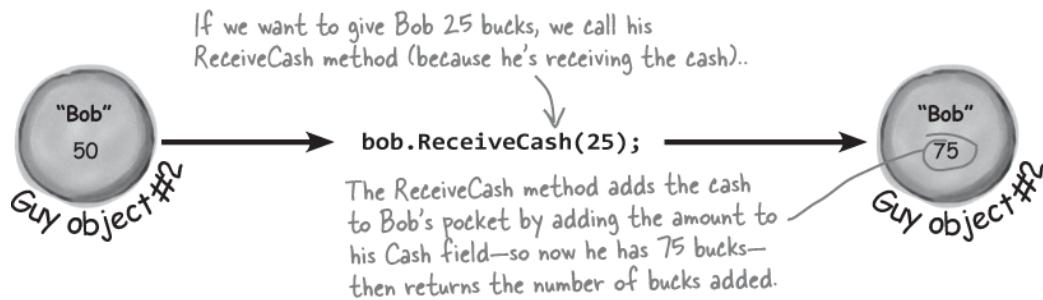
The two objects represent different guys, each with his own name and a different amount of cash in his pocket. Each guy has a Name field

that keeps track of his name, and a Cash field that has the number of bucks in his pocket.



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

We'll make a guy give cash from his pocket (and reduce his Cash field) by calling his GiveCash method, which will return the amount of cash he gave. And 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;    } The Name and Cash fields keep track of the guy's
    public int Cash;       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.

NOTE

Compare the comments in this code to the class diagrams and illustrations of the Guy objects. If something doesn't make sense at first, take the time to really understand it.

There's an easier way to initialize objects with C#

Almost every object that you create needs to be initialized in some way. And 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**. And the IDE's IntelliSense will help you do it.

NOTE

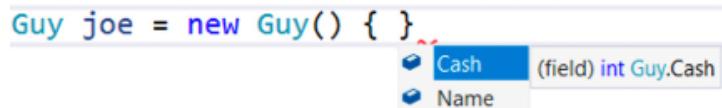
Object initializers save you time and make your code more compact and easier to read... and the IDE helps you write them.

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 `Guy joe = new Guy() {`

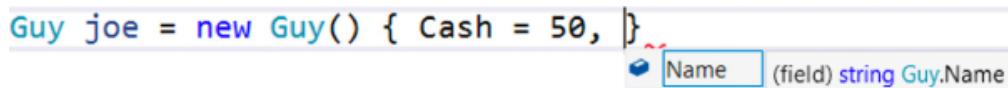
As soon as you add the left curly bracket, the IDE will pop up an IntelliSense window that shows all of the fields that you can initialize:



Choose the Name field and 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.



A screenshot of an IDE showing code completion. The code `Guy joe = new Guy() { Cash = 50, };` is typed. A tooltip appears over the word `Name`, showing the definition: `(field) string Guy.Name`.

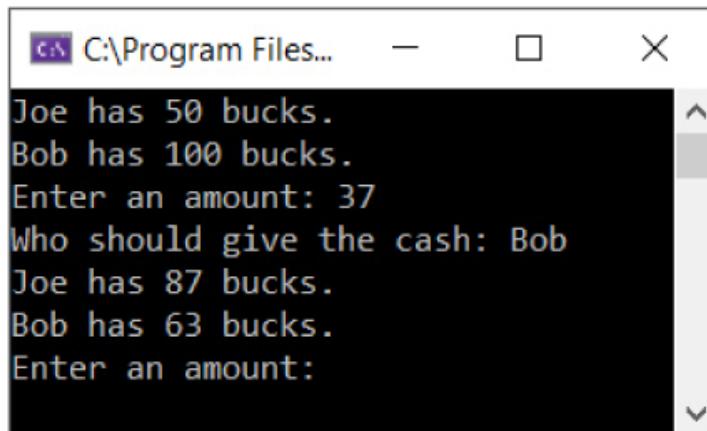
Set the Name field and add the semicolon. You now have a single statement that initializes your object.

```
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.

NOTE

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:



A screenshot of a terminal window titled "C:\Program Files...". The window displays the following text:
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:

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.



EXERCISE

Here's the Main method for a simple 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.

```
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.WriteLine("Enter an amount: ");
        string howMuch = Console.ReadLine();
        if (howMuch == "") return;
        if (int.TryParse(howMuch, out int amount))
        {
            Console.WriteLine("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

NOTE

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 simple betting game. Look closely at this screenshot to see how it works and what it prints to the console.

The screenshot shows a Microsoft Visual Studio Debug Console window. The output is as follows:

```
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.
```

Annotations explain the code:

- "These are the odds to beat." points to "The odds are 0.75".
- "The player makes a double-or-nothing bet each round." points to "How much do you want to bet: 36".
- "The program picks a random double from 0 to 1. If the number is greater than the odds, the player wins twice his bet, otherwise the player loses." points to the sequence of bets and outcomes.

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.
9. The program keeps running while the player has cash.



Bonus
Why or why not?

Sharpen your pencil question: Is Guy really the best name for this class?

.....
.....



EXERCISE SOLUTION

Here's the working Main method for the betting game. Can you think of ways to make it more fun? See if you can figure out how to add additional players, or give different options for odds, or maybe you can think of something more clever. *This is a chance to get creative!*

NOTE

...and to get some practice. Getting practice writing code is the best way to become a great developer.

```
static void Main(string[] args)
{
    double odds = .75;
    Random random = new Random();

    Guy player = new Guy() { Cash = 100, Name = "The player" };

    Console.WriteLine("Welcome to the casino. The odds are " + odds);
    while (player.Cash > 0)
    {
        player.WriteMyInfo();
        Console.Write("How much do you want to bet: ");
        string howMuch = Console.ReadLine();
        if (int.TryParse(howMuch, out int amount))
        {
            int pot = player.GiveCash(amount) * 2;
            if (pot > 0)
            {
                if (random.NextDouble() > odds)
                {
                    int winnings = pot;
                    Console.WriteLine("You win " + winnings);
                    player.ReceiveCash(winnings);
                } else
                {
                    Console.WriteLine("Bad luck, you lose.");
                }
            }
        } else
        {
            Console.WriteLine("Please enter a valid number.");
        }
    }
    Console.WriteLine("The house always wins.");
}
```

NOTE

Was your code a little different? If it still works and produces the right output, that's okay! There are many different ways to write the same program.

NOTE

...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. But remember, it's always okay to look at the solution when you're working on an exercise!



Here's our solution to the bonus
with a different answer?

Sharpen Your pencil question—did you come up

When we used Guy to represent Joe and Bob, the name made sense. But now that it's used for a player in a game, a more descriptive class name like Bettor or Player might be more intuitive.



SHARPEN YOUR PENCIL

Here's a .NET Console App that writes three lines to the console. Your job is to figure out what it writes without using a computer. Start at the first line of the Main method and keep track of the values of each of the fields in the objects as it executes.

```
class Pizzazz
{
    public int Zippo;

    public void Bamboo(int eek)
    {
        Zippo += eek;
    }
}

class Abracadabra
{
    public int Vavavoom;

    public bool Lala(int floq)
    {
        if (floq < Vavavoom)
        {
            Vavavoom += floq;
            return true;
        }
        return false;
    }
}

class Program
{
    public static void Main(string[] args)
    {
        Pizzazz foxtrot = new Pizzazz() { Zippo = 2 };
        foxtrot.Bamboo(foxtrot.Zippo);
        Pizzazz november = new Pizzazz() { Zippo = 3 };
        Abracadabra tango = new Abracadabra() { Vavavoom = 4 };
        while (tango.Lala(november.Zippo))
        {
            november.Zippo *= -1;
            november.Bamboo(tango.Vavavoom);
            foxtrot.Bamboo(november.Zippo);
            tango.Vavavoom -= foxtrot.Zippo;
        }
        Console.WriteLine("november.Zippo = " + november.Zippo);
        Console.WriteLine("foxtrot.Zippo = " + foxtrot.Zippo);
        Console.WriteLine("tango.Vavavoom = " + tango.Vavavoom);
    }
}
```

WHAT DOES THIS PROGRAM WRITE TO THE CONSOLE?

november.Zippo =

foxtrot.Zippo =

tango.Vavavoom =

To find the solution, enter the program into Visual Studio and run it. If you didn't get the answer right, step through the code line by line and add watches for each of the objects' fields.

If you don't want to type the whole thing in, you can download it from

<https://github.com/head-first-csharp/fourth-edition>

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 into the **C# Interactive window** is run immediately. You can open it by choosing C# Interactive from the View >> Other Windows menu. Try opening it and **pasting in code** from the exercise solution. You can run it by typing this and pressing enter:

```
Program.Main(new string[] {})
```

NOTE

You're passing an empty array for the "args" parameter.

The screenshot shows the C# Interactive window running on a Mac. The window title is "C# Interactive (64-bit)". The code being run is:

```
tango.Vavavoom = -1
november.Zippo = 4
foxtrot.Zippo = 8
tango.Vavavoom = -1
> |
```

The output shows the results of the console writes:

```
Macintosh HD — mono --gc-params=nursery-size=64m --clr-memory-mo
Andrews-MacBook-Pro % csi
Microsoft (R) Visual C# Interactive Compiler version 3.4.0-beta3-1
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
> |
```

A callout box points to the first few lines of output with the text: "Paste in each class. You'll see periods for each pasted line." Another callout box points to the last line of output with the text: "Run the Main method to see the output. Press Ctrl-D to exit." A third callout box on the right contains the text: "If you're using a Mac, your IDE may not have a C# Interactive window, but you can run csi from the command line to use the dotnet C# interactive compiler." A fourth callout box at the bottom right contains the text: "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 Mac or Linux, run `csharp` to start the Mono C# Shell. In both cases, you can paste 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`.
- 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.
- 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.
- 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`.
- The IDE makes it easy to 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. The IDE makes it easy to add them.

Part III. Unity Lab 3: GameObject Instances

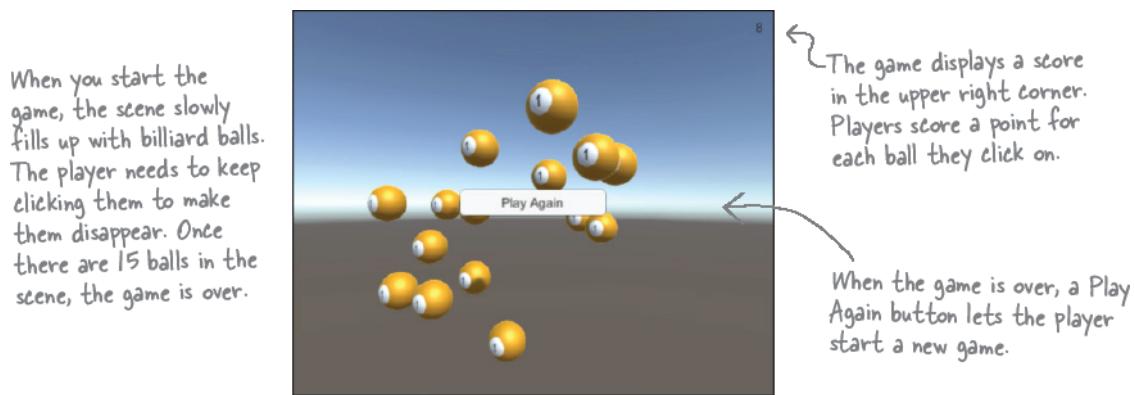
C# is an object oriented language, and since these Head First C# Unity Labs are all **about getting practice writing C# code**, it makes sense that these labs will focus on creating objects.

In [Chapter 3](#) we learned about creating objects in C#, using a class as a template to create instances of it, where each instance gets its own fields. In this Unity Lab you'll **create instances of a Unity GameObject** and use them in a complete, working game.

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 build:

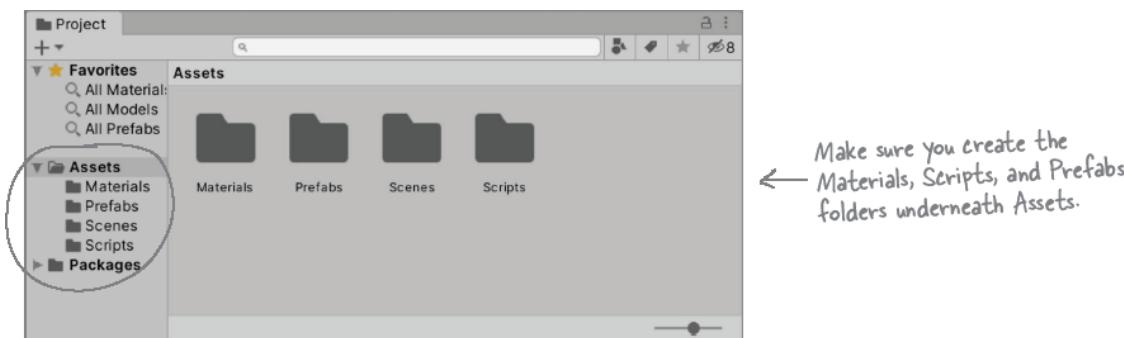


So let's get started. The first thing you'll do is get your Unity project set up. But this time we'll keep the files a little more organized, so you'll create separate folders for your materials and scripts—and one more folder for prefabs (which you'll learn about later in the Lab).

1. Before you begin, close any Unity project that you have open. Also close Visual Studio—you'll let Unity will 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 Lab 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**.

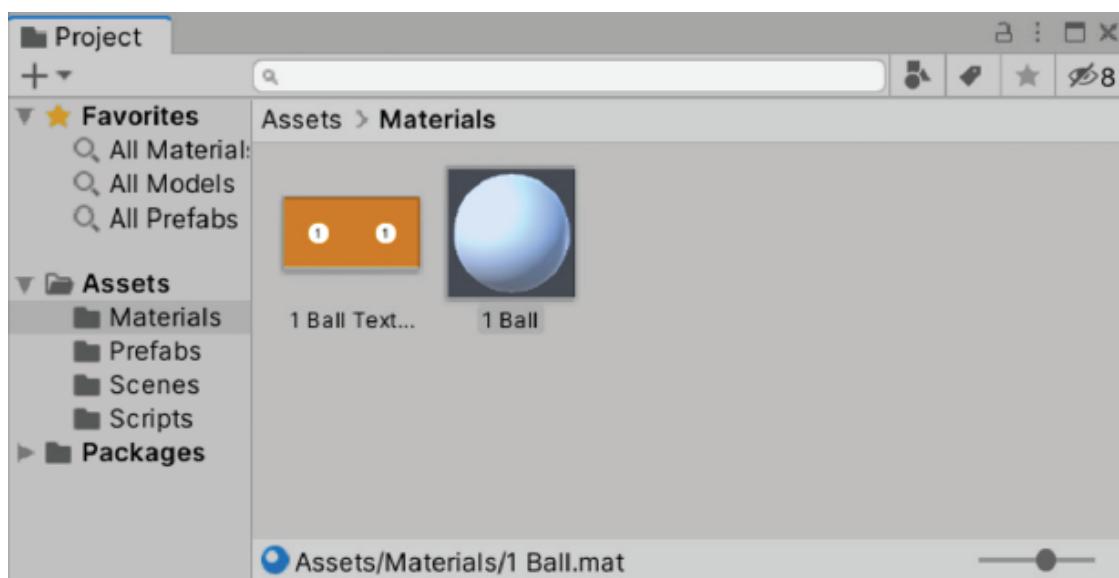


Create a new material inside the Materials folder

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/tree/master/Unity%20Labs/Billiard_Balls 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.

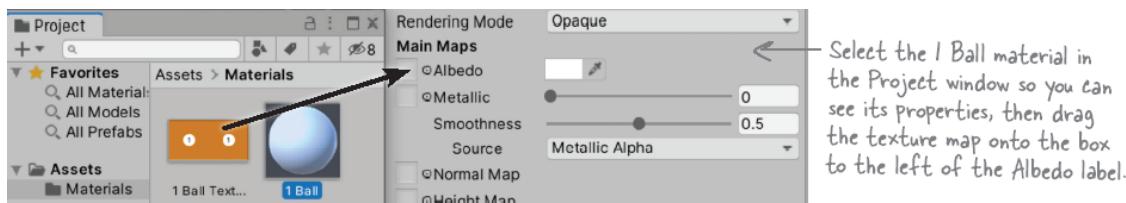
Now you can create the new material. Right-click on the Materials folder in the Project window and **choose Create > Material**. Name your new material **1 Ball**. You should see it appear in the Materials folder in the Project window.



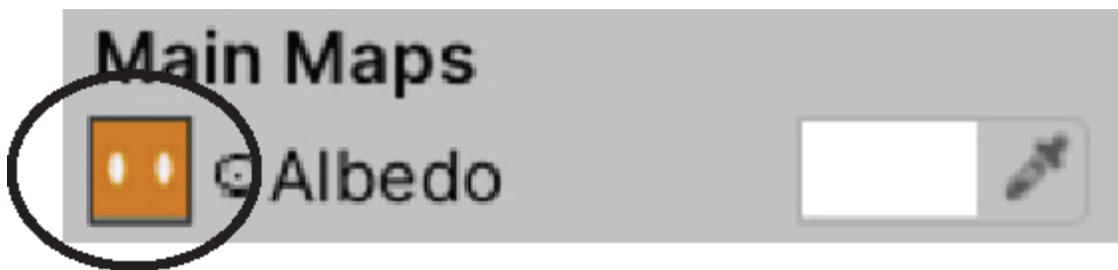
NOTE

In the first Unity Lab you had Unity create a material automatically for you by dragging a texture map file onto a GameObject. 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 PNG image file.

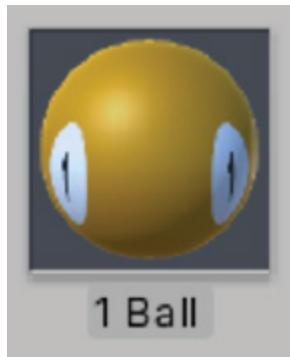
Make sure the 1 Ball material is selected in the Materials window, so it shows up in the Inspector. Click on the 1 Ball Texture file and **drag it into the box to the left of the Albedo label.**



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.



BEHIND THE SCENES

GameObjects reflect light from their surfaces.

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 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 >> 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, **create a new folder under Assets** in the Project window. Name your new folder **Scripts**. Then right-click on the new folder 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 BallBehavior 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()
    {

    }

    // Update is called once per frame
    void Update()
    {
        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 the Debug
    }
}

```

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.

You won't need the Debug DrawRayline, so comment it out.

Now modify the Start method to move the sphere to a random position when it's created. You'll do it 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);
}

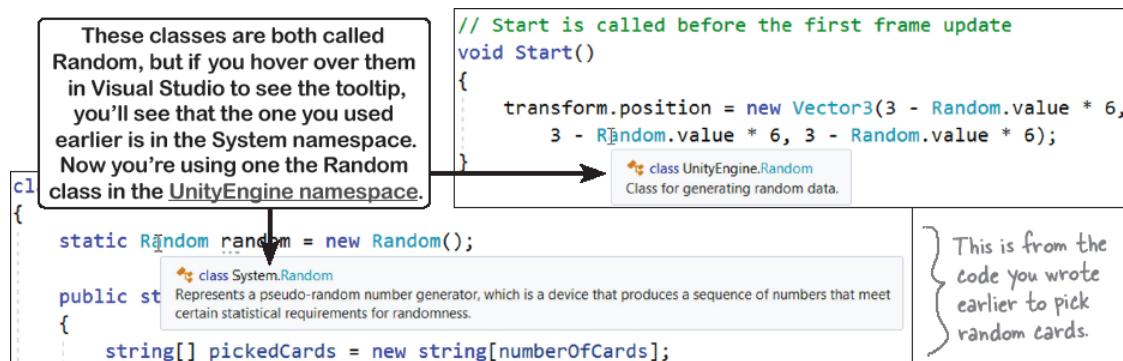
```

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.

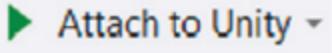
Use the debugger to understand Random.value

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 match game in [Chapter 1](#) and to pick random cards in [Chapter 3](#). But this Random class is different—try hovering over the Random keyword in Visual Studio.



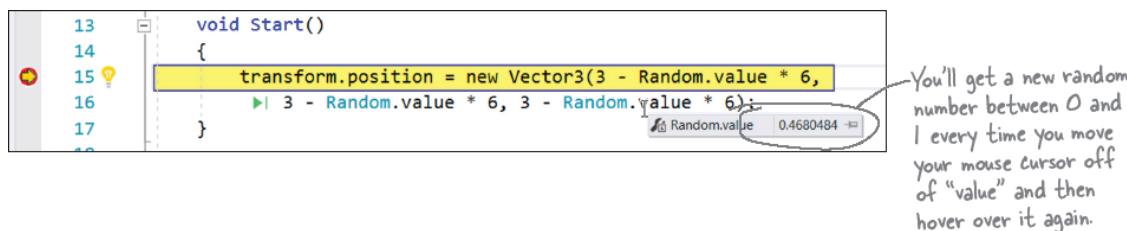
You can see from the code that this new Random class is different than the one you used before. Earlier called `Random.Next()` to get a random value, and that value was a whole number. This new code uses `Random.value`, but that's not a method—it doesn't end with parentheses. (It's actually a property, which you'll learn about in [Chapter 5](#).)

Let's use the Visual Studio debugger to see the kinds of values that this new Random class gives you. Use the Attach to Unity

button ( **Attach to Unity**) to attach Visual Studio to Unity.

Then **add a breakpoint** to the line you added to the Start method.

Now go back to Unity and **start your game**. It should break as soon as you press the play button in Unity. Hover your cursor over Random.value—make sure it's over **value**. Visual Studio will show you its value in a tooltip:



Move your mouse cursor off of **value**, then go back and hover over it 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.

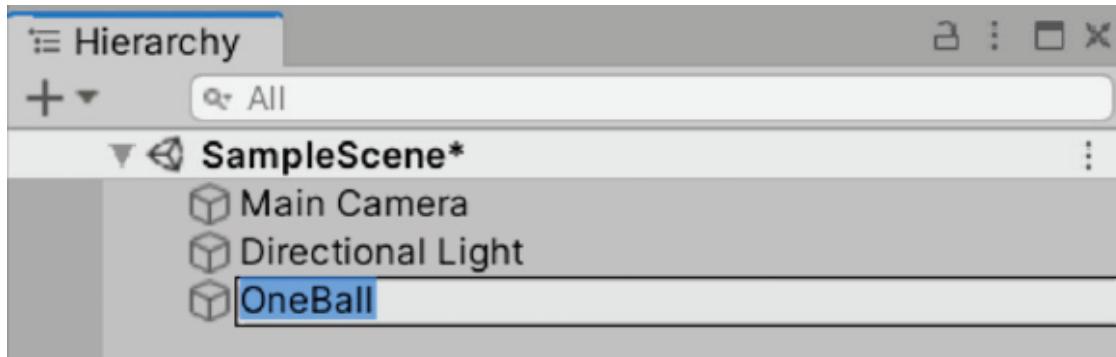
NOTE

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. In [Chapter 2](#) you learned about instances, and how you can create objects by instantiating a class. Unity makes it really easy to take advantage of objects and instances, so you can build games that reuse the same GameObjects over and over again. Let's turn your one ball GameObject into a prefab.

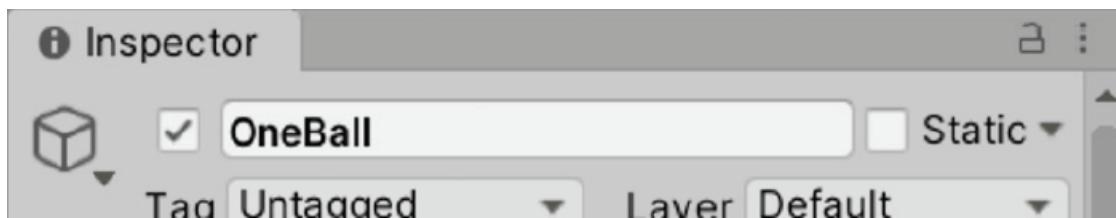
Go to the Hierarchy window, select your Sphere, and rename it by pressing F2 (or right-click and choose Rename). Change the name to OneBall.



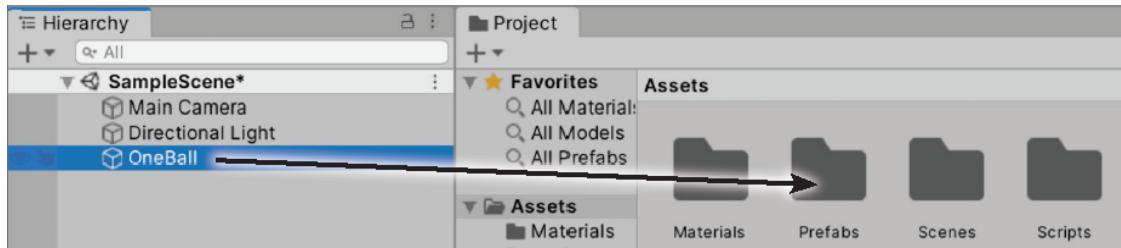
NOTE

If you try to edit your code but find that Visual Studio wouldn't let you make any changes, that means Visual Studio is probably still attached to Unity! Press the square Stop Debugging button to detach it.

You might have noticed that the name changed in the inspector window.



Now we can turn your GameObject into a prefab. Use the Project window to **create a new folder** under Assets called Prefabs, then **drag OneBall from the Hierarchy window into the Prefabs folder**.



OneBall should now appear in your Prefabs folder. And 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 prefab**. You should now only see it in the Project window, and not in the Hierarchy window or the scene.



When a GameObject is
blue in the Hierarchy
window, Unity is telling
you it's a prefab.

NOTE

Have you been saving your scene as you go? Save early, save often!

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). We'll do this by adding a script called GameController.cs to the main camera.

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.

Here's the source code for it:

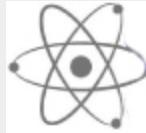
```
public class GameController : MonoBehaviour
{
    public GameObject OneBallPrefab;

    void Start()
    {
        InvokeRepeating("AddABall", 1.5F, 1); ←
    }
    void AddABall() ← This is a method called AddABall. All it
    {           does is create a new instance of a prefab.
        Instantiate(OneBallPrefab); ←
    }
}
```

Why do you think you added the F to the "1.5F" value when you called InvokeRepeating?

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).

You're passing the `OneBallPrefab` field as a parameter to the `Instantiate` method, which Unity will use to create an instance of your prefab.



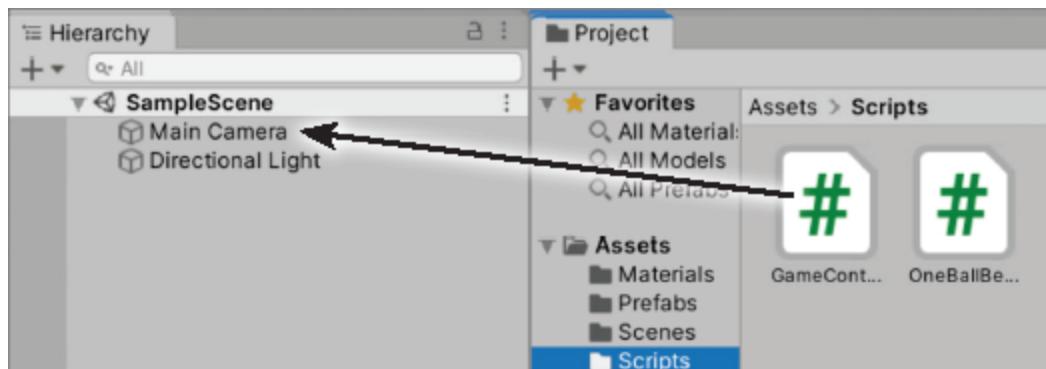
BRAIN POWER

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`). And the Main Camera will always be available in the scene. So... what do you think you'll do with your new `GameController` script?

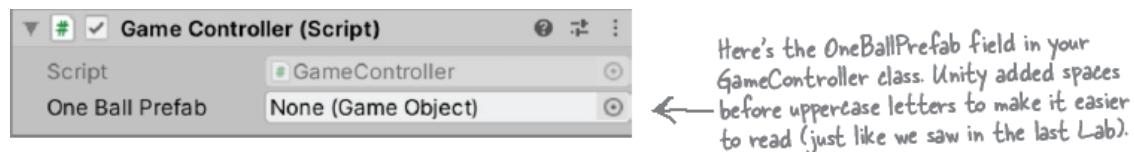
Attach the script to the main camera

Your new `GameController` script needs to be attached to a `GameObject` to run. Luckily, the Main Camera is just another

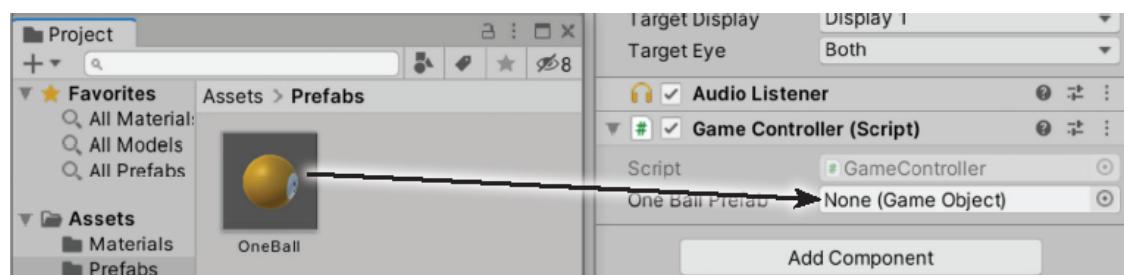
`GameObject`—it happens to be one with a `Camera` component and an `AudioListener` component—so let's attach your new script to it. **Drag your `GameController` script out of the Scripts folder in the Project window and onto the Main Camera in the Hierarchy window.**



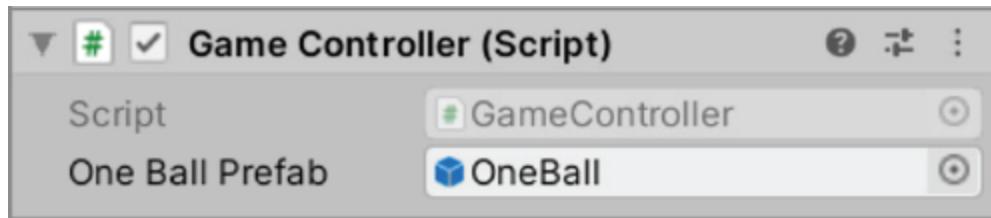
You'll see a component for it, exactly like you would for any other `GameObject`.



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.**



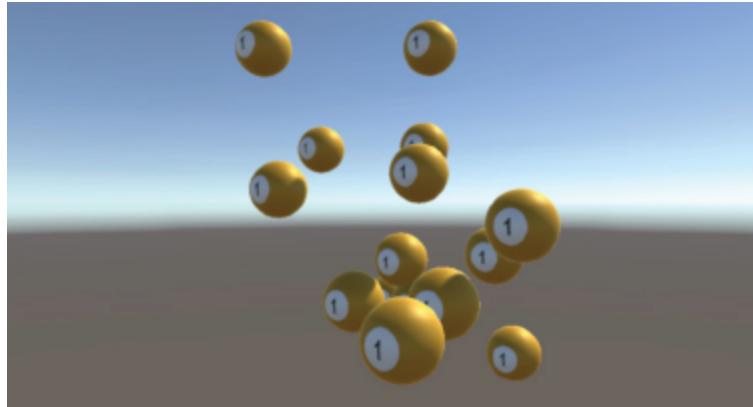
Now the GameController's OneBallPrefab field contains a **reference** to the OneBall prefab:



Now back to the code and look closely the AddABall method. It passes the OneBallPrefab field to Instantiate. 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.

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:



NOTE

Unity calls every GameObject's Update method before each frame. That's called the update loop.

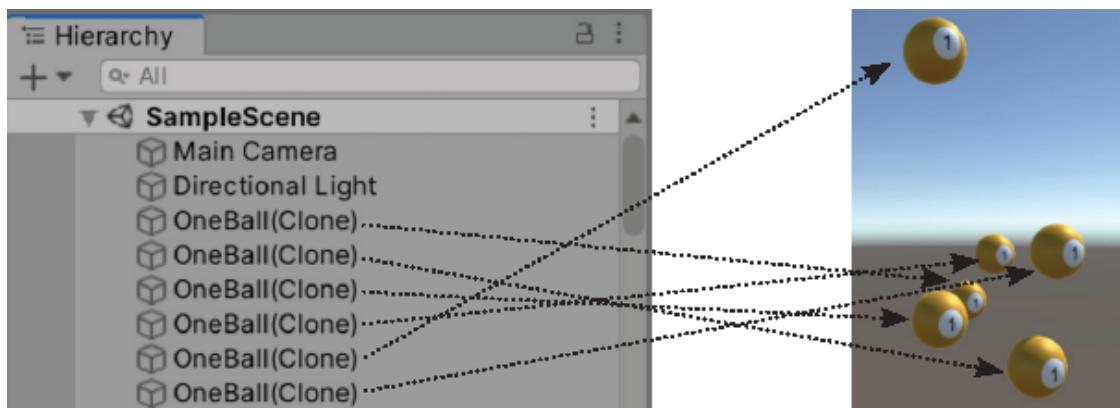
NOTE

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 GameController 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.



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.

NOTE

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.



EXERCISE

Figure out how to add a Ball Number 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:

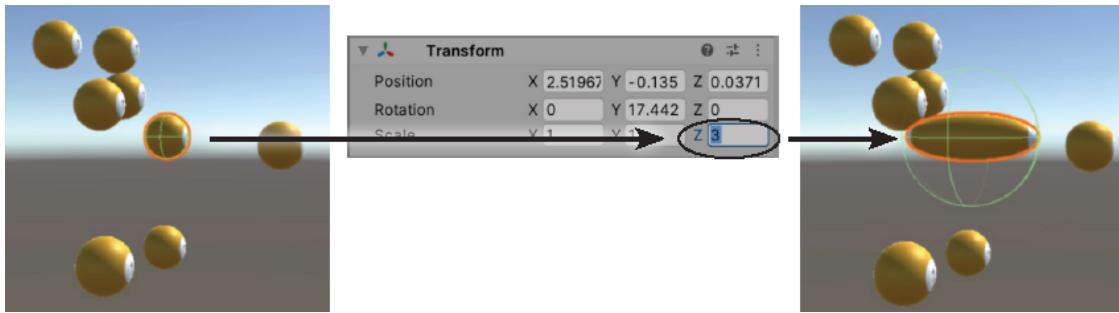
Ball Number

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

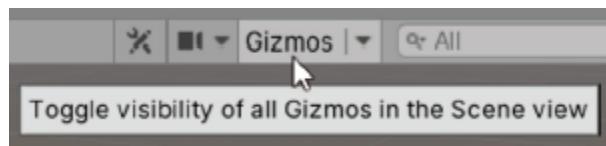
Run your game. Once a few balls have been instantiated, click the pause button—the Unity editor will jump back to the Scene view. Click one of the OneBall instances in the Hierarchy window to select it. The Unity editor will outline it in the Scene window to show you which object you selected. Go to the Transform component in the Inspector window and **set its Z scale value to 3** 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 it 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 add 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;
}
```

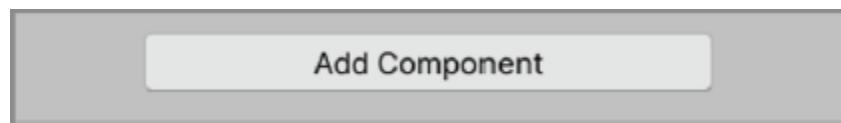
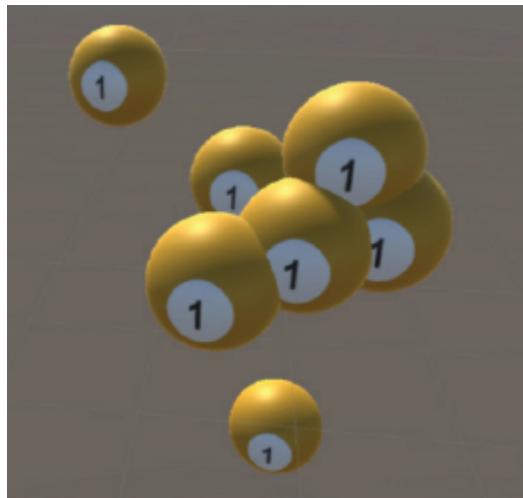
All of the OneBall instances share a single static BallCount field, so the first instance's Start method increments it to 1, the second instance increments BallCount to 2, the third increments it to 3, etc.

Use physics to keep balls from overlapping

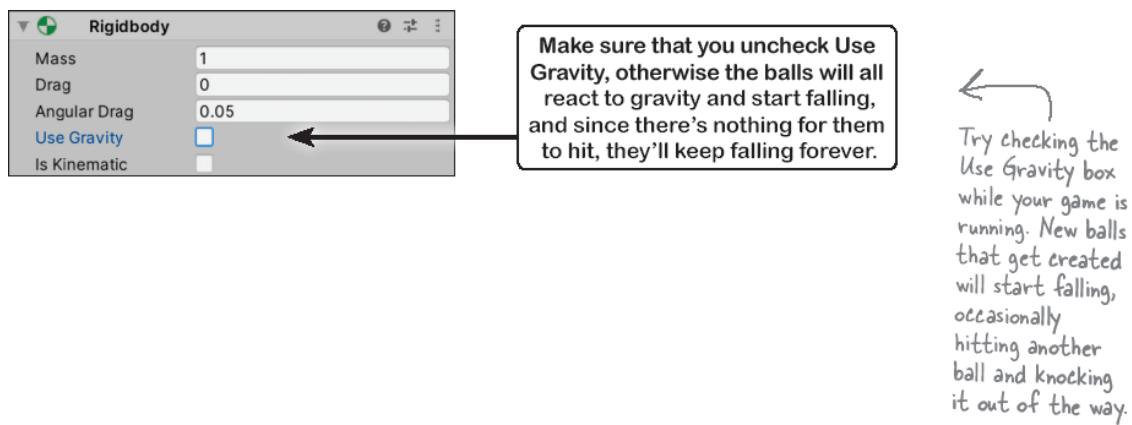
Did you notice that occasionally some of the balls overlap each other?

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. So 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.

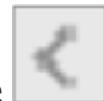
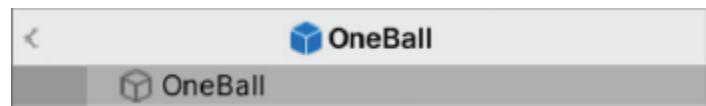


Click the button to pop up the Component window. **Choose Physics** to view the physics components, then **select Rigidbody** to add the component.



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 two balls created. Go to the hierarchy window. If it looks like this:

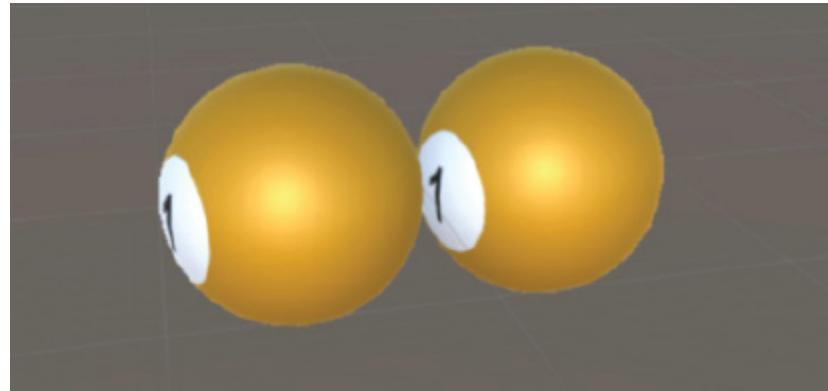


Then you’re editing the prefab—click the button in the top right corner of the Hierarchy window to get back to the scene (you may need to expand SampleScene again).

- Select the first OneBall instance and use its Transform component in the Inspector to set its position to (0, 0, 0).
- Select the second OneBall instance and set its position to (0, 0, 0) as well.
- Select all of the other instances, 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.

NOTE

You can use the Hierarchy window to delete GameObjects from your scene while the game is running.



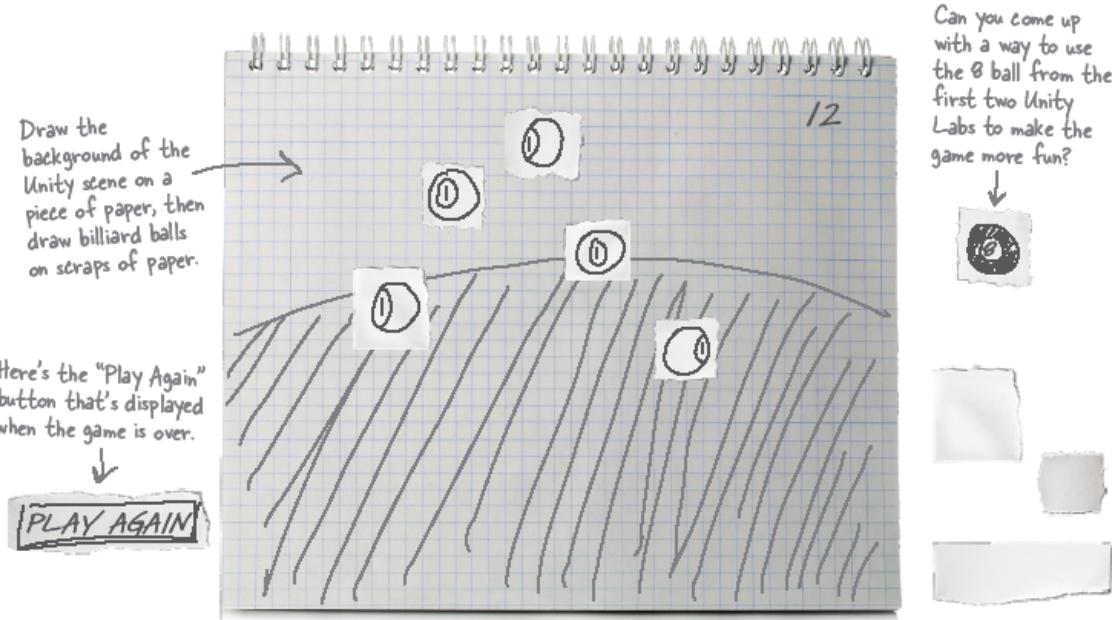
NOTE

**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?





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.

Chapter 4. Types and References: Getting the Reference



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

Ryan is looking to improve his game

Ryan 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

Ryan is a particularly good storyteller, and 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 Ryan 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'll

see about a character. If you wanted to make a class to hold a character sheet, what types would you use for the fields?

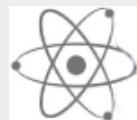
CharacterSheet	
CharacterName	
Level	
PictureFilename	
Alignment	
CharacterClass	
Strength	
Dexterity	
Intelligence	
Wisdom	
Charisma	
SpellSavingThrow	
PoisonSavingThrow	
MagicWandSavingThrow	
ArrowSavingThrow	
<hr/>	
ClearSheet	
GenerateRandomScores	

Character Sheet

<u>ELLIWYNN</u>	
Character Name	7
Level	
<u>LAWFUL GOOD</u>	
Alignment	
<u>WIZARD</u>	
Charcater Class	
<input type="text" value="9"/>	Strength
<input type="text" value="11"/>	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.



BRAIN POWER

Look at the fields in the CharacterSheet class diagram. What type would you use for each field?

A variable's type determines what kind of data it can store

There are many **types** built into C#, and you'll use them to store many different kinds of data. You've already seen some of the most common ones, like int, string, bool, and float. But there are a few that you haven't seen, and they can really come in handy, too.

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 Ryan 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.

Here are some types you'll use a lot.



- `int` can store any **whole** number from $-2,147,483,648$ to $2,147,483,647$. Whole numbers don't have decimal points.



- `float` can store **real** numbers from $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ with up to 8 significant figures.

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

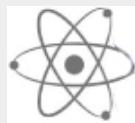
- `string` can hold text of any length (including the empty string `""`).



- `double` can store **real** numbers from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ with up to 16 significant figures. Double is really common when you're working with XAML properties.



- `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.



BRAIN POWER

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

C# has several types for whole numbers

C# has several different types for whole numbers. This may seem a little weird. Why have so many types for numbers without decimals? For most of the programs in this book, it won't matter if we use an int or a long. But if you're writing a program that has to keep track of millions and millions of whole number values, then choosing a smaller whole number type like byte instead of a bigger type like long can save you a lot of memory.

- `byte` can store any **whole** number between 0 and 255.
- `sbyte` can store any **whole** number from -128 to 127.
- `short` can store any **whole** number from -32,768 to 32,767.
- `long` can store any **whole** number from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

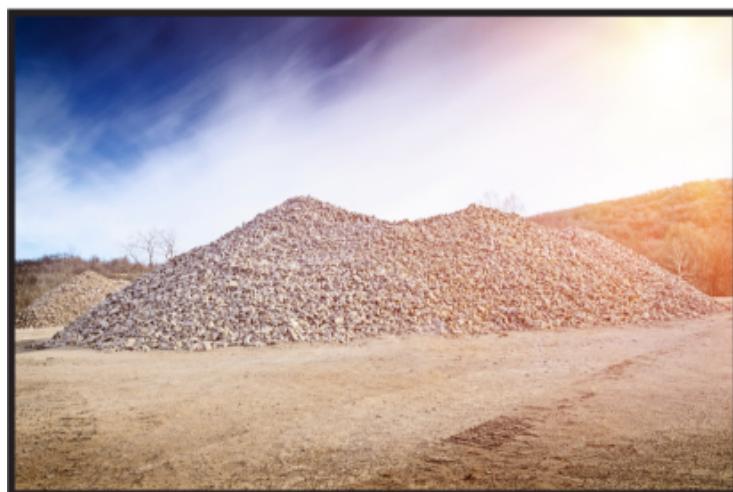


Byte only stores
small whole numbers
from 0 to 255.

If you need to store a larger number, you can use a short, which stores whole numbers from -32768 to 32767



Long also stores whole numbers, 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, and the “s” in sbyte stands for signed.

And 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 seven significant figures just isn’t precise enough. And, 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 multiple **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 7 significant digits.
- double can store any number from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ with 15-16 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.

NOTE

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 range may look complicated, but it’s actually pretty straightforward. “Significant figures” represents the precision of the number: 1,048,415, 104.8415, and .0000001048415 all have 7 significant figures. So when 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 eight digits followed by 30 zeroes, or as small as 37 zeroes followed by eight digits.

NOTE

If it’s been a while since you’ve used exponents, 3.4×10^{38} means 34 followed by 37 zeroes, and -1.5×10^{-45} is $-0.00000\dots(44\text{ zeroes})...00015$.

Float and double can also have special values, including both positive and negative zero, positive and negative infinity, and a special value called **NaN (not-a-number)** that represents, well, a value that isn’t a number at all. They also have static methods that let you test for those special values. Try running this loop:

```
for (float f = 10; float.IsFinite(f); f *= f)
{
    Console.WriteLine(f);
}
```

Now try that same loop with double:

```
for (double d = 10; double.IsFinite(d); d *= d)
{
    Console.WriteLine(d);
}
```

Let’s talk about strings

You’ve written code that works with **strings**. But 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 `Random` class you used). 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 we'll learn a lot more about later in the book). Sometimes you need to store a single character like Q or j or \$, and when you do you'll use the `char` type. Literal values for `char` are always inside single quotes ('x', '3'). You can include **escape sequences** in the quotes, too ('\\n' is a line break, '\\t' is a tab). You can write an escape sequence in your C# code using two characters, but your program stores each escape sequence as a single character in memory.

And finally, there's one more important type: `object`. If a variable has `object` as its type, ***you can assign any value to it***. The `object` keyword is also an alias—it's the same as `System.Object`.



SHARPEN YOUR PENCIL

Sometimes you declare a variable and set its value in a single statement like this: `int i = 37;` but you already know that you don't have to set a value. So what happens if you use the variable without assigning it a value? Let's find out! Use the **C# Interactive window** (or the .NET console if you're using a Mac) to declare a variable and check its value. 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.

The screenshot shows the C# Interactive window with the following code and output:

```
0 ..... We wrote in the first answer for you.
..... int i;
..... long l;
..... float f;
..... double d;
..... decimal m;
..... byte b;
..... char c;
..... string s;
```

The output window shows the declaration of variables `i`, `l`, `f`, `d`, `m`, `b`, `c`, and `s`. The value `0` is displayed next to `i`, indicating its default value.

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";
```

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

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

Use suffixes to give your literals types

When you added statements like this in Unity, you may have wondered about the F:

```
InvokeRepeating("AddABall", 1.5F, 1);
```

NOTE

C# assumes that a whole number literal without a suffix (like 371) is an int, and one with a decimal point (like 27.4) is a double.

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.



SHARPEN YOUR PENCIL SOLUTION

```
0 int i;  
0 long l;  
0 float f;
```

```
0 double d;  
0 decimal m;  
0 byte b;  
'\0' char c; ←  
null string s;
```

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 keywords 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. Here's a list of keywords, write down what you think they do in C#.

namespace	
for	
class	
else	
new	
using	
if	
while	

NOTE

If you really want to use reserved keywords as variable names, you can put @ in front of it, but that's as close as the compiler will let you get to the reserved keyword. You can also do that with nonreserved names too, if you want to.



SHARPEN YOUR PENCIL SOLUTION

C# has dozens of **reserved words called *keywords***. They're keywords 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. Here's a list of keywords, write down what you think they do 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 Avis true.

A variable is like a data to-go cup

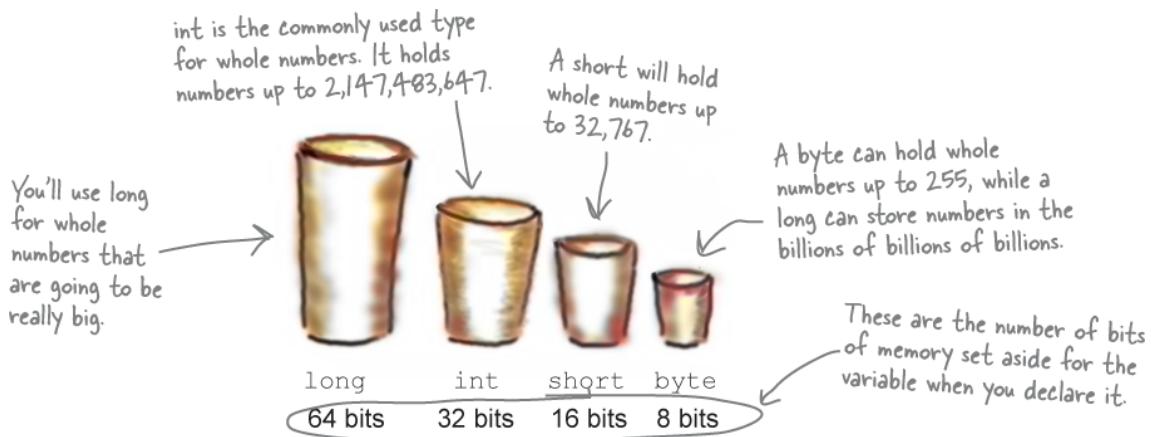
All of your data takes up space in memory. (Remember the heap from the last chapter?) So part of your job is to think about how *much* space you're going to need whenever you use a string or a number in your

program. That's one of the reasons you use variables. They let you set aside enough space in memory to store your data.

NOTE

Not all data ends up on the heap. Value types usually keep their data in another part of memory called the stack. You'll learn all about that later in the book.

Think of a variable like a cup that you keep your data in. C# uses a bunch of different kinds of cups to hold different kinds of data. And just like the different sizes of cups at the coffee shop, there are different sizes of variables, too.



Convert this!

Use the Convert class to explore bits and bytes

You've always heard that programming is about 1's and 0's. .NET has a **static Convert class** in .NET that converts between different numeric data types. Let's use it to see an example of how the bytes and bytes work with whole numbers.

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 convert binary numbers to bytes:

```
Convert.ToByte("10111", 2) // returns 23  
Convert.ToByte("11111111", 2); // returns 255
```

The first argument of the 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 it uses 8 bits of memory—an 8-bit number is a binary number between 0 and 11111111 binary (or 0 and 255 decimal).

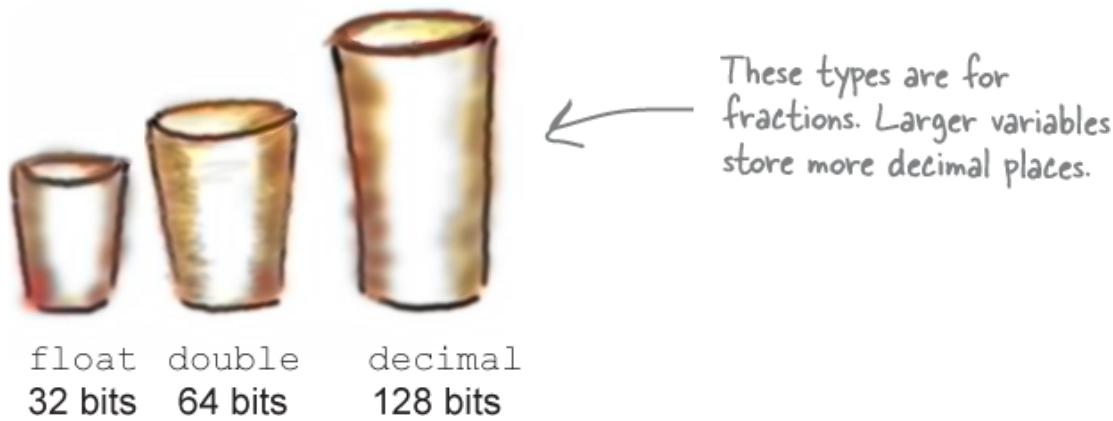
A short is a 16-bit value. Let's use Convert.ToInt16 convert the binary value 11111111111111 (15 1's) to a short. And an int is a 32-bit value, so we'll use Convert.ToInt32 to convert the 31 1's to an int:

```
Convert.ToInt16("11111111111111", 2); // returns 32767  
Convert.ToInt32("1111111111111111111111111111", 2); //  
returns 2147483647
```

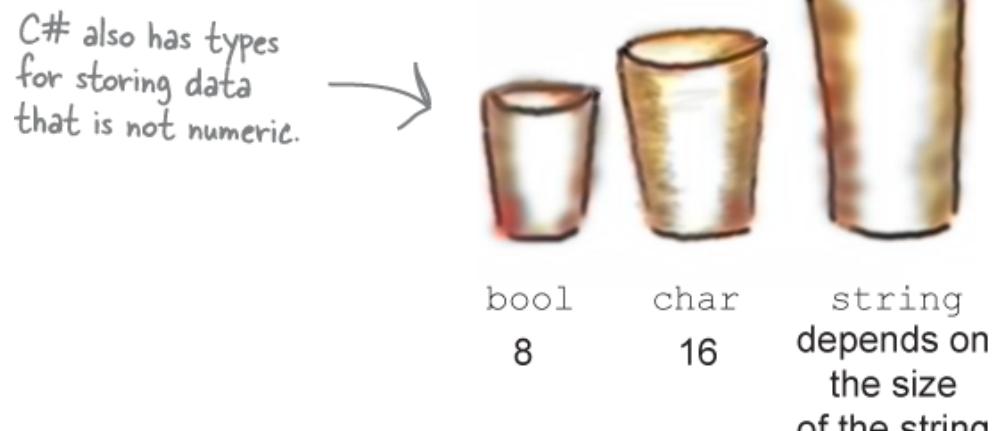
Other types come in different sizes, too

Numbers that have decimal places are stored differently than whole numbers, 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**. And 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, either. 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.



Strings can be big... REALLY big! C# uses a 32-bit integer to keep track of the string length, so the maximum string length is 2^{31} (or 2147483648) characters.

NOTE

The different floating-point types take up different amounts of memory:
float is smallest, and double is largest.

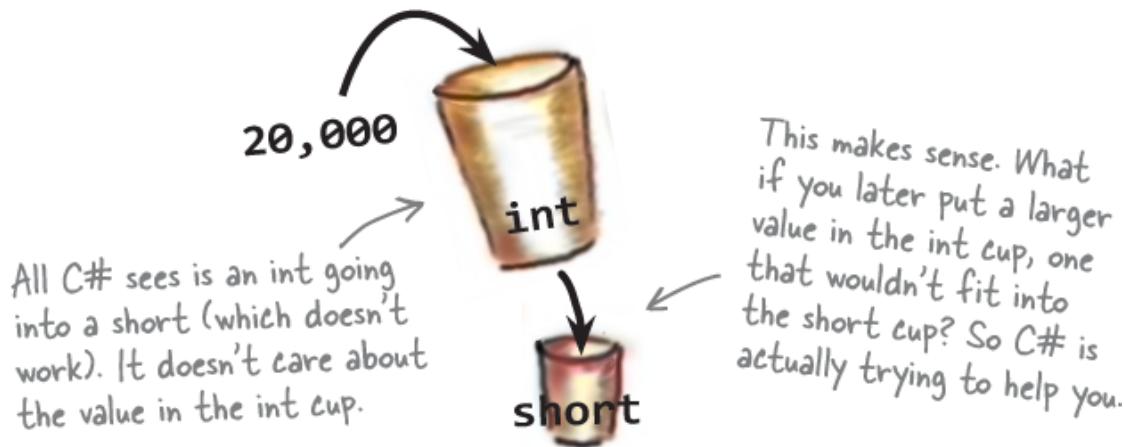
10 pounds of data in a 5-pound bag



When you declare your variable as one type, the C# compiler **allocates** (or reserves) all of the memory it would need to store the maximum value of that type. Even if the value is nowhere near the upper boundary of the type you've declared, the compiler will see the cup it's in, not the number inside. So this won't work:

```
int leaguesUnderTheSea = 20000;  
short smallerLeagues = leaguesUnderTheSea;
```

20,000 would fit into a `short`, no problem. But since `leaguesUnderTheSea` is declared as an `int`, C# sees it as `int`-sized and considers it too big to put in a `short` container. But 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.





SHARPEN YOUR PENCIL

Three of these statements won't build, either because they're trying to cram too much data into a small variable or because they're putting the wrong type of data in. Circle them and write a brief explanation of what's wrong.

```
int hours = 24;  
  
short y = 78000;  
  
bool isDone = yes;  
  
short RPM = 33;  
  
int balance = 345667 - 567;  
  
string taunt = "your mother";  
  
byte days = 365;  
  
long radius = 3;  
  
char initial = 'S';  
  
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 whole number.

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 making a promise to the compiler that you know the types are different, and 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. But that doesn't mean the **value** stays intact through the casting. If you cast an `int` variable that's set to 365 to a `byte` variable, 365 is too big for the `byte`. But instead of giving you an error, the value will just **wrap around**: for example, 256 cast to a `byte` will have a value of 0. 257 would be converted to 1, 258 to 2, etc., up to 365, which will end up being **109**. And 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';

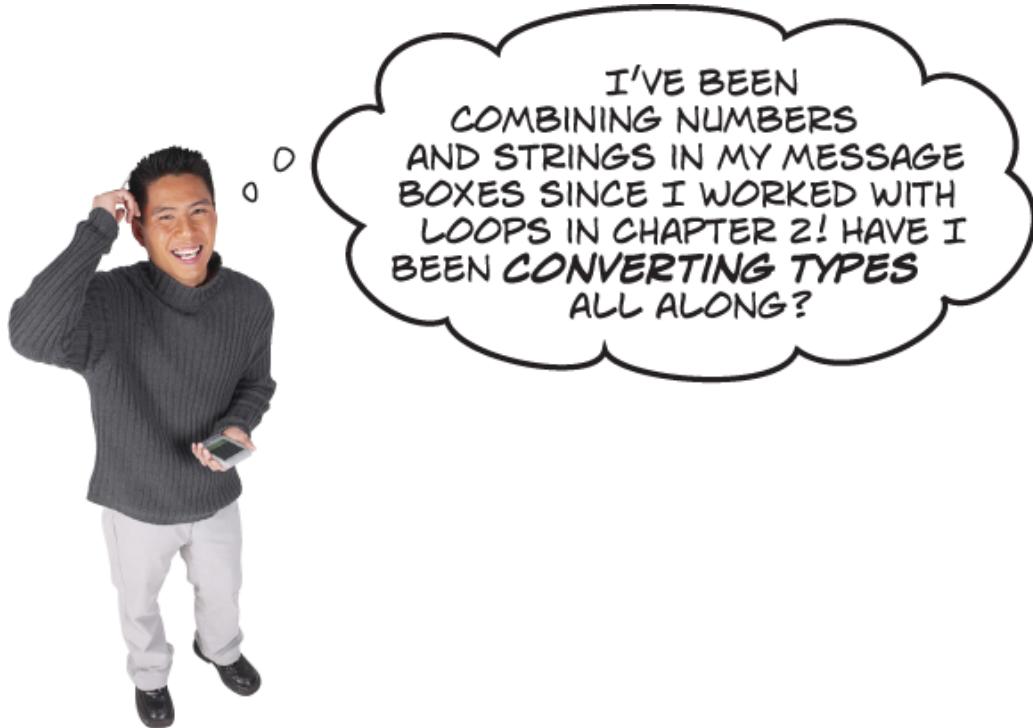
myString = (string)myChar;

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>



Yes! When you concatenate strings, C# converts values.

When you use the + operator to combine a string with another value, it's called **concatenation**. And when you concatenate a string with an int, bool, float, or another value type, it automatically converts the value. But 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.)



SHARPEN YOUR PENCIL SOLUTION

You can't always cast any type to any other type. Create a new Console App project and type these statements into its Main method. Then build your program—it will give lots of errors. Cross out the ones that give errors. That'll help you figure out which types can be cast, and which can't!

```
int myInt = 10;  
  
byte myByte = (byte)myInt;  
  
double myDouble = (double)myByte;  
bool myBool = (bool)myDouble;  
  
string myString = "false";  
myBool = (bool)myString;  
myString = (string)myInt;  
  
myString = myInt.ToString();  
myBool = (bool)myByte;  
myByte = (byte)myBool;  
  
short myShort = (short)myInt;  
  
char myChar = 'x';  
myString = (string)myChar;  
  
long myLong = (long)myInt;  
  
decimal myDecimal = (decimal)  
myLong;  
  
myString = myString + myInt +  
myByte + myDouble + myChar;
```

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.

C# does some conversion 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
d = d / 123.456;          the = operator converted
Console.WriteLine("The answer is " + d);      the result to a double.
```

The other way C# converts types for you automatically is when you use the + operator to **concatenate** strings (which just means sticking one string on the end of another, like you've been doing with message boxes). When you use + to concatenate a string with something that's another type, it automatically converts the numbers to strings for you. Here's an example—try adding these lines to any C# program. The first two lines are fine, but the third one won't compile.

```
long number = 139401930;
string text = "Player score: " + number;
text = number;
```

The C# compiler gives you this error on the third line:

 CS0029 Cannot implicitly convert type 'long' to 'string'

ScoreText.text is a string field, so when you used the + operator to concatenate a string it assigned the value just fine. But when you try to assign x to it directly, it doesn't have a way to automatically convert the long value to a string. But you can convert it to a string really easily 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 whole number values. Can you convert whole number 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, and used it 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'

NOTE

When the compiler gives you an “invalid argument” error, it means that you tried to call a method with variables whose types didn’t match the method’s parameters.

Sometimes C# can do the conversion automatically. It doesn’t know how to convert a double to an int (like converting 1.0 to 1), but it does know how to convert an int to a double (like converting 1 to 1.0).

- The C# compiler knows how convert a whole number to a floating-point.
- And it knows how to convert a whole number type to another whole number 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 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. And by the way, a lot of people have trouble remembering **the difference between parameter and argument**. So just to be clear:

A parameter is what you define in your method. An argument is what you pass to it. You can pass a byte argument to a method with an int parameter.

THERE ARE NO DUMB QUESTIONS

Q: That last if statement only said `if (add3)`. Is that the same thing as `if (add3 == true)`?

A: Yes. Let's take another look at that if/else statement:

```
if (add3)
    value += 3;
else
    value -= 2;
```

An if statement always checks if something's true. So because the type of the add3 variable is Boolean, it evaluates to either true or false, which means we didn't have to explicitly include `== true`.

You 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 the 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 (like `short` to `int`) that C# knows how to **convert** automatically (an implicit conversion). 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—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.
- C# can do some conversions automatically, like `int` to `double` or `float` to `double`.
- 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.

Ryan is constantly improving his game

Good game masters are dedicated to creating the best experience they can for their players. Ryan'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.



THE STANDARD RULES
FOR THIS GAME ARE A GOOD
STARTING POINT, BUT I BET WE
CAN DO BETTER.

When players fill out their character sheets at the start of the game, they follow these steps to calculate each of the ability scores for their character.



ABILITY SCORE FORMULA

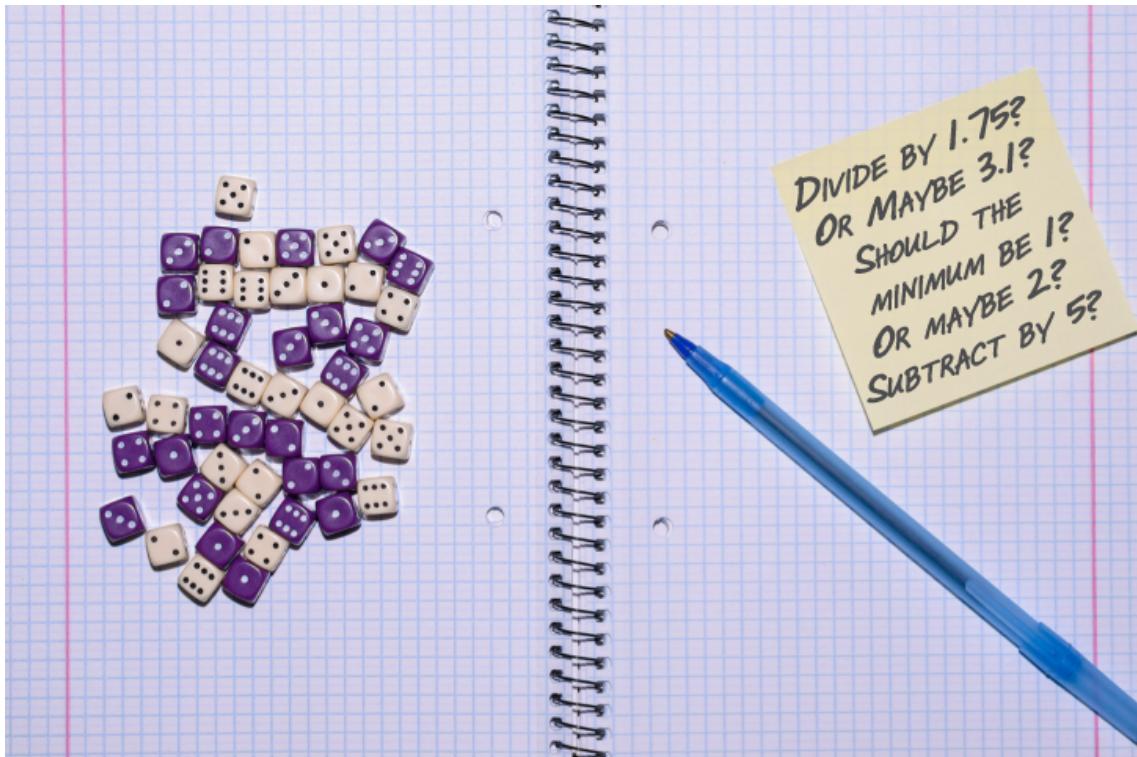
A "4d6 ROLL" means
rolling four normal
6-sided dice and
adding up the results.



- * START WITH A 4d6 ROLL TO GET A NUMBER BETWEEN 4 AND 24
- * DIVIDE THE ROLL RESULT BY 1.75
- * ADD 4 TO THE RESULT OF THAT DIVISION
- * ROUND DOWN TO THE NEAREST WHOLE NUMBER
- * IF THE RESULT IS TOO SMALL, USE THE MINIMUM VALUE OF 3

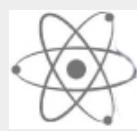
Let's help Ryan experiment with ability scores

Ryan'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.



Ryan 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.





BRAIN POWER

What can we do to help Ryan find the best combination of values for an updated ability score formula?



SHARPEN YOUR PENCIL

We've built a class to help Ryan 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.

See if you can spot the problem without typing the class into your IDE. Can you find the broken line that will cause a compiler error?

```
class AbilityScoreCalculator
{
    public int Roll = 14;
    public double DivideBy = 1.75;
    public int AddAmount = 2;
    public int Minimum = 3;
    public int Score;

    public void CalculateAbilityScore()
    {
        // Divide the starting roll
        double divided = Roll / DivideBy;

        // Add to the result
        int added = AddAmount += divided;

        // Make sure it's greater than the minimum
        if (added < Minimum)
        {
            Score = Minimum;
        } else
        {
            Score = added;
        }
    }
}
```

After you **circle the line of code that has problems**, write down the problems that you found with it.

.....

.....



EXERCISE

Let's build a console app to test the AbilityScoreCalculator class. Create a new Console App project and add the AbilityScoreCalculator class from the "Sharpen your pencil" exercise—it won't build properly yet..In this exercise, first you'll fix the class, then you'll create program that uses it.

1. Fix the problems in the line of code that you circled in the "Sharpen your pencil" exercise.
2. Add the Main method. Everything in it should be familiar—except for one method that it uses, `Console.ReadKey`:

```
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` type.

NOTE

You'll use a single instance of `AbilityScoreCalculator`, using the user input to update its fields so it remembers a new default value for the next iteration of the while loop.

Here's the full Main method—add it to your program:

```
static void Main(string[] args)
{
    AbilityScoreCalculator calculator = new
AbilityScoreCalculator();
    while (true)
    {
        calculator.Roll = ReadInt(calculator.Roll, "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;
```

```
    }  
}
```

3. Implement the ReadInt method, which 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 it can parse it, 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 try to parse it  
    // If it can be parsed, write " using value" + value to the  
    // console  
    // Otherwise write " using default value" + lastUsedValue to  
    // the console  
}
```

4. 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.



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.Write(prompt + " [" + lastUsedValue + "]: ");
    string line = Console.ReadLine();
    if (int.TryParse(line, out int value))
    {
        Console.WriteLine("    using value " + value);
        return value;
    } else
    {
        Console.WriteLine("    using default value " + lastUsedValue);
        return lastUsedValue;
    }
}

static double ReadDouble(double lastUsedValue, string prompt)
{
    Console.Write(prompt + " [" + lastUsedValue + "]: ");
    string line = Console.ReadLine();
    if (double.TryParse(line, out double value)) ← Here's the call to double.TryParse,
    {                                            which works exactly like the int
        Console.WriteLine("    using value " + value); version except that you need to use
        return value;                           double as the output variable type.
    }
    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 saves the values the user entered and uses them as the default values in the next iteration.

Use casting to fix the problematic line of code in AbilityScoreCalculator

If you tried adding the AbilityScoreCalculator class to the project without making any changes, you saw a “Cannot implicitly convert type” error on this line:

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

It caused that error because `AddAmount += divided` returns a double value, which can't be assigned to the int variable subtracted. You can fix it by casting divided to an int, so adding it to AddAmount returns another int.

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

This line of code now compiles—but there's still a problem! Can you spot it?

NOTE

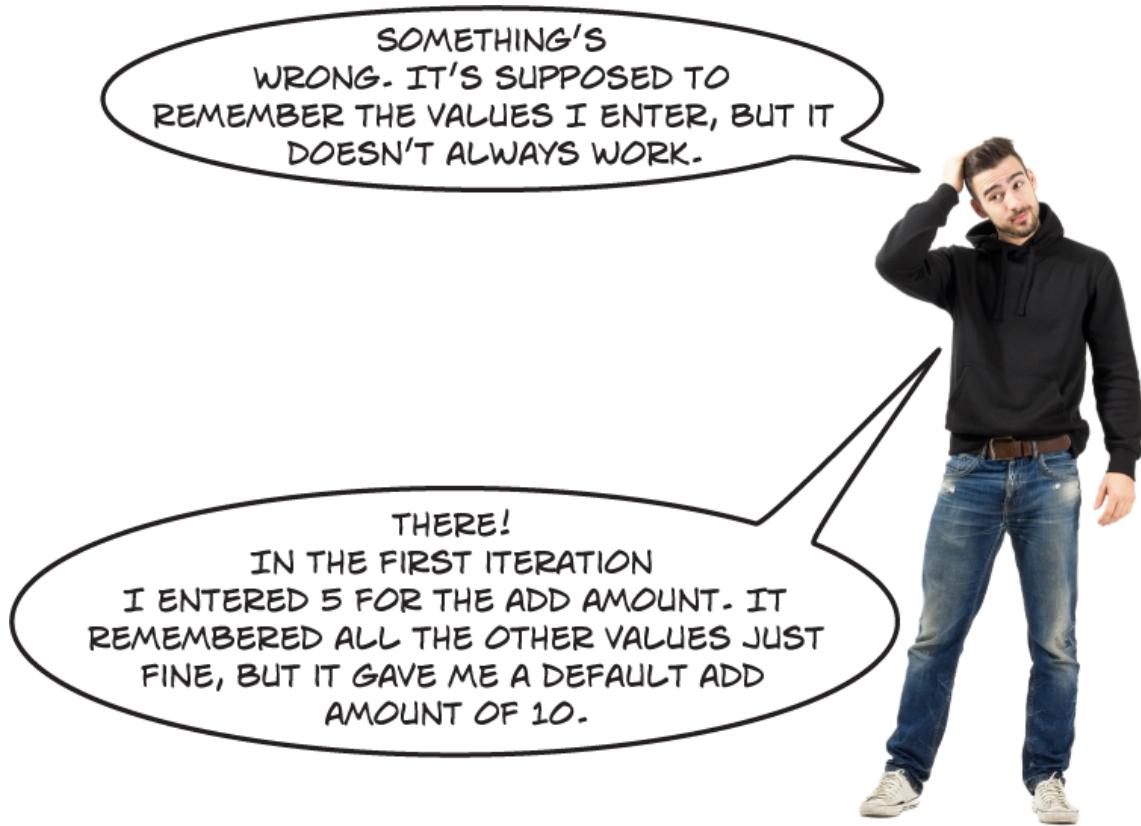
Looks like we can't answer the "Sharpen your pencil" exercise just yet!

Here's the output from the program.

```
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]: 3.5
    using default value 3.5
Add amount [10]: 7 ←
    using value 7
Minimum [3]:
    using default value 3
Calculated ability score: 12
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
    using default value 18
Divide by [3.5]:
    using default value 3.5
Add amount [12]: 4
    using value 4
Minimum [3]:
    using default value 3
Calculated ability score: 9
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
    using default value 18
Divide by [3.5]:
    using default value 3.5
Add amount [9]: ←
    using default value 9
Minimum [3]:
    using default value 3
Calculated ability score: 14
Press Q to quit, any other key to continue
```

That's strange.
Ryan entered 5
for the previous
add amount, but
the program is
giving him 10 as
a default option.

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



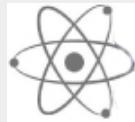
You're right, Ryan. There's a bug in the code.

Ryan 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 Ryan 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 Ryan entered 5, but it gave

him 10 as a default option. He entered 7, but it says he default 12.
What's going on?



BRAIN POWER

What steps can you take to track down the bug in the ability score calculator app?



SLEUTH IT OUT

When you're debugging code, you're acting like a **code detective**. Something is causing the bug, so your job is to identify suspects and retrace their steps. Let's do an investigation and see if we can apprehend the culprit, Sherlock Holmes style.

The problem seems to be isolated to the "add amount" value, so let's start by looking for any line of code that touches the AddAmount field. Here's a line in the Main method that uses the AddAmount field—put a breakpoint on it:

```
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. Okay, that's probably not the issue—you can disable or remove that breakpoint.

Continue running your program. When the breakpoint in AbilityScoreCalculator.CalculateAbilityScore hits, **add a watch for AddAmount**. According to Ryan'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...

Two side-by-side screenshots of the Visual Studio Watch window. Both windows have the search bar set to 'Search Depth: 3'. The left window shows a single entry for 'AddAmount' with a value of '2' and type 'int'. The right window shows the same entry, but its value has been updated to '10', also with type 'int'. A red circle highlights the value '2' in the first window, and another red circle highlights the value '10' in the second window.

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 a whole number, then add it to AddAmount and store the result in added. But it also has an unexpected side-effect: it's updating AddAmount with the sum. And the reason is that **the statement uses the += operator**, which returns the sum but assigns the sum to AddAmount.

And now we can finally fix Ryan'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. It's elementary.



SHARPEN YOUR PENCIL SOLUTION

NOTE

Now that we've found the problem, we can finally answer the "Sharpen your pencil" exercise.

We've built a class to help Ryan 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 equal 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, because "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.

For a binary operator, 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.

NOTE

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 **assignment operator = also returns a value**. You can do this:

```
int q = (a = b + c)
```

Which will calculate `a = b + c` as usual. But since the `=` operator returns a value **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 increased by `divided`, but also 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);
```

Then both `first` and `second` will end up equal to 4. Open up a Console App and use the debugger. 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");
```

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 zeroes between the 3 and 4.). Now it prints `They're equal` again. So apparently `0.1D` plus `0.2D` equals `0.3000000000000004D`.

NOTE

Wait what?!



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 0.0100000000000002 , 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). But 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 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. And you saw another rule earlier about wrapping for whole number 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. But now you’re talking about remainders. What’s the difference?

A: Sure. 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: 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: Ryan’s formula had me dividing two values and then rounding the result down to the nearest whole number. 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 whole numbers, so your program needs to do *something* to the decimal part of the number.

So C# has a simple 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.

NOTE

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.

NOTE

The $0.1D + 0.2D \neq 0.3D$ 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, read this 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>

NOTE

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!

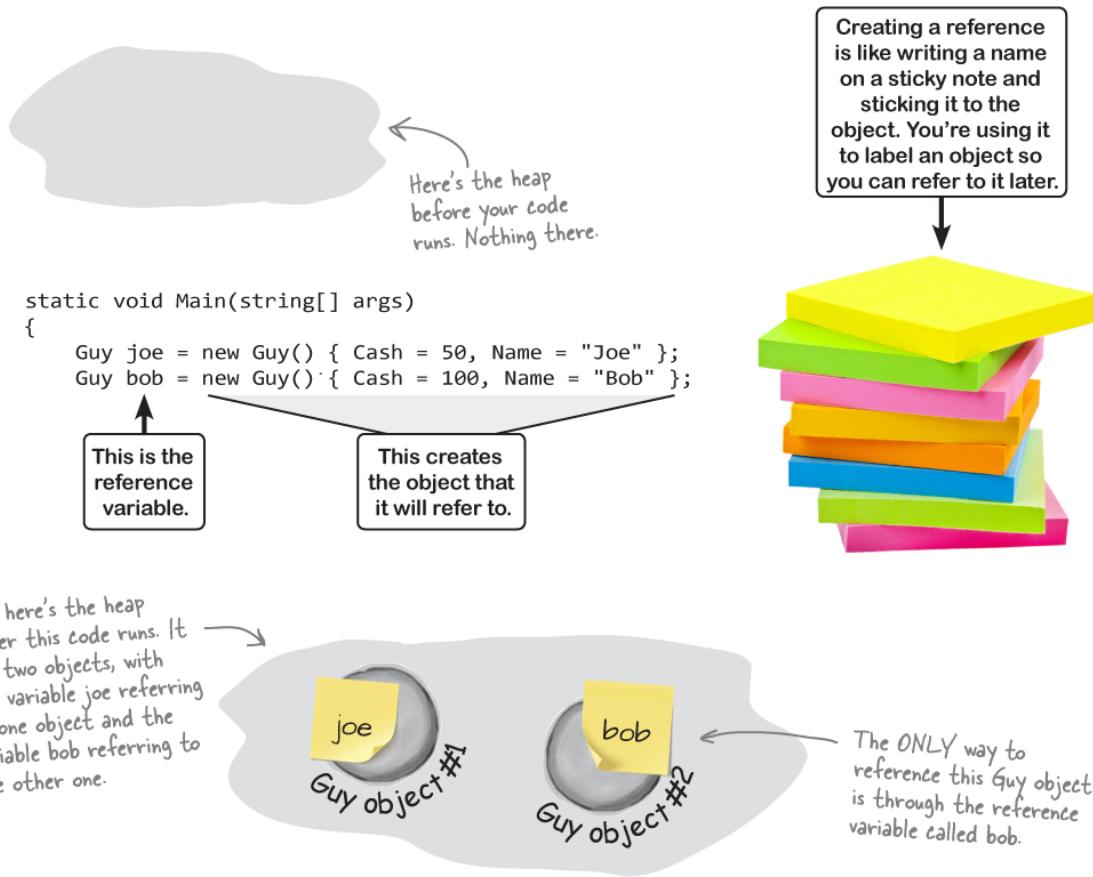
Use reference variables to access your objects

When you create a new object, you use a `new` statement to instantiate, like `new Guy()` in your program the end of the last chapter—the `new` statement created a new Guy object on the heap. But 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`.

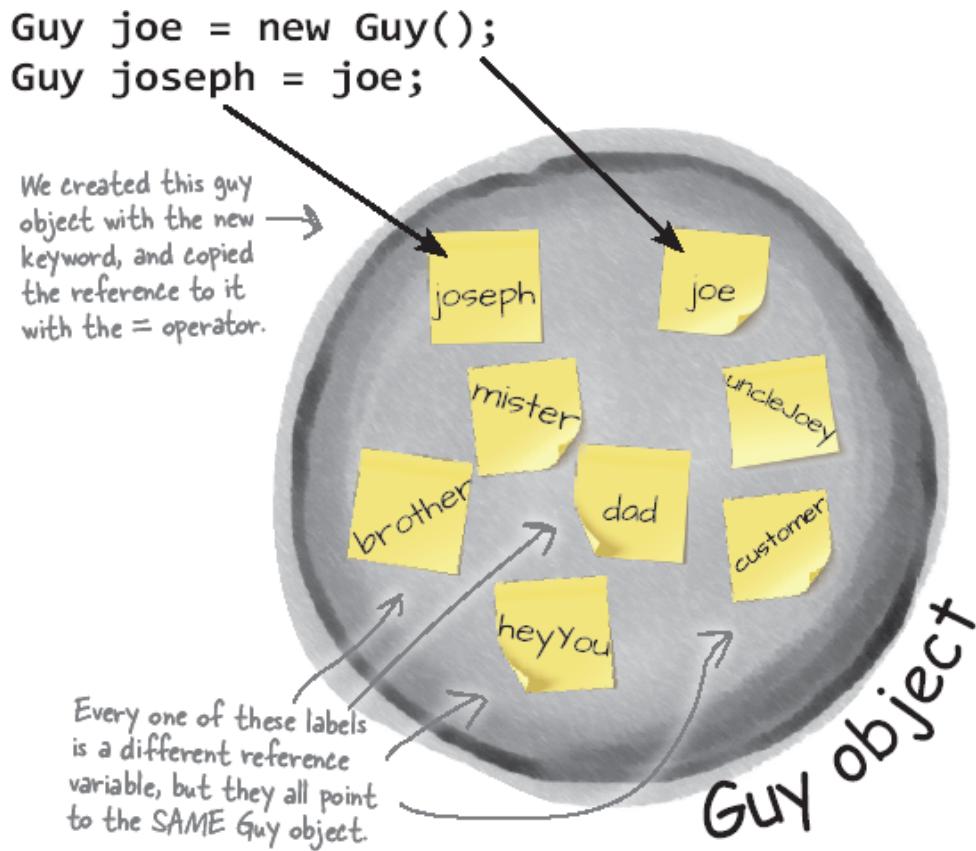
When you have a variable that's an object type, it's a reference variable: a reference to a particular object. Let's just make sure we get the

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 the labels changed, 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.



NOTE

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 him 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). But if you want to write code that works with an object, you need a reference

to that object. If you don't have that reference, you have no way to access the object.

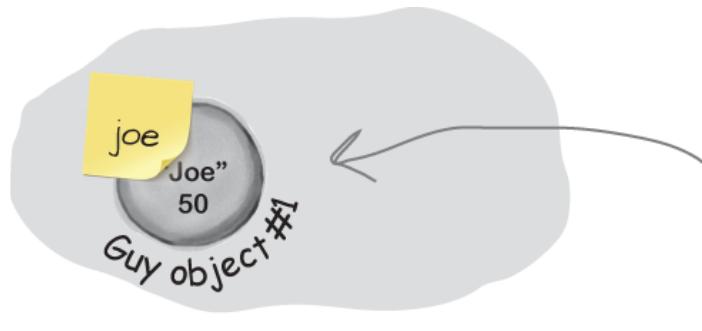
If there aren't any more references, your object gets garbage-collected

If all of the labels come off of an object, programs can no longer access that object. That means C# can mark the object for **garbage collection**. That's when C# gets rid of any unreferenced objects and reclaims the memory those objects took up for your program's use.

1. Here's some code that creates an object.

And 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" };
```



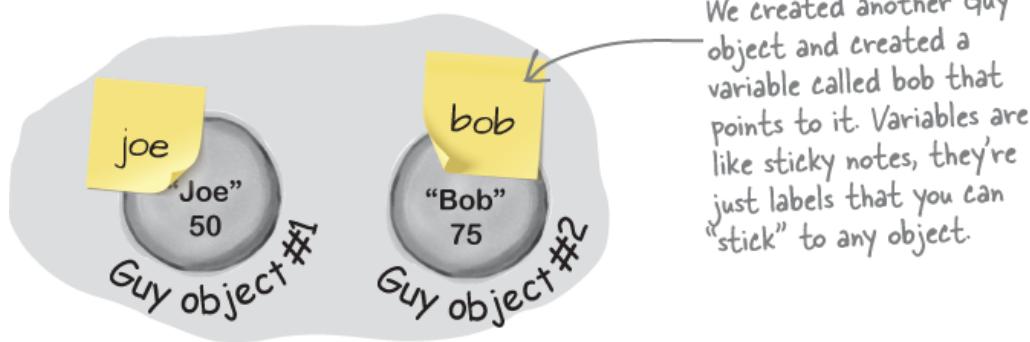
We used an object initializer to create this Guy object. Its Name field has the string "Joe" and Cash field has the int 50, and we put a reference to the object in a variable called `joe`.

2. Now let's create our second object.

Now 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" };
```



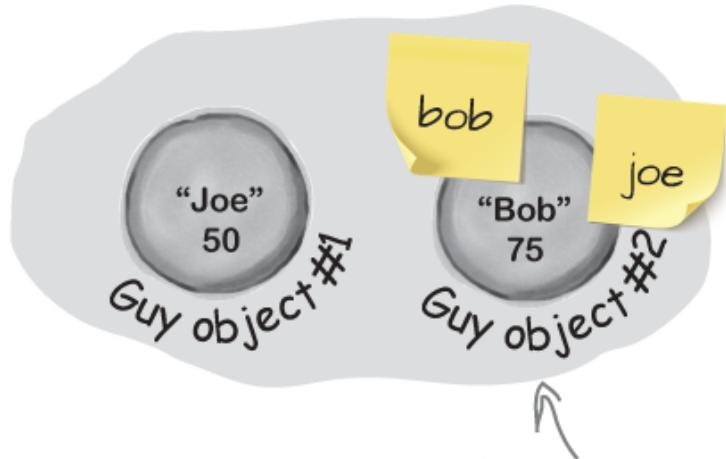
3. Let's take the reference to the first Guy object and change it to point to the second Guy object.

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**.

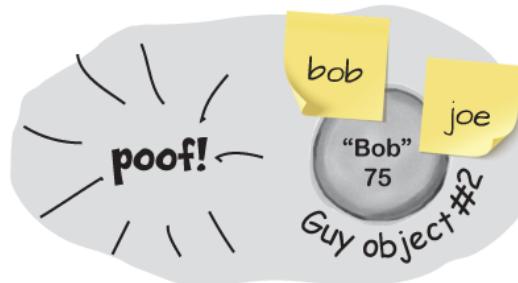


After the CLR removes the last reference to the object, it marks it for garbage collection.

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!



NOTE

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.

Multiple references and their side effects

you can pet the dog in head first c#

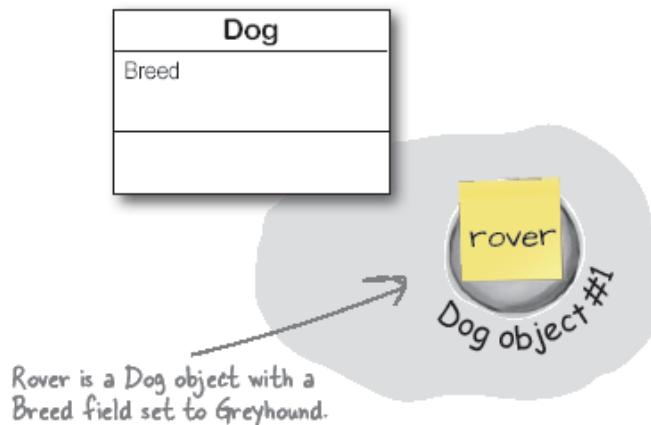
```
public partial class Dog {  
    public void GetPet() {  
        Console.WriteLine("Woof!");  
    }  
}
```

You've got to be careful when you start moving around reference variables. Lots of times, it might seem like you're simply pointing a variable to a different object. But 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

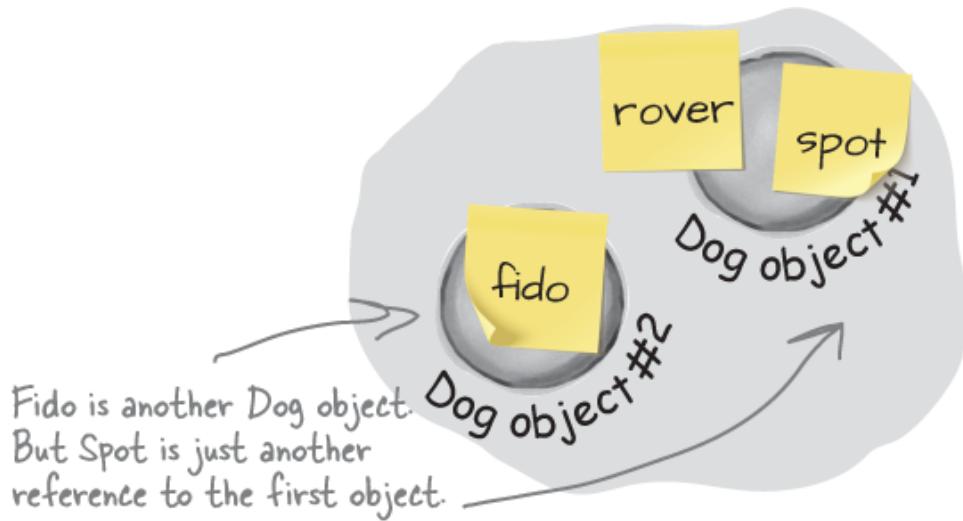
References: 1



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

Objects: 2

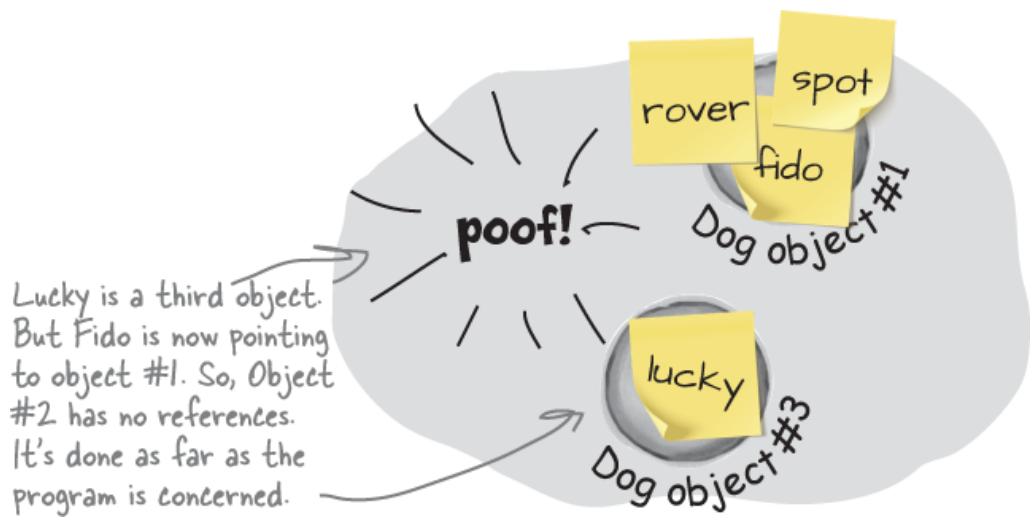
References: 3



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

Objects: 2

References: 4





SHARPEN YOUR PENCIL

Now it's your turn. Here's one long block of code. Figure out how many objects and references there are at each stage. On the right-hand side, draw a picture of the objects and sticky notes in the heap.

```
1. Dog rover = new Dog();
   rover.Breed = "Greyhound";
   Dog rinTinTin = new Dog();
   Dog fido = new Dog();
   Dog greta = fido;
```

Objects:_____

References:_____

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

Objects:_____

References:_____

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

Objects:_____

References:_____

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

Objects:_____

References:_____

5. **charlie** = laverne;
 lucky = rinTinTin;

Objects:_____

References:_____

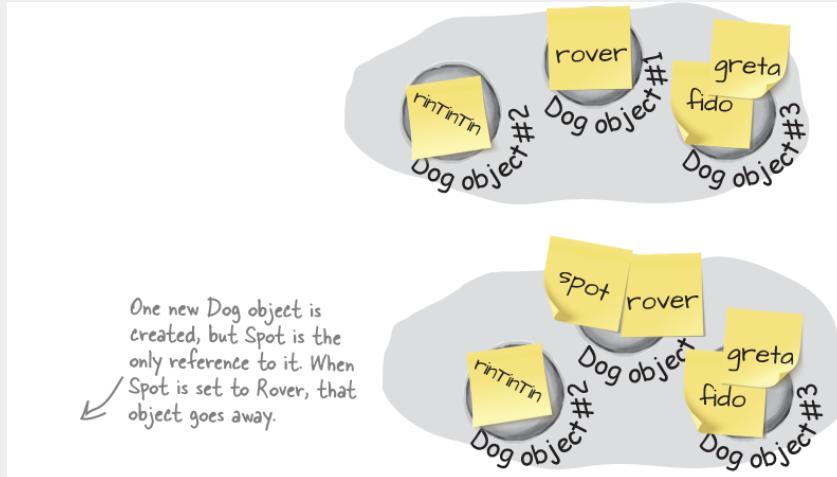


SHARPEN YOUR PENCIL SOLUTION

```
1. Dog rover = new Dog();  
rover.Breed = "Greyhound";  
Dog rinTinTin = new Dog();  
Dog fido = new Dog();  
Dog greta = fido;
```

Objects: 3

References: 4



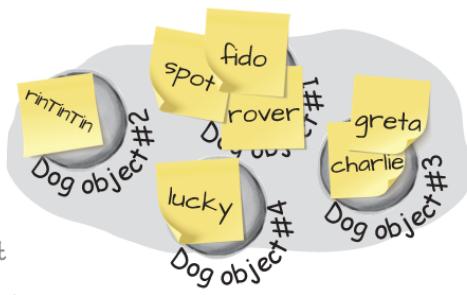
```
2. Dog spot = new Dog();  
spot.Breed = "Dachshund";  
spot = rover;
```

Objects: 3

References: 5

Here a new Dog object is created, but when Fido is set to Rover, Fido's object from #1 goes away.

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



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

Objects: 4

References: 7

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

Objects: 4

References: 8

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

When Rin Tin Tin moved to Lucky's object, the old Rin Tin Tin object disappeared.



```
5. charlie = laverne;
   lucky = rinTinTin;
```

Objects: 4

References: 8

Here the references move around, but no new objects are created. And setting Lucky to Rin Tin 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 we 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 file system 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 know it at the time, but I've been making your job of learning C# a whole lot easier.

Head First: You mentioned Windows binaries. But what if I'm running .NET programs on Mac and Linux? Are you doing the same thing for those OSes?

CLR: If you're using a Mac 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.



EXERCISE

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.

Elephant

Name
EarSize

WhoAmI

NOTE

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....



EXERCISE

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:

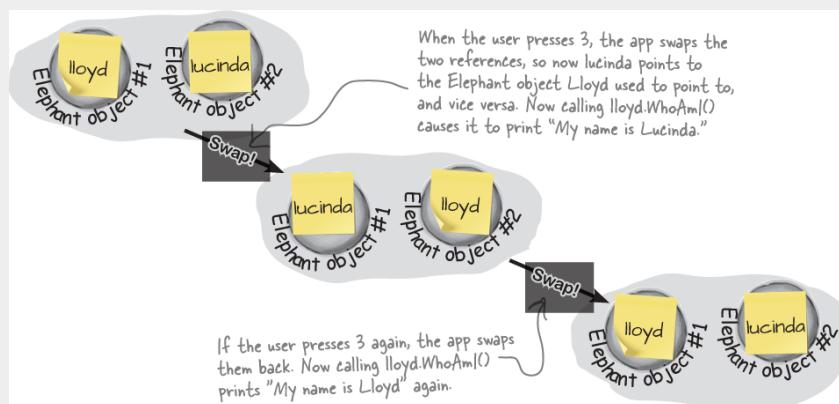
```
Elephant lucinda = new Elephant() { Name =
    "Lucinda", EarSize = 33 };
Elephant lloyd = new Elephant() { Name =
    "Lloyd", EarSize = 40 };
```

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 first 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.

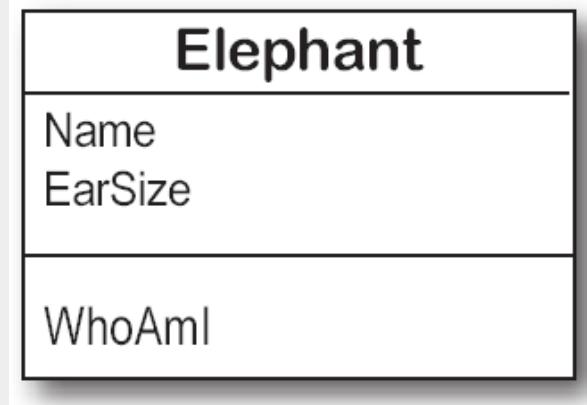




EXERCISE SOLUTION

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");
        }
        else return;
        Console.WriteLine();
    }
}

```

If you just point Lloyd to Lucinda, there won't be any more references pointing to Lloyd, and his object will be lost. That's why you need to have an extra variable (we called it "holder") to keep track of the Lloyd object reference until Lucinda can get there.

There's no "new" statement when we declare the holder variable because we don't want to create another instance of Elephant.

Two references mean TWO variables that can change the same object's data

Besides losing all the references to an object, when you have multiple references to an object, you can unintentionally change an 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?

```

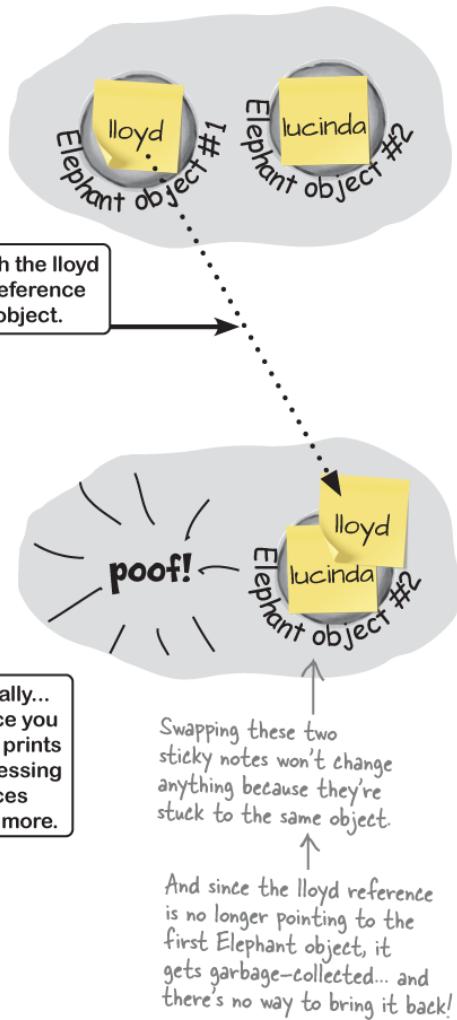
else if (input == '3')
{
    Elephant holder;
    holder = lloyd;
    lloyd = lucinda;
    lucinda = holder;
    Console.WriteLine("References have been swapped");
}
else if (input == '4')
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
else
{
    return;
}

```

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.

The program acts normally... until you press 4. But once you do, pressing either 1 or 2 prints the same output—and pressing 3 to swap the references doesn't do anything any more.



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. And swapping them makes no difference because you're swapping two identical references.

Objects use references to talk to each other

Elephant	
Name	
EarSize	
WhoAmI	
HearMessage	
SpeakTo	

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**. And 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 others' methods.

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

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

Do this!

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

Here's what it looks like when it's called:

```
lloyd.HearMessage("Hi", lucinda);
```

We called Lloyd's TellMe 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 parameter of whatever elephant was passed into TellMe using its second parameter.

2. Add a method that lets an elephant send a message.

Now let's add this SpeakTos method to the Elephant class. It uses a special keyword: `this`. That's a reference that **lets an object get a reference to itself**.

```
public void SpeakTo(Elephant whoToTalkTo, string message) {  
    whoToTalkTo.HearMessage(message, this);  
}
```

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

Let's take a closer look at what's going on.

When we call the Lucinda object's SpeakTo method:

```
lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
```

It calls the Lloyd object's HearMessage method like this:

```
whoToTalkTo.HearMessage("Hi, Lloyd!", this);
    ↓
    Lucinda uses whoToTalkTo
    (which has a reference to
    Lloyd) to call HearMessage.
    ↓
    [a reference to Lloyd].HearMessage("Hi, Lloyd!", [a reference to Lucinda]);
```

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;
}
```

NOTE

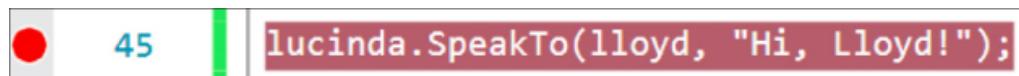
The “this” keyword lets an object get a reference to itself.

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

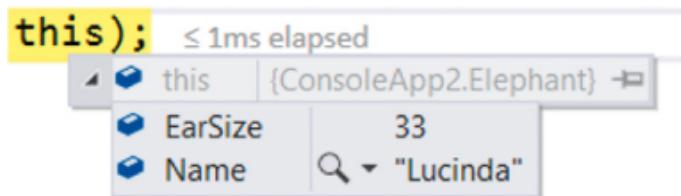
```
You pressed 5
Lloyd heard a message
Lucinda said this: Hi, Lloyd!
```

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

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



- a. Run your program and press 5.
- b. When it hits the breakpoint, use Debug >> Step Into (F11) to step into the Speak To method.
- c. 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.
- d. 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.

- e. The SpeakTo method has one statement—it calls whoToTalkTo.HearMessage. Step into it.
- f. 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.
- g. Hover over **whosaidit** and expand it. It’s a reference to the Lucinda object.

Finish stepping through the code. Take a few minutes and really understand what’s going on.

Arrays hold multiple values

NOTE

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;
```

NOTE

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 zero.

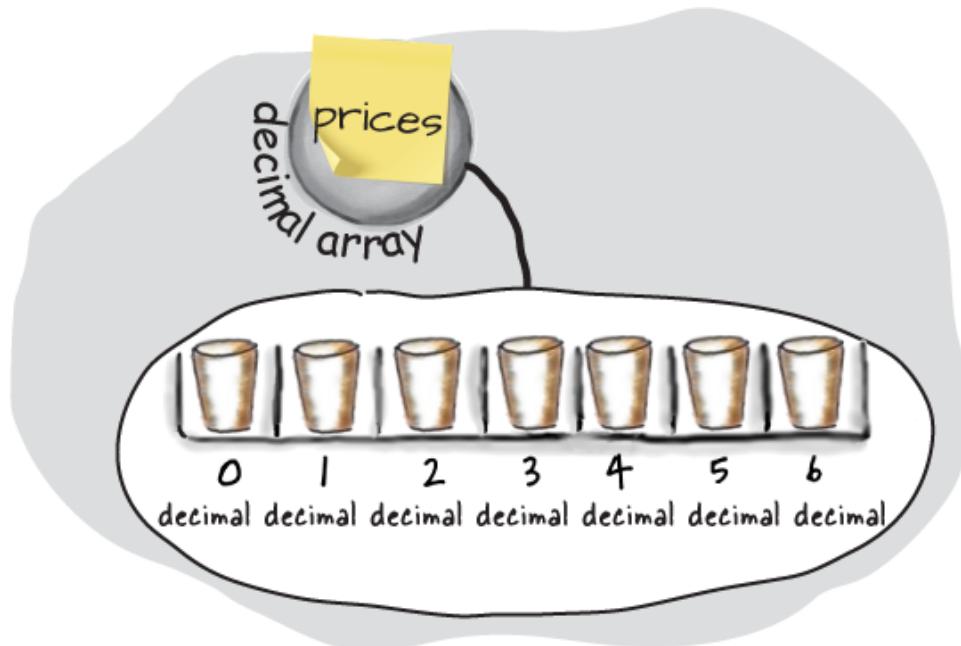
Use each element in an array like it's a normal variable

When you use an array, first you need to **declare a reference variable** that points to the array. Then you need to **create the array object** using the new statement, specifying how big you want the array to be. Then you can **set the elements** in the array. Here's an example of code that declares and fills up an array—and what's happening on the heap when you do it. The first element in the array has an **index** of zero.

NOTE

The `prices` variable is a reference, just like any other object reference. The object it points to is an array of decimal values, all in one chunk on the heap.

```
// declare a new 7-element decimal array
decimal[] prices = new decimal[7];
prices[0] = 12.37M;
prices[1] = 6_193.70M;
// we didn't set the element
// at index 2, it remains
// the default value of 0
prices[3] = 1193.60M;
prices[4] = 58_000_000_000M;
prices[5] = 72.19M;
prices[6] = 74.8M;
```



Arrays can contain reference variables

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

NOTE

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 zero.

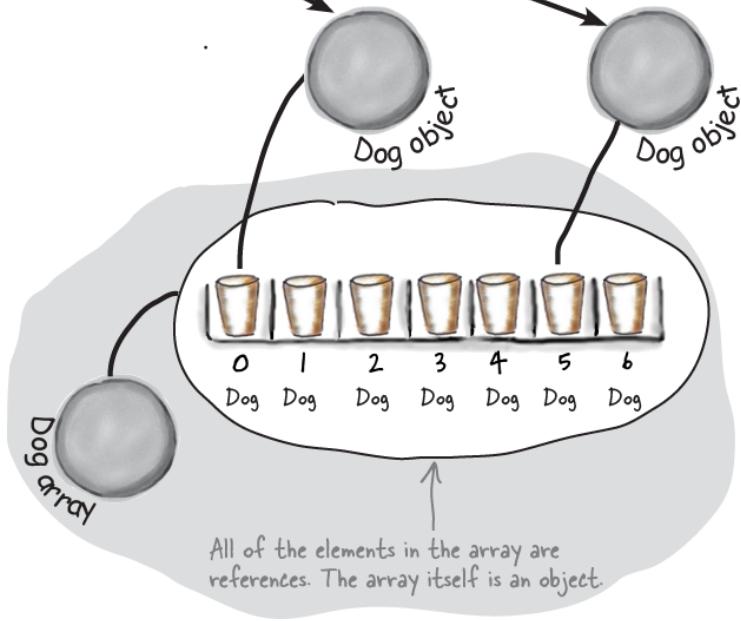
Here's code that creates an array of seven Dog variables. The line that initializes the array only creates reference variables. Since there are only two new Dog() lines, only two actual instances of the Dog class are created.

```

// 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();

```



NOTE

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.



SHARPEN YOUR PENCIL

Here's an array of Elephant objects and a loop that will go through it and find the one with the biggest ears. What's the value of the `biggestEars.Ears` **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());
    }
}

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 #4 biggestEars.EarSize = _____
                                         Iteration #5 biggestEars.EarSize = _____
                                         Iteration #6 biggestEars.EarSize = _____
```

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

Iteration #1 `biggestEars.EarSize` = _____

Iteration #2 `biggestEars.EarSize` = _____

Iteration #3 `biggestEars.EarSize` = _____

Iteration #4 `biggestEars.EarSize` = _____

Iteration #5 `biggestEars.EarSize` = _____

Iteration #6 `biggestEars.EarSize` = _____

null means a reference points to nothing

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

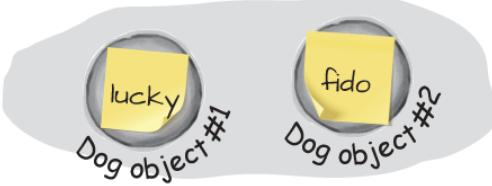
The default value for any reference variable is null. Since we haven't assigned a value to fido, it's set to null.

```
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 pointed 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

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 the `biggestEars.Ears` **after** each iteration of the for loop?

```
private static void Main(string[] args)
{
    Elephant[] elephants = new Elephant[7];
    elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
    elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
    elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
    elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
    elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
    elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
    elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };

    Elephant biggestEars = elephants[0];
    for (int i = 1; i < elephants.Length; i++)
    {
        Console.WriteLine("Iteration #" + i);
        if (elephants[i].EarSize > biggestEars.EarSize)
        {
            biggestEars = elephants[i];
        }
        Console.WriteLine(biggestEars.EarSize.ToString());
    }
}
```

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?

Iteration #1 `biggestEars.EarSize` = 40

Iteration #2 `biggestEars.EarSize` = 42

Iteration #3 `biggestEars.EarSize` = 42

Iteration #4 `biggestEars.EarSize` = 44

Iteration #5 `biggestEars.EarSize` = 44

Iteration #6 `biggestEars.EarSize` = 45

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`.

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.

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 a Mac, 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. And second, it manages the memory that your program uses. That means that it keeps track of all of your objects, figures out when the last reference to the 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 that you don't need. 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 whatever type of number you call it.

NOTE

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.



GAME DESIGN... AND BEYOND

Tabletop Games

There's a rich history to tabletop games – and, as it turns out, a long history of tabletop games influencing video games, at least as early as the very first commercial role-playing game.

- The first edition of Dungeons and Dragons (D&D) was released in 1974, and that same year games with names like “dungeon” and “dnd” started popping up on university mainframe computers.
- You've used the `Random` class to create numbers. The idea of games based on random numbers has a long history—for example, tabletop games that use dice, cards, spinners, and other sources of randomness.
- We saw in the last chapter how a paper prototype can be a valuable first step in designing a video game. Paper prototypes have a strong resemblance to tabletop games. In fact, you can often turn the paper prototype of a video game into a playable tabletop game, and use it to test some game mechanics.
- You can use tabletop games – especially card games and board games – as learning tools to understand the more general concept of game mechanics. Dealing, shuffling, dice rolling, rules for moving pieces around the board, use of a sand timer, and rules for cooperative play are all examples of mechanics.
- Take a minute and read the rules of the game Go Fish in the first section of the Wikipedia page for the game: https://en.wikipedia.org/wiki/Go_Fish – 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.



NOTE

If you've never played Go Fish, take a few minutes and read the rules. We'll use them later in the book!

Even if we're not writing code for video games, there's a lot we can learn from tabletop games.

A lot of our programs depend on **random numbers**. For example, you've already used the Random class to create random numbers for several of your apps. But 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 that you'll use in many of your programs, and your experience using random numbers 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.

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);
```

Now **simulate a the roll of a die**. We 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);
```

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);
```

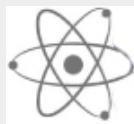
Use **casting** to convert the random double to other types. Try running this code a bunch of times—you'll see tiny precision differences in the float and decimal values.

```
Console.WriteLine((float)randomDouble * 100F);  
Console.WriteLine((decimal)randomDouble * 100M);
```

Use a maximum value of 2 to **simulate a coin toss**. That generates a random value of either 0 or 1. Use the special **Convert class**, which has

a static `ToBoolean` method that will convert it to a boolean value.

```
int zeroOrOne = random.Next(2);
bool coinFlip = Convert.ToBoolean(zeroOrOne);
Console.WriteLine(coinFlip);
```



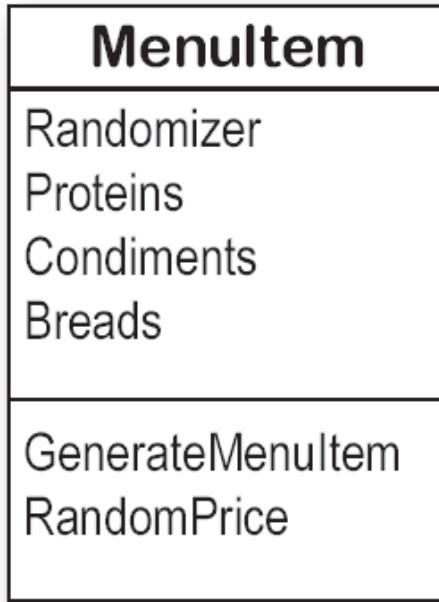
BRAIN POWER

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

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

Sloppy Joe has a pile of meat, a whole lotta bread, and more condiments than you can shake a stick at. But 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 project**, some arrays, and a couple of useful new techniques.

Do this!



1. Start with the fields for a new MenuItem class.

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" };  
}
```

2. Add the GenerateMenuItem method.

This method uses the same Random.Next method you've seen many times to random items from the arrays in the Proteins, Condiments, and Breads fields and concatenate them together into a string.

```
public string GenerateMenuItem()  
{  
    string randomProtein =  
Proteins[Randomizer.Next(Proteins.Length)];  
  
    string randomCondiment =  
Condiments[Randomizer.Next(Condiments.Length)];  
  
    string randomBread =  
Breads[Randomizer.Next(Breads.Length)];  
    return randomProtein + " with " +  
randomCondiment + " on " + randomBread;  
}
```

3. Add the RandomPrice method.

This method makes a random price by converting two random ints to decimals to make a random price between 2.01 and 5.97. 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 E.U. you'll see €, etc.

```
public string RandomPrice()
{
    decimal bucks = Randomizer.Next(2,
5);
    decimal cents = Randomizer.Next(1,
98);
    decimal price = bucks + (cents *
.01M);
    return price.ToString("c");
}
```

4. 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`. And 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 two columns width widths $5*$ and $1*$

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

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>

    <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>
```

Name each of the TextBlocks in the left column item1, item2, item3, etc., and in the TextBlocks in the right column price1, price2, price3, etc. Name the bottom TextBlock *guacamole*.

5. Here's 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. And the last is a special item with its own set of ingredients.

```

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

private void MakeTheMenu()
{
    MenuItem[] menuItems = new MenuItem[5];
    for (int i = 0; i < 5; i++)
    {
        This uses "new string[]" if (i < 3) {
            menuItems[i] = new MenuItem();
            to declare the type } else {
            of the array being menuItems[i] = new MenuItem()
            initialized. The MenuItem {
            fields didn't need to → Breads = new string[] { "plain bagel",
            include that because "onion bagel",
            they already have a type. "pumpernickel bagel",
            }; "everything bagel" }
        }

        item1.Text = menuItems[0].GenerateMenuItem();
        price1.Text = menuItems[0].RandomPrice();
        item2.Text = menuItems[1].GenerateMenuItem();
        price2.Text = menuItems[1].RandomPrice();
        item3.Text = menuItems[2].GenerateMenuItem();
        price3.Text = menuItems[2].RandomPrice();
        item4.Text = menuItems[3].GenerateMenuItem();
        price4.Text = menuItems[3].RandomPrice();
        item5.Text = menuItems[4].GenerateMenuItem();
        price5.Text = menuItems[4].RandomPrice();

        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" }
        };

        item6.Text = specialMenuItem.GenerateMenuItem();
        price6.Text = specialMenuItem.RandomPrice();

        guacamole.Text = "Add guacamole for " + specialMenuItem.RandomPrice();
    }
}

```

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 have 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 field already had the type definition in its declaration.

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.

You can call the special menu item object's `RandomPrice` method to get a 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".

NOTE

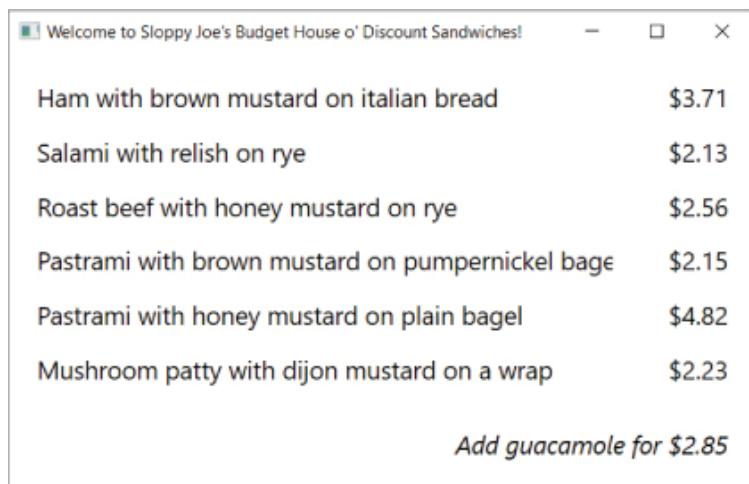
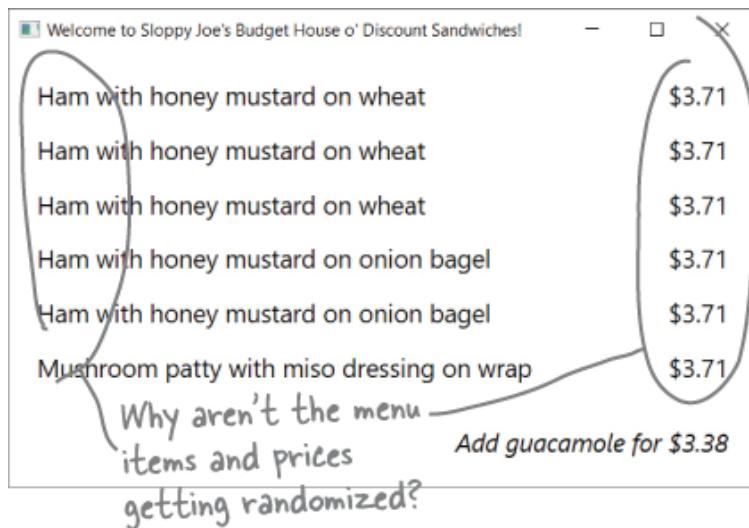
If your computer is fast enough, your program may not run into this problem. But if you run it on a much slower computer, you'll see it.

6. 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.

So 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? Easy—just 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.



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 arrays 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.

Part IV. 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 in random points around your game's 3D space and fly in circles around circles. This Unity Lab picks up where the last one left off.

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 ClickOneBall to the GameController class**. This method that 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 Score will go up.



THERE ARE NO DUMB QUESTIONS

Q: Why are we using 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. And 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. And the second half is actually not hard to read. 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.

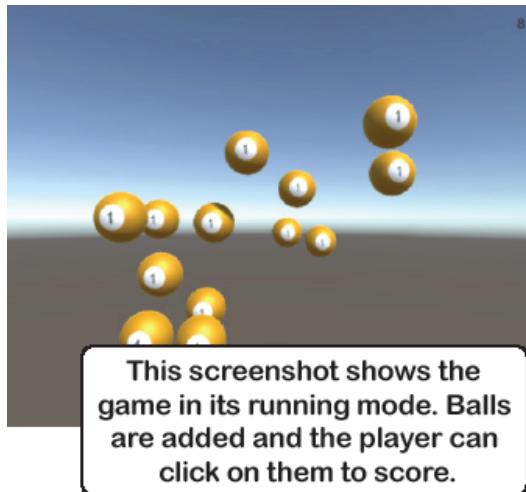
Add two different modes to your game

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

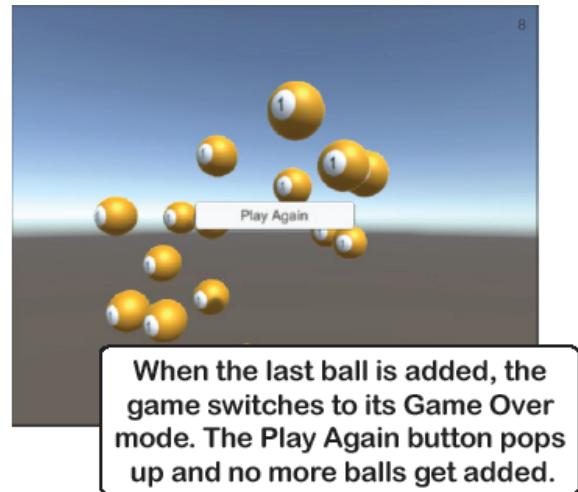
player is dying that can't be interrupted. These are all examples of **game modes**.

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



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



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

NOTE

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.

NOTE

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 scripts

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 GameController to know what mode the game is in. And when we need to keep track of what an object knows, we use fields. Since there are two modes—playing and game over—we can use a Boolean field to keep track of the mode, so **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 playing. 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** it 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.

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 GameOver is false.

NOTE

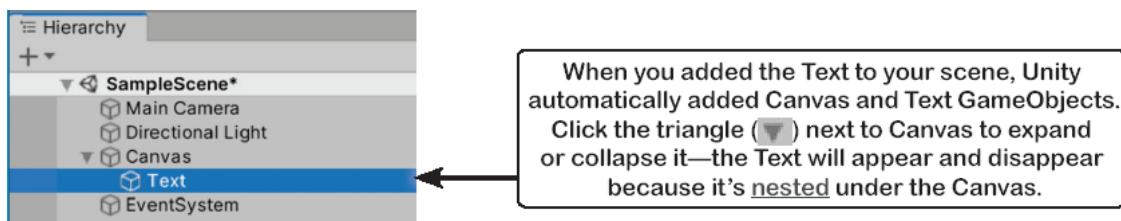
Your game needs to keep track of its game mode. Fields are a great way to do that.

Add a UI to your game

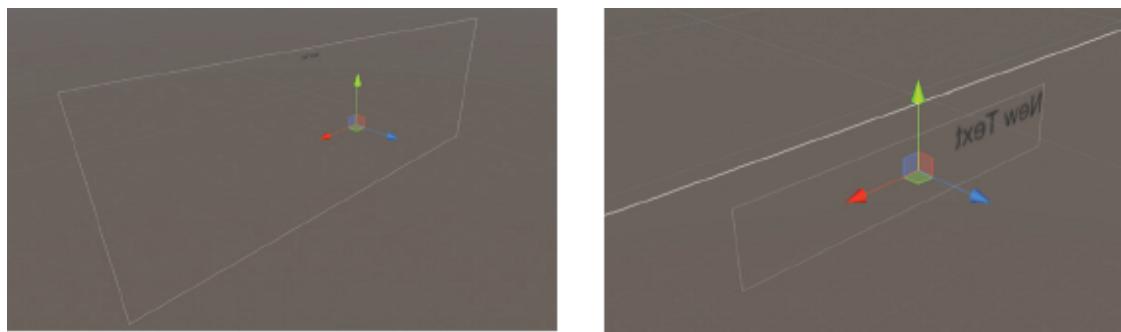
Almost any game you can think of—from Pac Man to Super Mario Brothers to Grand Theft Auto 5 to Minecraft—features a **user interface (or UI)**. Some games like Pac Man have a very simple UI that just shows the score, high score, lives left, and current level. But many games feature an intricate UI

incorporated it 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:



Double-click on the 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.



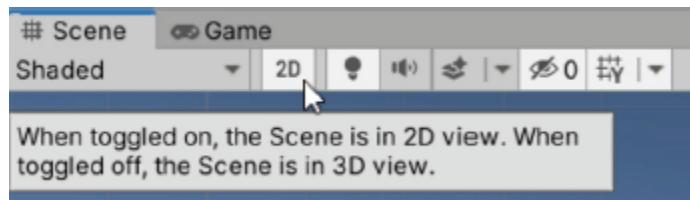
Then double-click on Text to focus on it—the editor will zoom in, but the default text ("New Text") will be backward because the Main Camera is pointing at the back of the Canvas.

NOTE

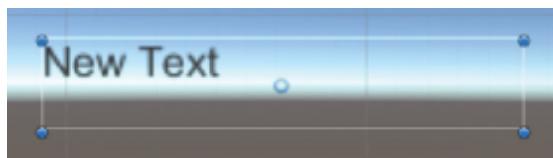
A Canvas is a 2-dimensional GameObject that lets you lay out your game's UI. Your game's Canvas will have two GameObjects nested under it: the Text GameObject that you just added will be in the upper left corner to display the score, and a Button GameObject that to let the player start a new game.

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. Hover over it to see its tooltip:



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.



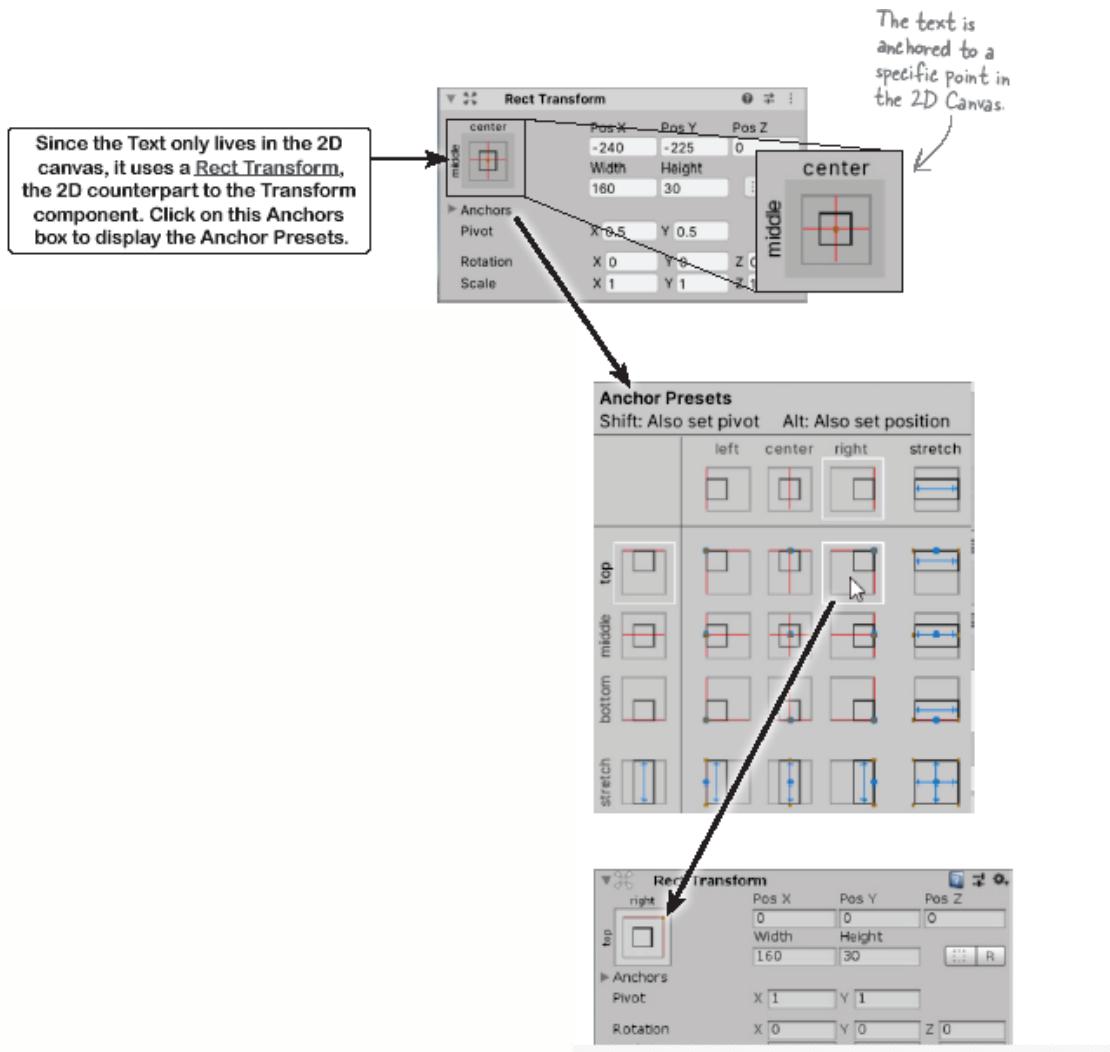
Use the mouse wheel to zoom in and out in 2D view.

You can **click the 2D button to switch between 2D and 3D**. Click it again to return to the 3D view.

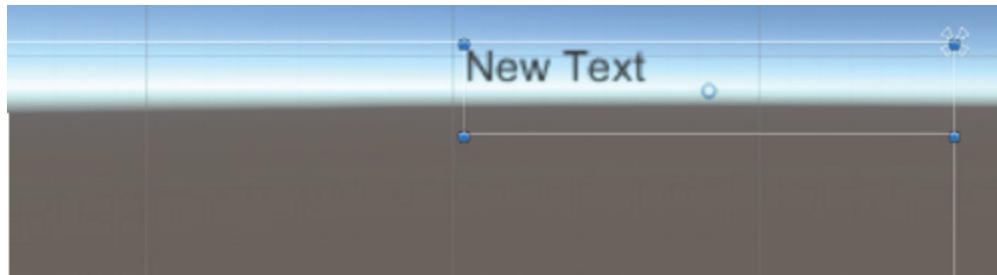
Set up the Text that will display the score in the UI

Your game's UI will feature three Text GameObjects and one Button. Each of those GameObjects will be **anchored** to a different part of the UI. For example, 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 Rect Transform.



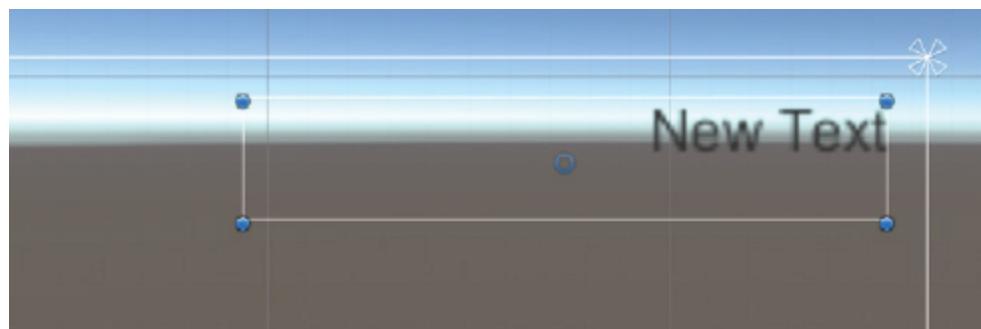
The Anchors Presets window lets you anchor your UI GameObjects to various parts of the Canvas. **Hold down alt and shift** (or option-shift on a Mac) and **choose the top right anchor preset**. Click the Anchor Presets button again to dismiss the window. The Text is now in the upper right corner of the Canvas—double-click on it again to zoom into it.



Let's add a little space above and to the right of the text. Go to the Rect Transform and **set the Pos X and Pos Y both to -10**. Then **set the Alignment 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 end of the Canvas.

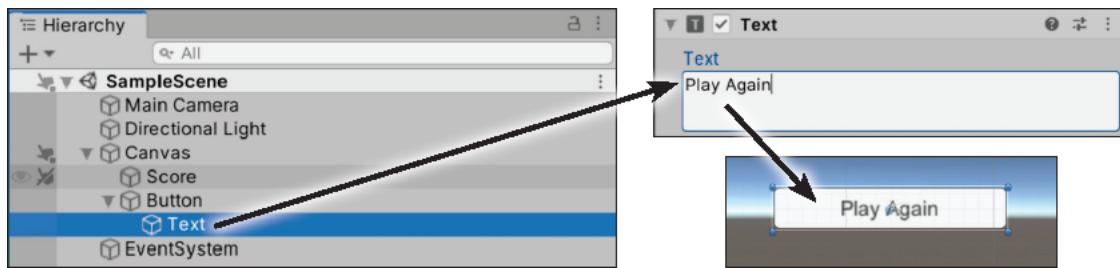


Add a button that calls a method to start the game

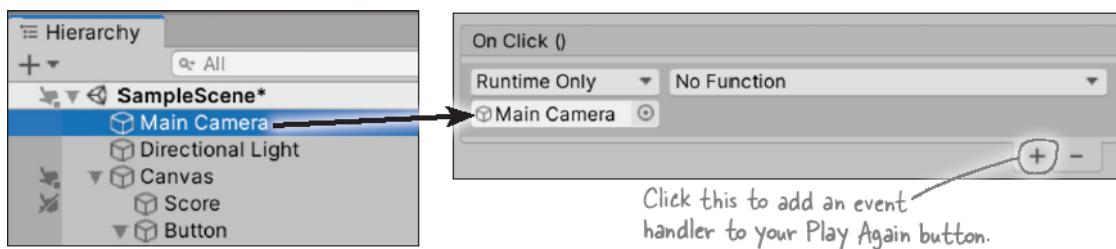
When the game is in its “game over” mode, it will display a button that says “Play Again” that calls a method to restart the game. **Add an empty StartGame method** to your GameController class (we’ll add its code later):

```
public void StartGame()
{
    // We'll add the code for this method later
}
```

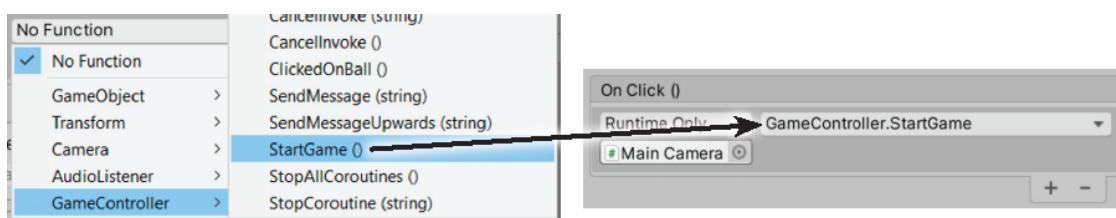
Double-click on Canvas to focus on it. Then **choose UI >> Button** from the GameObject menu to add a button. Since you focused on the canvas, the Unity editor will add it to its center. Did you notice that Button has a triangle next to it in the Hierarchy? Expand it—there’s a Text nested under it. Click 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

Here’s what you’ll do to make your game’s UI work:

- The game starts in the “Game Over” mode
- Clicking a “Play Again” button starts the game
- Text in the upper right corner of the screen displays the current score

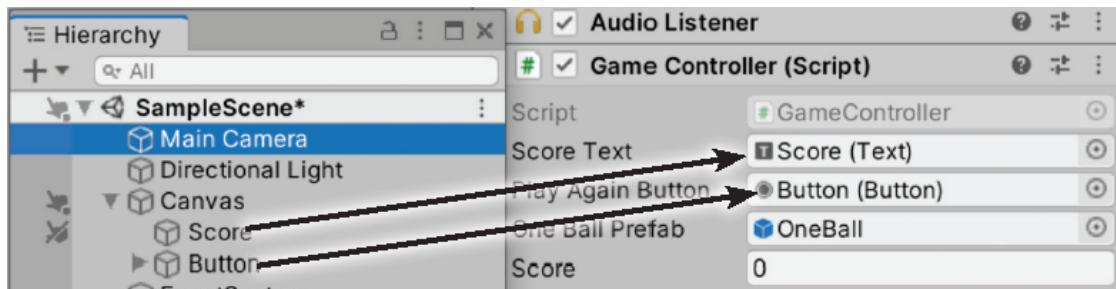
You’ll be using the Text and Button classes in your code. They’re in the UnityEngine.UI namespace, so **add this using statement** to the top of your GameController class:

```
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 Score 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.

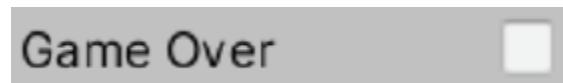


Go back to your GameController code and **set the GameController field's default value to true**:

```
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 your game will start in the “Game Over” mode. The player can click the Play Again button to start the game.



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.

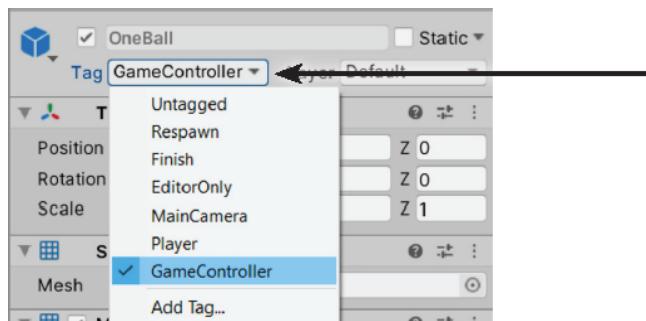
```
void Update()
{
    ScoreText.text = Score.ToString();
}
```

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 field 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 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.



EXERCISE

Here's a Unity [coding challenge](#) for you! Every GameObject 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);
```

Each of the parameters is 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?

.....
.....

NOTE

Before you start working on the game, figure out what these lines of code do.

- 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?

.....
.....

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();           ← When the ball is first instantiated,
                             its Start method calls ResetBall to
                             give it a random position and speed.
    }

    // 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.

First the update method moves the ball, then updates the speed, and finally checks if it went out of bounds. It's okay if you did these things in a different order.

```

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 okay 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?

By speeding up or slowing down the ball's speed along all three axes, we're giving each ball a wobbly random path.

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 either speed up or slow down a small amount for each frame.

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.

NOTE

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 how 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 faster, slower, or move differently.

- Can you figure out how to make a “shooting star” ball that flies off really quickly in a 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 GameContorller.ClickedOnBall method to take a score parameter instead of incrementing the Score field 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 the OneBall prefab! Otherwise, it will remember the old values.

NOTE

The more practice you get writing C# code, the easier it will get. Getting creative with your is a great opportunity to get some practice!



BULLET POINTS

- Unity games display a **user interface (UI)** with controls and graphics on a flat, 2-dimensional plane in front of the game's 3D scene.
- Unity provides a set of **2-dimensional UI GameObjects** specifically made for building user interfaces.
- A **Canvas** is a 2-dimensional 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 reset it from the gear 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. (You'll learn more about collections in Chapter 7.)

Chapter 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 safe, flexible, easy to use, and difficult to misuse. You'll **make your object's data private**, and add **properties** to protect how that data is accessed.

Let's help Ryan roll for damage

Ryan 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 3d6 and uses a formula that calculates the damage. Ryan 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.

Here's the
description ↗
of the sword
damage formula
in Ryan'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.

```

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 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 variable excepts 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 to the formula.

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 Ryan 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

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

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

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...

Let's double-check some of those calculations, just to make sure we understand the formula. Rolling 11 for a non-magic, non-flaming sword will cause $11 + 3 = 14$ HP of damage. Rolling 11 for a magic sword will cause (round down $11 \times 1.75 = 19$) + 3 = 22. Rolling 17 for a magic flaming sword causes (round down $17 \times 1.75 = 29$) + 3 + 2 = 34.



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 adding the 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");
    }
}
```



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

for options. But we think it would be cleaner and more visually obvious to use a **checkbox**.

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

Design the XAML for a WPF version of the damage calculator

Create a new WPF app—give its main window `Title="Sword Damage" Height="175" Width="300"`. It contains a grid with three rows and two columns. The top row has two CheckBox controls labeled Flaming and Magic, the middle row has a Button control labeled “Roll for damage” that spans both columns, and the bottom row has a TextBlock control that spans both columns.

Design this!

Sword Damage

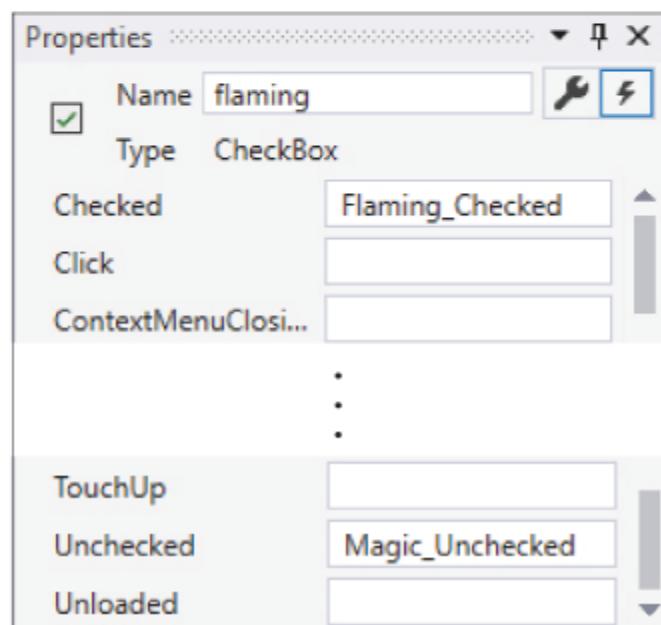
Flaming

Magic

Roll for damage

damage

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:

NOTE

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.

```
<Grid>
    <Grid.RowDefinitions>
        <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" /> ← The Checked and
    <Button Grid.Row="1" Grid.ColumnSpan="2" Margin="20,10"
            Content="Roll for damage" Click="Button_Click"/> Unchecked event
    <TextBlock x:Name="damage" Grid.Row="2" Grid.ColumnSpan="2" Text="damage"
              VerticalAlignment="Center" HorizontalAlignment="Center"/> handlers get called
    </Grid> when the user checks or unchecks the boxes.

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

The code-behind for the WPF damage calculator

Do this!

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



READY BAKE CODE

You've already seen that there 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 Ryan'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.**

```
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();
}

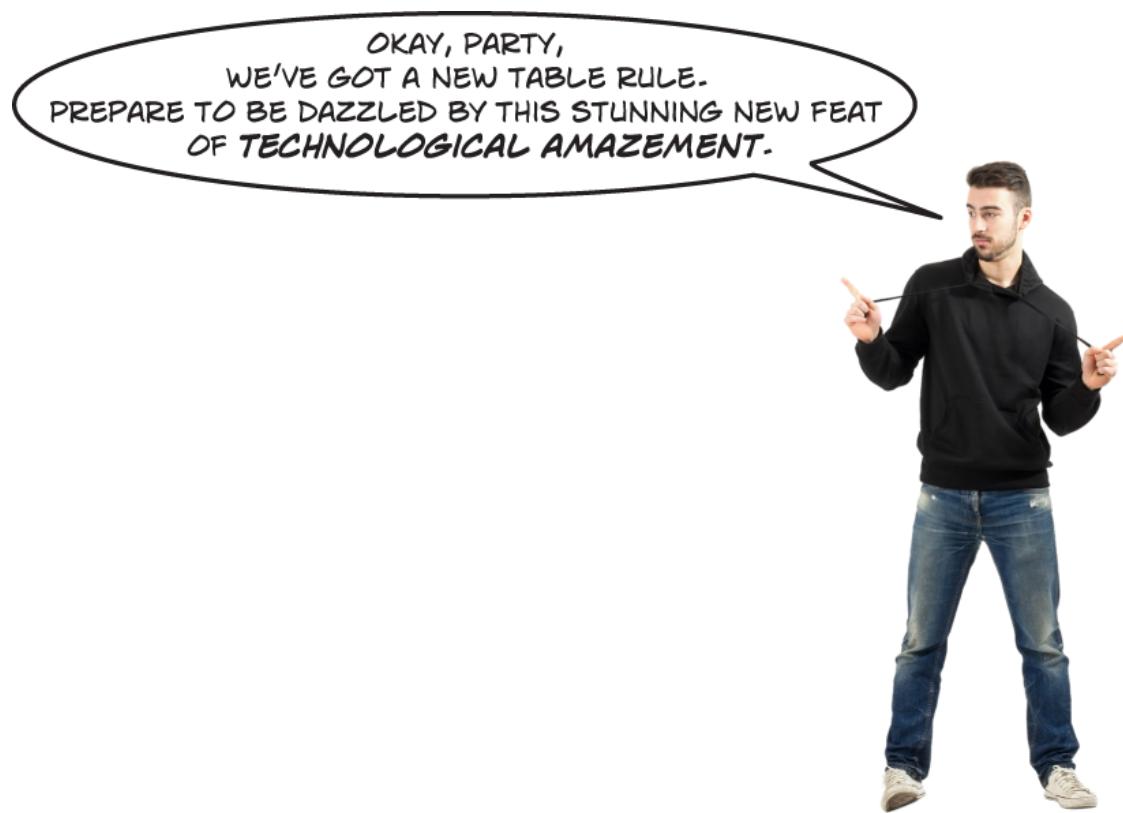
    private void Magic_Unchecked(object sender,
RoutedEventArgs e)
{
    swordDamage.SetMagic(false);
    DisplayDamage();
}
}
```

NOTE

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



Jayden: Ryan, what are you talking about?

Ryan: 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.

Ryan: 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: Okay. 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. Ryan, 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.

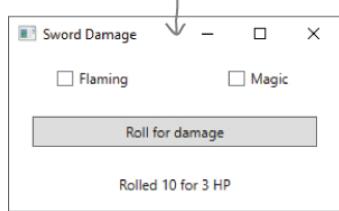
Jayden: You're right. We just have to click things in a **really specific order**. *First* we click roll, *then* we check the right boxes, and *just to be sure* we check the Flaming box twice.

Ryan: 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.

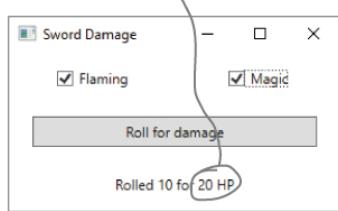
Okay, we can work with this.

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

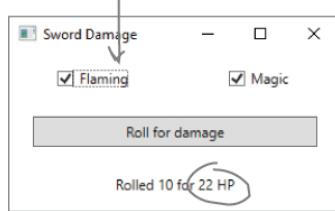
Brittany and Jayden are right. The program works, but only if you do things in a specific order. Here's what it looks like when it starts.



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



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



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.



That method is a **constructor**. When a class has a constructor, it's the very first thing that gets run when a new instance of that class is created . So when your app starts up 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 RollDice, so maybe we can fix this by hacking a solution into the RollDice method. **Make these changes to the RollDice method:**

Re-do this!

```
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();
}
```

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. Okay, 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 3HP for flaming. You still need to check and uncheck the Flaming checkbox to get the right number. So the program's still broken.



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

When something goes wrong in our 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 really easy to just add more buggy code. It’s always faster to take the time to figure out what really caused the bug and fix it, 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, 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. But you should only use

`Console.WriteLine` for displaying output to your users. 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`, 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.

NOTE

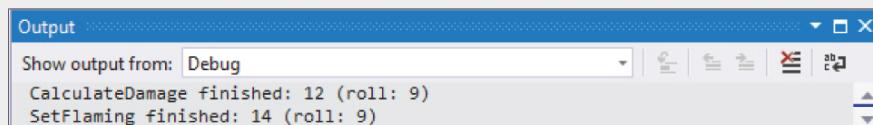
You can sleuth out this bug without setting any breakpoints. That's something developers do all the time... so you should learn how to do it, too!



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).

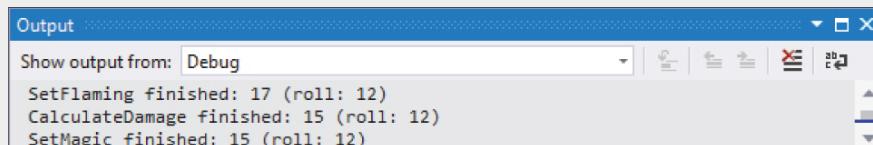
Once you see the main window, press the Clear All () button to clear the Output window. Then check the Flaming box. When we took this screenshot our roll was 9, so this is what it printed:



14 is the correct answer—9 plus base damage of 3 plus 2 for a flaming sword. Okay, 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:



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.

NOTE

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

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.

NOTE

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.



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. Luckily, C# gives you a powerful tool to make sure your program always works correctly— even when people do things you never thought of. It’s called **encapsulation** and it’s a really helpful technique 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: 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 another 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.

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: 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**: standard input (stdin), standard output (stdout), and standard error (stderr) provided by your operating system. 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.

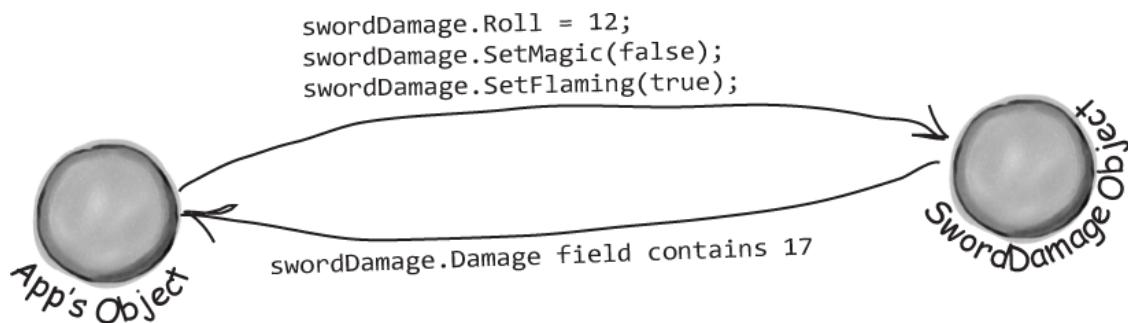
It's easy to accidentally misuse your objects

Ryan'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.

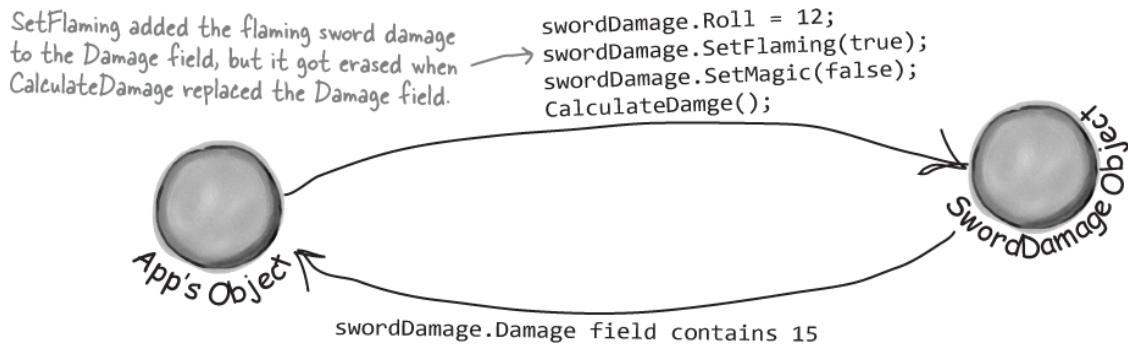
How the SwordDamage class expected to be used

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.



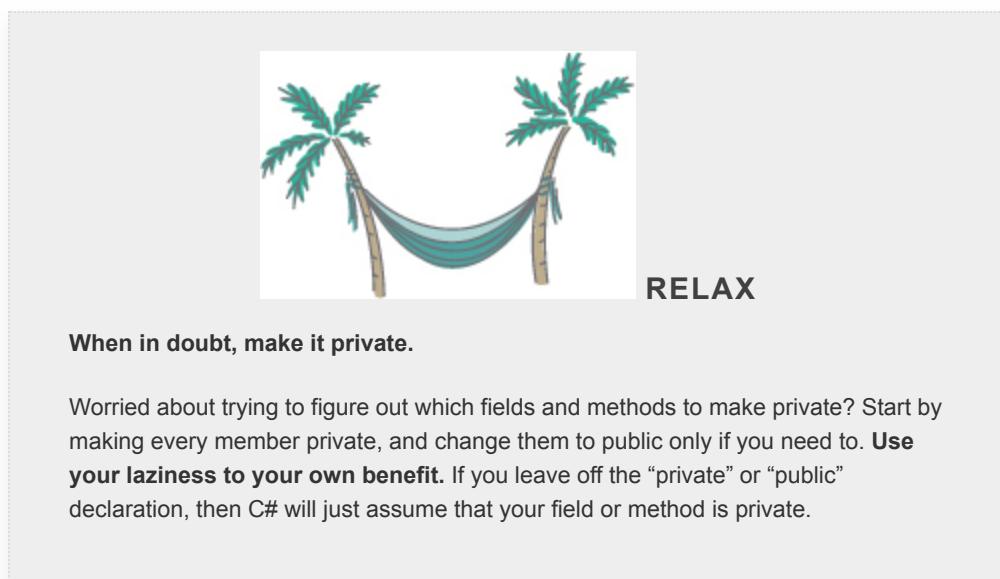
How the SwordDamage class was actually used

But that's not what the app did. Instead, set the Roll field, then it called SetFlaming, which added the extra damage for the flaming sword to the Damage field. But 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 an easy way to avoid this kind of problem: make sure that there's only one way to use your class. Luckily, C# makes it easy to 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. But 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()
    {
        ...
    }
}

```

If you want to make a field private, all you need to do is use the `private` keyword when you declare it. That 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 field names so they start with lowercase letters?

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*. And 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*.

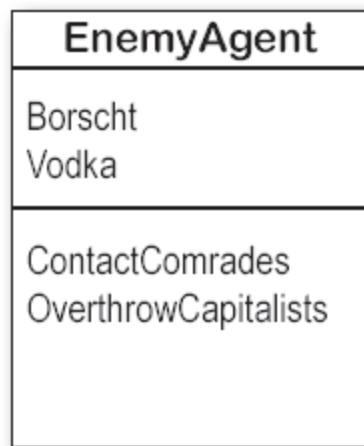
en-cap-su-la-ted, adj.

enclosed by a protective coating or membrane. *The divers were fully encapsulated by their submersible, and could only enter and exit through the airlock.*

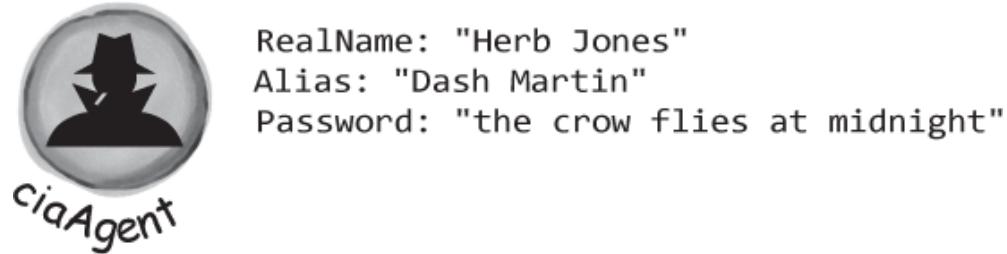
Use encapsulation to control access to your class's methods and fields

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.

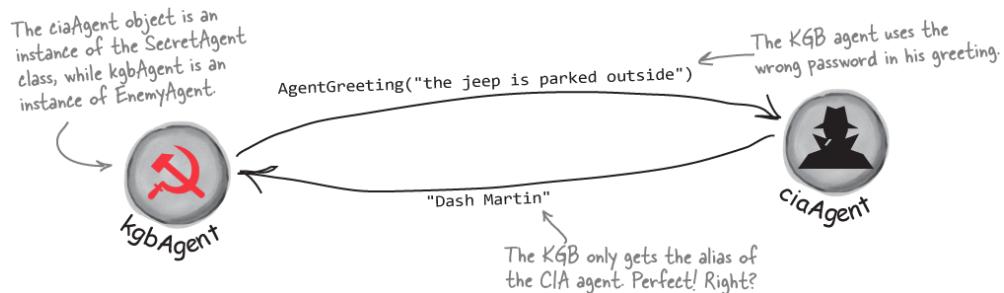
That's why the `public` and `private` keywords are called **access modifiers**: they modify access to class members. Encapsulation lets you control what you share and what you keep private inside your class. Let's see how this works:



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

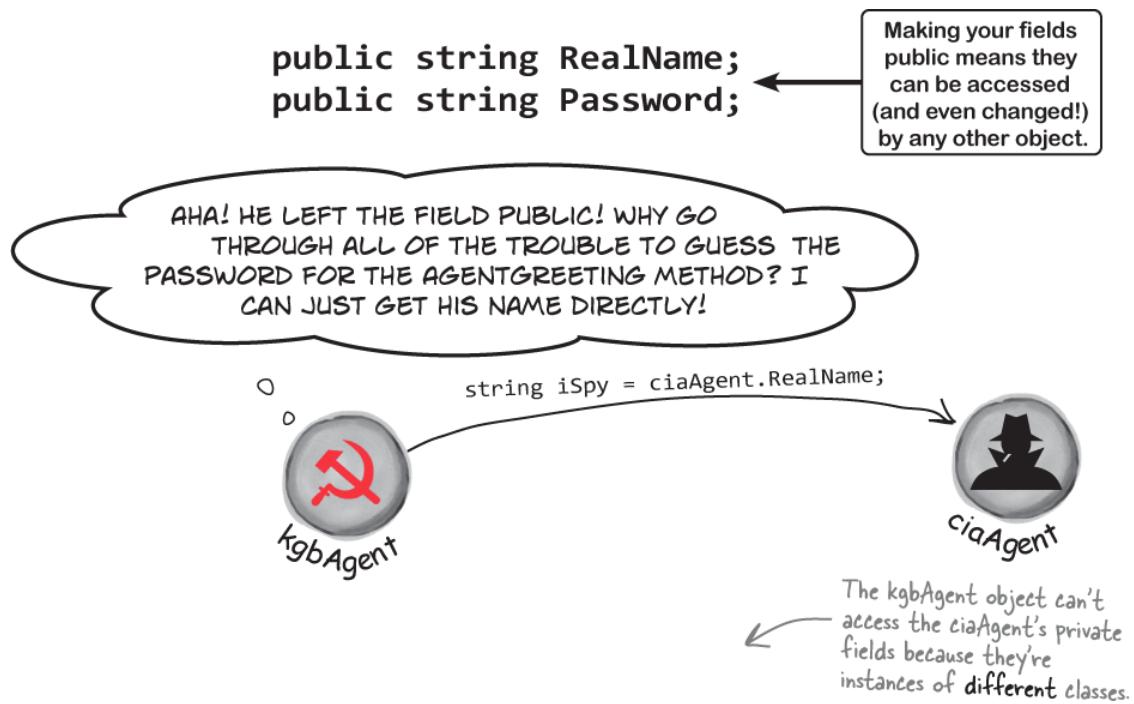


2. Agent Jones has a plan to help him evade the enemy KGB agent object. He added an `AgentGreeting` method that takes a password as its parameter. If he doesn't get the right password, he'll only reveal his alias, Dash Martin.
3. Seems like a foolproof way to protect the agent's identity, right? As long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



But is the `RealName` field REALLY protected?

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

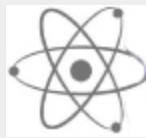


What can Agent Jones do? He can use **private** fields to keep his identity secret from enemy spy objects. Once he declares the `realName` field as private, the only way to get to it is **by calling methods that have access to the private parts of the class**. So the KGB agent is foiled!

NOTE

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;
```



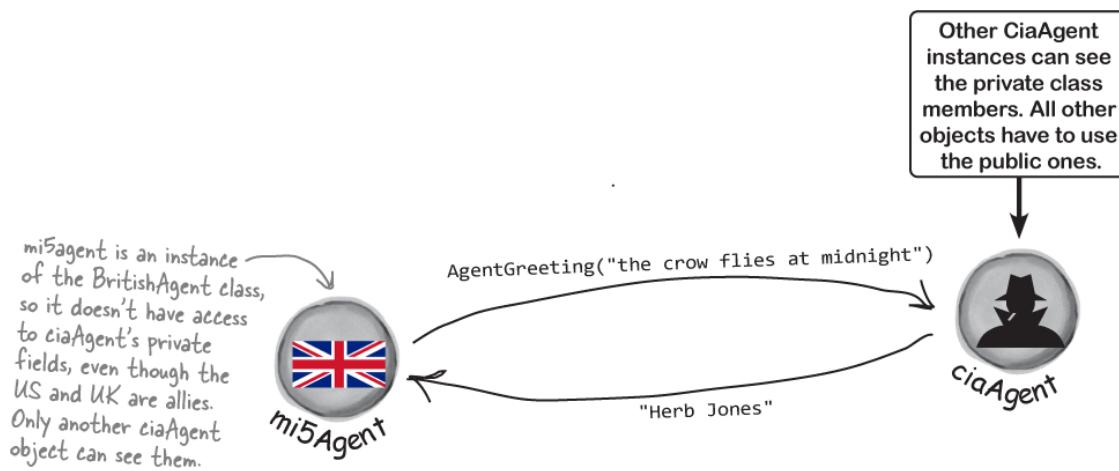
BRAIN POWER

Making the method 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 you think of a way to 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. But while KGB and MI5 agents need to use the AgentGreeting method, friendly spies that are also CiaAgent instances can see everything... because

any class can **see private fields in other instances of the same class.**



NOTE

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

THERE ARE NO DUMB QUESTIONS

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: 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 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: 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.



EXERCISE

Let's get a little practice using the `private` keyword by **creating a simple 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();
        else return;
    }
    Console.WriteLine("The pot is empty. Bye!");
}
```

NOTE

Don't forget—it's not cheating to peek at the solution!

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.**
5. 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?
6. 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.
 - **Replaces** the current number with the one it chose at the beginning of the method and **writes** "The current number is" followed by the number to the console.
7. 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 it's higher or lower it picks 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; } ←

    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--;
    }
}
```

If you try to add the static keyword to a constant you'll get a compiler error because all constants are static. Try adding one to any class—you can access from another class just like any other static field.

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.

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 single line doesn't need brackets.

BONUS: You can take 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 < 10; i++)
    Console.Write($"{seededRandom.Next(1,
HiLoGame.MAXIMUM + 1)}, ");
```

NOTE

Every instance of Random initialized with the same seed will generate the same sequence of pseudo-random numbers.



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. But there are some very good reasons that you'll want to think about what information in your class to expose to the rest of the program.

NOTE

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. But 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.

*And 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 the field for that object—even if the field is private.

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 only see how it behaves, but we have no way of knowing how it actually works.

When you call an object's methods, 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...

- **Easy to use**

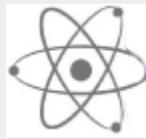
You already know that classes use fields to keep track of their state. And a lot of them use methods to keep those fields up to date—methods that no other class will ever call. It's pretty common to have a class that has fields, methods, and properties that will never be called by any other class. If you make those members private, then they won't show up in the IntelliSense window later when you need to use that class.

- **Easy to maintain**

That bug in Ryan 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 later on.



BRAIN POWER

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.



WE USED CONSTANTS FOR THE BASE AND FLAME DAMAGE. IT'S OKAY THAT THEY'RE PUBLIC BECAUSE THEY CAN'T BE MODIFIED.

BUT SINCE THEY'RE NOT BEING USED BY ANOTHER CLASS, MAYBE WE CAN ALSO MAKE THEM PRIVATE.

- **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.



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.

NOTE

And just like chess, there are an almost unlimited number of possible encapsulation strategies!

NOTE

If you encapsulate your classes well today, that makes them a lot easier to reuse tomorrow.



BULLET POINTS

- In WPF, **CheckBox** controls use the Content property to display the label to the right of the box and call Checked and Unchecked event handlers when the user clicks the box.
- 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**. 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 window.
- People won't always use your classes in exactly the way you expect. **Encapsulation** is a technique for making your class members safe 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**.



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 saw 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, MagicMultiple, 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. And even our attempt to fix it failed because we misused the methods by calling them in the wrong order.

NOTE

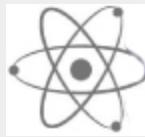
Making members of a class private can prevent bugs caused by other classes calling its public methods or updating its public fields in unexpected ways.

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.



BRAIN POWER

Think about those questions, then take another look at how the SwordDamage class works. What would you do to fix the SwordDamage class?

Encapsulation keeps your data safe

We've seen how you can use the private keyword to **protect fields from being written directly** to make class members private, 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 readonly 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 simple class

Let's build a MachineGun class for a video game where the player can pick up weapons and refill ammo. There are several guns and the ammo for one doesn't fit any of the others, so we want the class to keep track of the total number of bullets the player has. 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 bullets that calls the Reload method.

```

class MachineGun
{
    public const int MAGAZINE_SIZE = 16; ← We'll keep this constant public because
                                            it's going to be used by the Main method.

    private int bullets = 0;
    private int bulletsLoaded = 0;

    public int GetBulletsLoaded() { return bulletsLoaded; } ← When the game needs
                                                            to display the number of
                                                            bullets left and the number
                                                            of bullets loaded in the UI,
                                                            it can call the GetBullets and
                                                            GetBulletsLoaded methods.

    public bool IsEmpty() { return bulletsLoaded == 0; }

    public int GetBullets() { return bullets; } ←

    public void SetBullets(int numberOfBullets)
    {
        if (numberOfBullets > 0)
            bullets = numberOfBullets;
        Reload(); ← The game needs to be able to set the number of bullets.
    } ← The SetBullet method protects the bullets field by only
         allowing the game to set a positive number of bullets.
         Then it calls Reload to automatically reload the gun.

    public void Reload()
    {
        if (bullets > MAGAZINE_SIZE)
            bulletsLoaded = MAGAZINE_SIZE; ← 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
                                              bullets if there isn't a full magazine's worth.
                                              This keeps the bullets and bulletsLoaded
                                              fields from getting out of sync.
        else
            bulletsLoaded = bullets;
    }

    public bool Shoot()
    {
        if (bulletsLoaded == 0) return false;
        bulletsLoaded--;
        bullets--;
        return true; ← The Shoot method returns true
    } ← and decrements the bullet fields 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 MachineGun class

Do this!

Let's try out our new MachineGun class. **Create a new Console App** and add the MachineGun 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)
{
    MachineGun gun = new MachineGun();
    while (true)
    {
        Console.WriteLine($"{gun.GetBullets()} bullets, {gun.GetBulletsLoaded()} loaded");
        if (gun.IsEmpty()) Console.WriteLine("WARNING: You're out of ammo");
        Console.WriteLine("Space to shoot, r to reload, + to add ammo, q to quit");
        char key = Console.ReadKey(true).KeyChar;
        if (key == ' ') Console.WriteLine($"Shooting returned {gun.Shoot()}");
        else if (key == 'r') gun.Reload();
        else if (key == '+') gun.SetBullets(gun.GetBullets() + MachineGun.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 definitely encapsulated it well. The **bullets field is protected**: it doesn't let you set a negative number of bullets, and it keeps the bullets and bulletsLoaded fields in sync. 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.SetBullets(gun.GetBullets() +
        MachineGun.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 bullets 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 makes encapsulation easy. 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 GetBullets and SetBullets methods with a property

Here are the GetBullets and SetBullets methods from your MachineGun class:

```
public int GetBullets() { return bullets; }

public void SetBullets(int numberOfBullets)
{
    if (numberOfBullets > 0)
        bullets = numberOfBullets;
```

```
    Reload();  
}
```

Let's replace them with a property. **Delete both methods.**
Then **add this Bullets property:**

```
public int Bullets  
{  
    get { return bullets; }  
    set  
    {  
        if (value > 0)  
            bullets = value;  
        Reload();  
    }  
}
```

This is the declaration. It says that the name of the property is Bullets, and its type is int.

The get accessor (or getter) is identical to the GetBullets method that it replaced.

The set accessor (or setter) is almost identical to the SetBullets method. The only difference is it uses the value keyword where the SetMethod used its parameter. The value keyword will always contain the value being assigned by the set accessor.

The old SetBullets method took an int parameter called numberOfBullets with the new value for the backing field. The setter uses the value keyword everywhere the SetBullets method used numberOfBullets.

Modify your Main method to use the Bullets property

Now that you've replaced the GetBullets and SetBullets with a single property called Bullets, your code won't build anymore. We need to update the Main method to use the Bullets property instead of the old methods.

The GetBullets method was called in this Console.WriteLine statement:

Update this!

```
Console.WriteLine($"{gun. GetBullets()} bullets,  
{gun.GetBulletsLoaded()} loaded");
```

That's easy enough to fix—just **replace `GetBullets()` with `Bullets`** and statement will work just like before. But now let's have a look at the other place where the `GetBullets` and `SetBullets` were used:

```
else if (key == '+') gun. SetBullets(gun.GetBullets()  
+ MachineGun.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 `Bullets` property like a field—**replace that line** with this statement that uses the `+=` operator exactly like it would if `Bullets` were a field:

```
else if (key == '+') gun.Bullets += MachineGun.MAGAZINE_SIZE;
```

If Bullets were a field, this is how you would use the `+=` operator to update it. You use properties exactly the same way.

Here's the updated Main method:

```
static void Main(string[] args)  
{  
    MachineGun gun = new MachineGun();  
    while (true)  
    {  
        Console.WriteLine($"{gun.Bullets} bullets,  
{gun.GetBulletsLoaded()} 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.Bullets +=  
MachineGun.MAGAZINE_SIZE;  
        else if (key == 'q') return;  
    }  
}
```

Debug your MachineGun class to understand how the property works

Use the debugger to really get a good sense of how your new Bullet property works.

- Place a breakpoint inside the curly brackets of the get accessor (`return bullets;`).
- 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 the `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 **bullets** and the **value** keyword.

Auto-implemented properties simplify your code

Add this!

A very common way to use a property is to create a backing field and provide simple get and set accessors for it. Let's create a new BulletsLoaded property that **uses the existing bulletsLoaded field** as a backing field.

```
private int bulletsLoaded = 0;  
  
public int BulletsLoaded {  
    get { return bulletsLoaded; }  
    set { bulletsLoaded = value; }  
}
```

This is a very simple example of a property that uses a private backing field. Its getter returns the value in the field, and its setter updates the field.

Now you can **delete the GetBulletsLoaded method** and modify your Main method to use the property:

```
Console.WriteLine($"{gun.Bullets} bullets,  
{gun.BulletsLoaded} 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 BulletsLoaded property that you just created. But 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.

And 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 BulletsLoaded auto-property.

- 1. Remove the BulletsLoaded property and backing field.** Delete the BulletsLoaded property you added, because we're going to replace it with an auto-implemented property. Then delete the bulletsLoaded backing field (`private int bulletsLoaded = 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 BulletsLoaded** and press enter to finalize the snippet

```
public int BulletsLoaded { 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 bulletsLoaded field, your MachineGun class doesn't compile anymore. That's easy to fix—the bulletsLoaded field appears five times in the code (once in the IsEmpty method, and twice in the Reload and Shoot methods). Change them to BulletsLoaded—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 BulletsLoaded { get; set; }
```

This is definitely a great replacement for a property with get and set accessors that just update a backing field. And it's more readable and has less code than the bulletsLoaded field and GetBulletsLoaded method. So that's an improvement, right?

But there's just one problem: ***we've broken the encapsulation.*** The whole point of the private field and public method was to make the number of bullets loaded read-only. The Main method could easily set the BulletsLoaded 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 MachineGun class.

Make the BulletsLoaded setter private

Luckily, there's a very easy way to make our MachineGun 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 make 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.

So let's **make the set accessor private:**

```
:  
public int BulletsLoaded { get; private set; } ←
```

You can make your automatic property read-only by making its setter private.

Now the BulletsLoaded field is a **read-only property**. It can be read anywhere, but it can only be updated from inside the MachineGun class. The MachineGun 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 Ryan’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?

Right now the MachineGun class uses a `const` for the magazine size:

```
public const int MAGAZINE_SIZE = 16;
```

Replace this!

What if we want the game to set the magazine size when the gun is spawned? Let's **replace it with a property**.

- 1. Remove the MAGAZINE_SIZE constant and replace it with a read-only property.**

```
public int MagazineSize { get;  
    private set; }
```

- 2. Modify the Reload method to use the new property.**

```
if (bullets > MagazineSize)  
    BulletsLoaded =  
    MagazineSize;
```

- 3. Fix the line in the Main method that adds the ammo.**

```
else if (key == '+')  
    gun.Bullets += 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 we 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 bullets? 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 any of that?***

THERE ARE NO DUMB QUESTIONS

Q: Can you explain what a constructor is again?

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**. You can 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}");
    }
}
```

Then **add this `new` statement** to the Main method: `new ConstructorTest();`

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, 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.

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 `MachineGun` 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 bullets, an int called  
magazineSize, and a bool called loaded.  
  
public MachineGun(int bullets, int magazineSize, bool loaded)  
{  
    this.bullets = bullets; ←  
    MagazineSize = magazineSize;  
    if (!loaded) Reload();  
}
```

The constructor runs as soon new instance is created, so we put code into the body of the method to set the number of bullets, magazine size, and reload the gun if it starts out loaded. 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)'
The error message indicates that there is no argument provided for the required formal parameter 'bullets' in the call to the constructor.

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 `bullets` parameter has the same name as the field called `bullets`. And 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 and not the field. That's why we need to use the `this` keyword in the `MachineGun` constructor:

```
this.bullets = bullets;
```

When we just use `bullets` it refers to the parameter. But we want to set the field, but since it has the same name, we need to use `this.bullet` 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 (`MachineGun gun = new MachineGun()`). 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.

And, in fact, if you start typing the new statement again, 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, you need to specify arguments with types that match those parameters.

Modify this!

Let's modify your Main method to **pass parameters to the MachineGun constructor.**

- 1. Add the ReadInt method that you wrote for Ryan's ability score calculator in Chapter 3.**

You need to get the arguments for the constructor from somewhere. And you already have a perfectly good method that prompts the user for int values, so it makes sense reuse it here.

- 2. Add code to read values from the console input.**

Now that you've added the ReadInt method from Chapter 3, you can use it to get two int values. Add

these four lines of code to the top of your Main method.

```
int numberOfBullets = ReadInt(20, "Number of bullets");
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.

```
MachineGun gun = new
MachineGun(numberOfBullets,
magazineSize,
isLoaded);
```

4. Run your program.

Now run your program. It will prompt you for the number of bullets, magazine size, and whether or not the gun is loaded. Then it will create a new instance of `MachineGun`, passing arguments to its constructor that matches 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.

```
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!");
            }
        }
    }
}
```

```

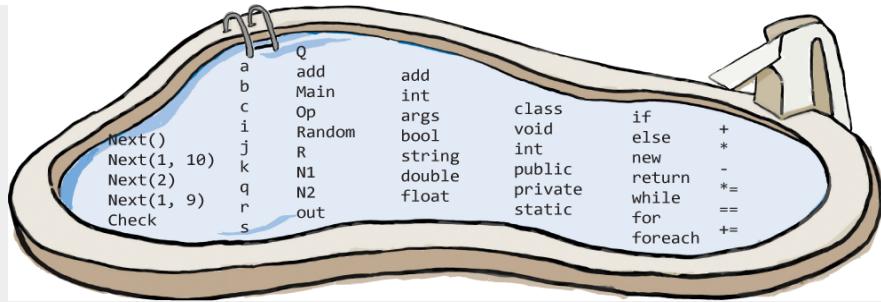
    _____ ;
}
if ( _____ . _____ (_____)) {
    Console.WriteLine("Right!");
    _____ = _____ Q( _____ .R.
    _____ == 1);
}
else Console.WriteLine("Wrong! Try again.");
}
}
}

```

Note: each snippet from the pool can be used more than once!

This program is a simple 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:

$8 + 5 = 13$	The game generates random addition or multiplication questions.
Right!	
$4 * 6 = 24$	If you get a question wrong, it keeps asking until you get it right.
Right!	
$4 * 9 = 37$	Wrong! Try again.
$4 * 9 = 36$	
Right!	Wrong! Try again.
$9 * 8 = 72$	
Right!	$6 + 5 = 12$
Wrong! Try again.	
$6 + 5 = 9$	The game ends when you enter an answer that isn't a number.
Wrong! Try again.	
$6 + 5 = 11$	Bye
Right!	
$8 * 4 = 32$	Thanks for playing!
Right!	
$8 + 6 = Bye$	



NOTE

We leveled up the difficulty for this puzzle! Remember, it's not cheating to peek at the solution if you get stuck.



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.

```
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

Note: each snippet from the pool can be used more than once!

This program is a simple 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

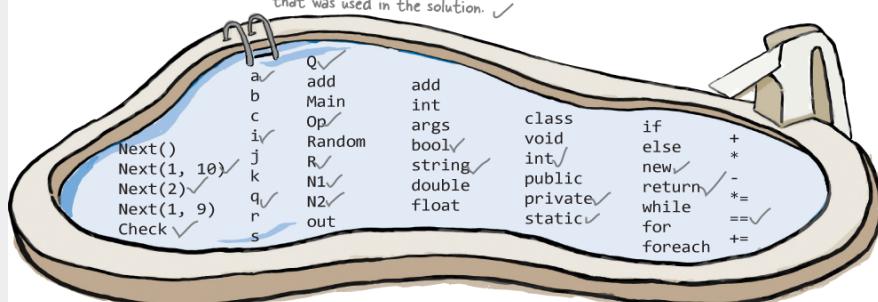
Thanks for playing!

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.

We put a check next to each snippet
that was used in the solution. ✓



A few useful facts about methods and properties

- **Every method in your class has a unique signature.**

The first line of a method that 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.

NOTE

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.



GAME DESIGN... AND BEYOND

Aesthetics

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 PacMan when Blinky (the red ghost) is just pixels behind you when you swallow the power pellet.

- It's obvious how the **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**, how
- 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 others' hands; the crescendo towards 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 to bring 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 multi-player game brings a sense of **fellowship** with others.
- A game that provides **fantasy** brings a player not just 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. But let these ideas sink in for now—we'll return to this in the next chapter.



NOTE

Some developers are really skeptical when they read about aesthetics because they assume that only the mechanics of the game matters. Here's a quick thought experiment to show how important aesthetics can be. Let's say you 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.



SHARPEN YOUR PENCIL

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 code 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 problems that will cause bugs. Here's what's wrong with them.

```
Lowercase-p price refers to the
parameter to the constructor, 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;
}

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";
    }
}

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.

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.
```

It's worth your time to take an extra few minutes **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: Why I don't add a return value to my constructor?

A: Your constructor doesn't have a return value 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 get without a set?

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 get without a set, 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.

NOTE

You can have a set without a get, or a get without a set (unless it's an automatic property).



EXERCISE

Use what you've learned about encapsulation to fix Ryan'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. And 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 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 `Magic` and a backing field named `magic`. 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 work 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 Ryan 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; } ← Since these constants aren't going
                                                to be used by any other class, it
                                                makes sense to keep them private.

    private int roll; ← The Damage property's private set
                      accessor makes it read-only, so it
                      can't be overwritten by another class.

    /// <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();
        }
    }
}
```

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.WriteLine("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 3d6 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.

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.

NOTE

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.

```
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)
```

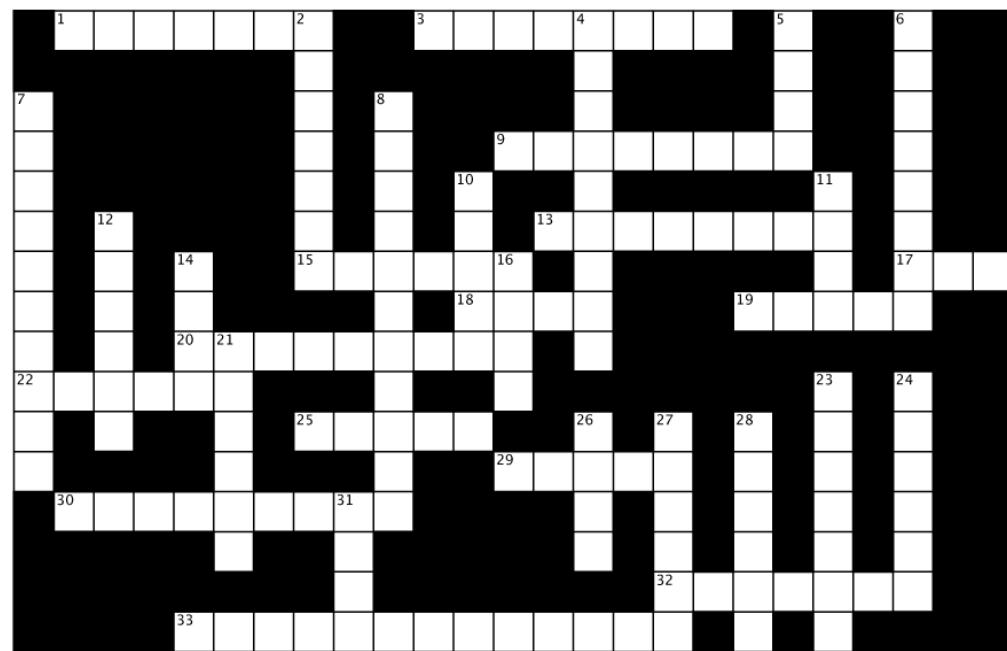
```

    {
        swordDamage.Magic = true;
        DisplayDamage();
    }
    private void Magic_Unchecked(object sender,
RoutedEventArgs e)
    {
        swordDamage.Magic = false;
        DisplayDamage();
    }
}

```

Objectcross

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 four chapters of the book.



Across

1. Draw one of these for your class before you start writing code
3. Every object has this method that converts it to a string
9. The four whole-number types that only hold positive numbers
13. Looks like a field but acts like a method
15. What not make a class if you want to be able to create instances of it
17. The statement you use to create an object
18. The second part of a variable declaration
19. A variable declared directly in a class that all its members can access
20. What you use to pass information into a method
22. What kind of sequence is \n or \r?
25. An object's fields keep track of its _____
29. What an object is an instance of
30. A variable that points to an object

32. The kind of collection that happens when the last reference to an object goes away

33. What you're doing when you use \$ and brackets include a value in a string

Down

2. They define the behavior of a class

4. namespace, for, while, using, and new are examples of _____ keywords

5. If a method's return type is _____, it doesn't return anything

6. += and -= are _____ assignment operators

7. How a method tells you what to pass to it

8. What you're doing when you use the + operator to stick two strings together

10. What floats in a float

11. How you start a variable declaration

12. You can assign any value to a variable of this type

14. Where objects live

16. What (int) is doing in this line of code: x = (int) y; 21. Tells a method to stop immediately, possibly sending a value back to the statement that called it

23. If you want to store a currency value, use this type

24. The numeric type that holds the biggest numbers

26. A good method _____ makes it clear what that method does

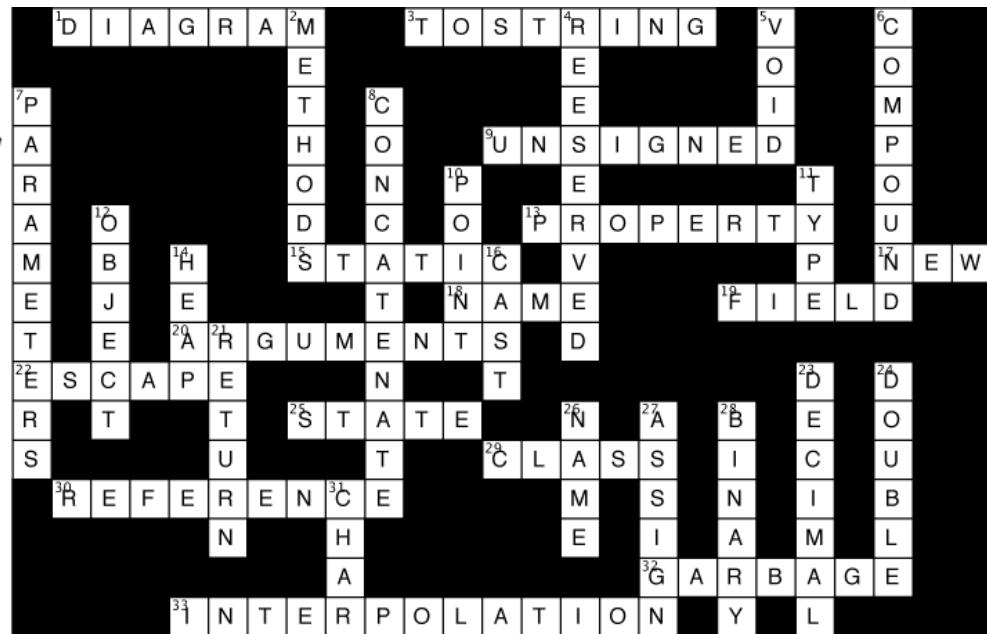


BULLET POINTS

- **Encapsulation** keeps your data safe by preventing classes from modifying it unexpectedly or otherwise misusing it.
- 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.
- Use 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 it 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.
- When parameter has the same name as a field, it **masks** the 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



Part V. 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. 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 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.

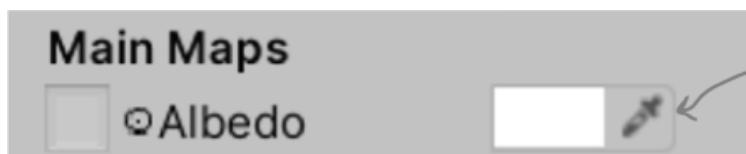
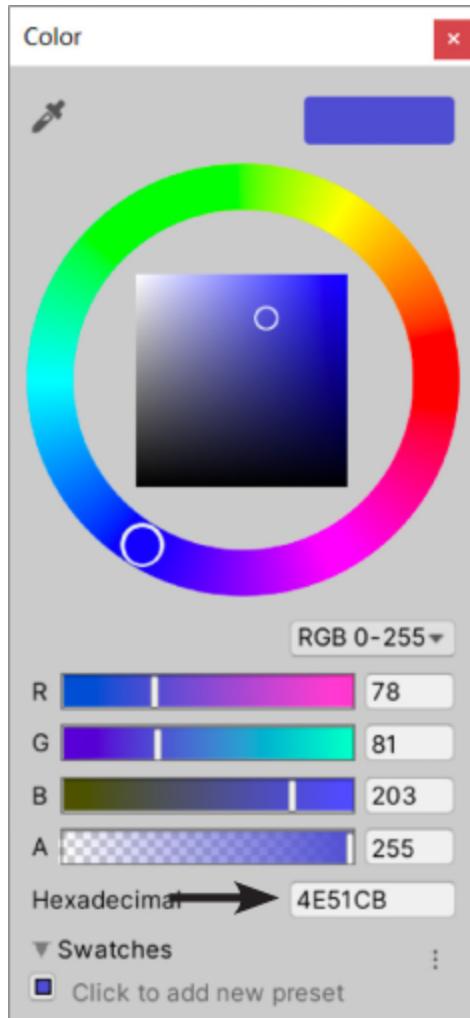
In this Unity Lab, you’ll use **raycasting** to write code that’s responsive to the geometry of the scene, **capture input** and use it to move a GameObject to a point that the player clicked. And just as importantly, you’ll **get practice writing C# code** with classes, fields, references, and other topics we’ve learned about.

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** (3D Object >> Plane) and name your new plane GameObject *Floor*.

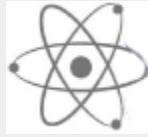
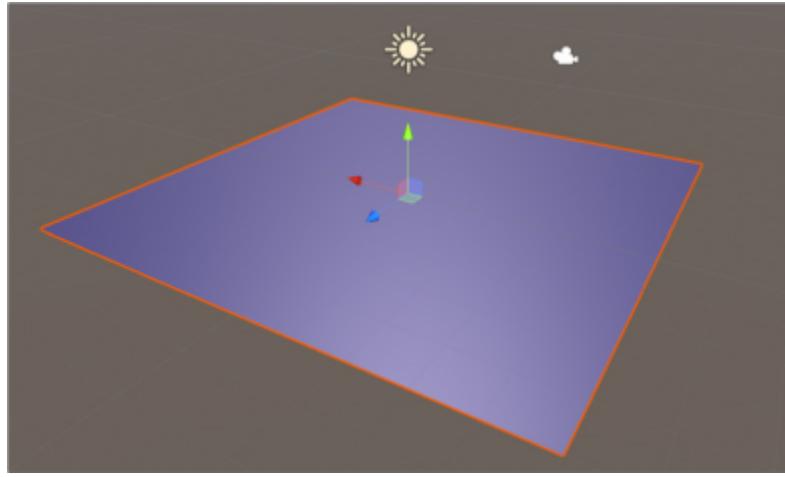
Right-click on the Assets folder in the Project window and **create a folder inside it called Materials**. Then right-click on the new Materials folder you created and choose **Create >> Material**. Call the new material *FloorMaterial*. Let's keep this material simple for now—we'll just make it a color. 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.



BRAIN POWER

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?

NOTE

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!)

NOTE

A Plane is a flat square object that's 10 units long by 10 units wide (in the XZ plane), and 0 units tall (in the Y plane). Unity creates it at point (0, 0, 0), the center of the plane. 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 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

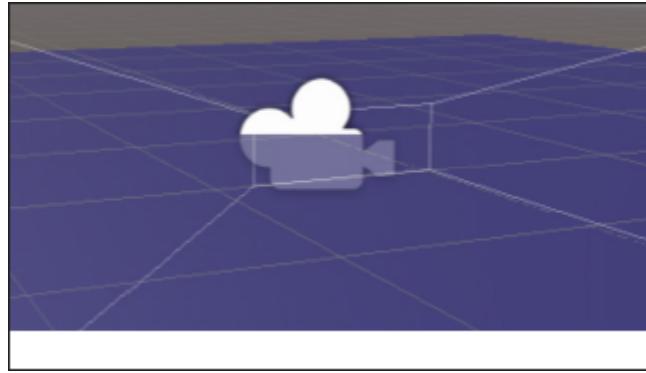
In the last two Unity Labs you learned that a GameObject is essentially a “container” for components, and that the Main Camera has just three components: a Transform, a Camera, and an Audio Listener. And that makes sense, because all a camera really needs to do is be at a location and record what it 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 rotating 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 reset both the camera's position and rotation to (0, 0, 0). You'll see the camera intersecting the plane in the Scene window.



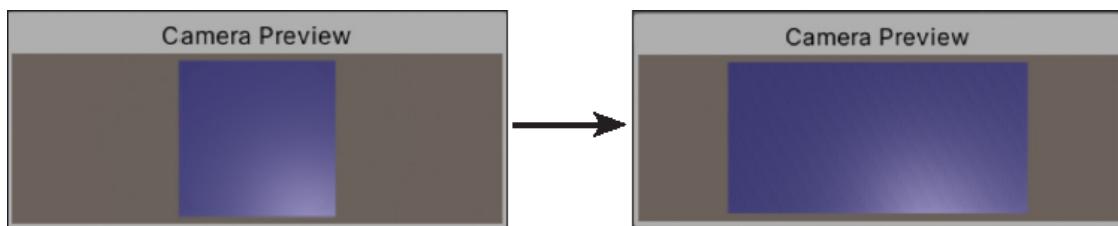
Now let's point the camera straight down. Start by clicking on the X label next to Rotation in the camera's Transform inspector and dragging up and down. You'll see the viewport in the camera preview move. Now use the Inspector window to **set the camera's X rotation to 90** to point it straight down.

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.



NOTE

You can switch between the Scene and Game windows to see what the camera sees.

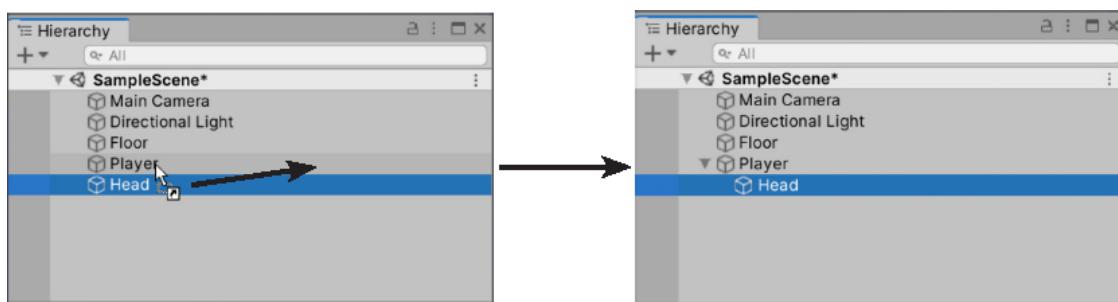
Create a GameObject for the player

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 >> Cylinder)—you'll see a cylinder appear in the middle of the scene. Change its name to *Body*, then **choose Reset from the context menu** in the Transform component to make sure it has all of its default values. Next, **create a Sphere GameObject** (3D >> Sphere). Change its name to *Head*, and reset its Transform

component as well. Each of them will have a separate line in the Hierarchy window.

But we don't want to 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*.

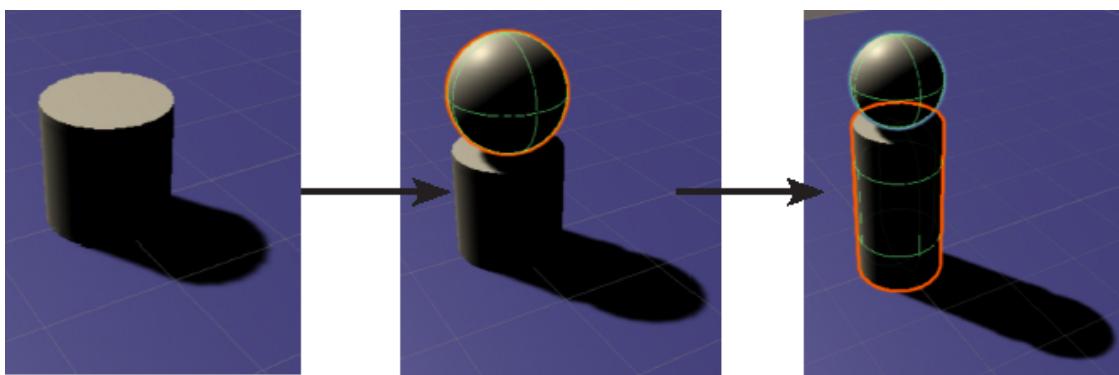


Select *Head* in the Hierarchy window. It was created at (0, 0, 0) like all of the other spheres you created. You can see the outline of the sphere, but you can't see the sphere itself because it's hidden by the plane and the cylinder. Use the Transform component in the Inspector window to **change the 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. But notice how it took the *Head* sphere along with it. Since *Head* is nested under *Player*, moving *Player* causes *Head* to move along

with it because moving a parent GameObject moves its children—in fact, *any* change that you make to its Transform component will automatically get applied to the children. If you scale it down, children will scale, too.

Switch to the Game window—your player is in the middle of the play area.



NOTE

When you modify the Transform 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... any 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

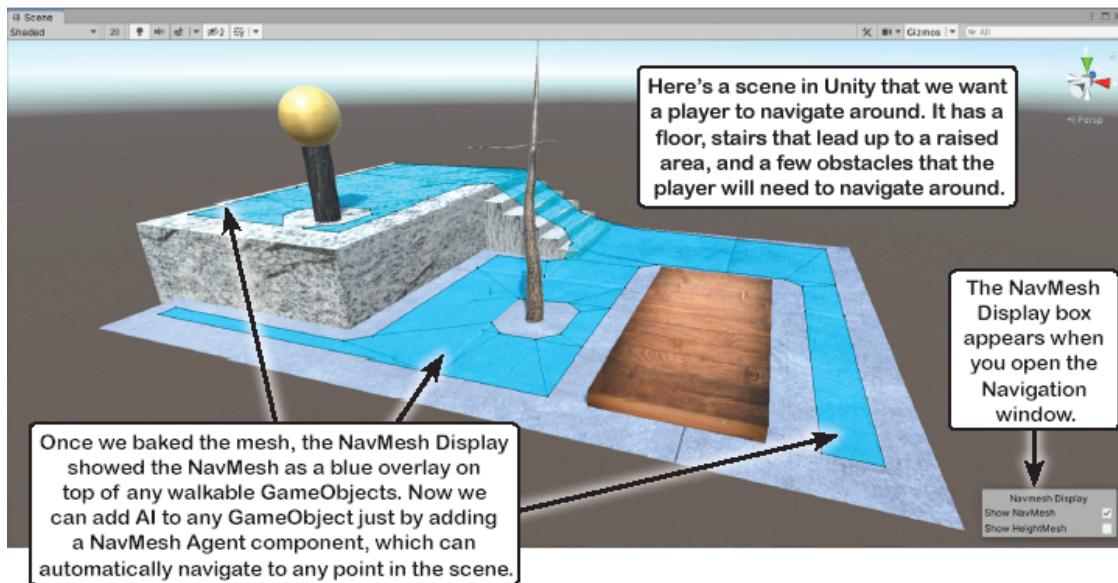
scenes. We'll take advantage of the navigation system to make our player move towards a target.

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 pre-compute (or bake)** the geometric details to make the agents work more efficiently.

NOTE

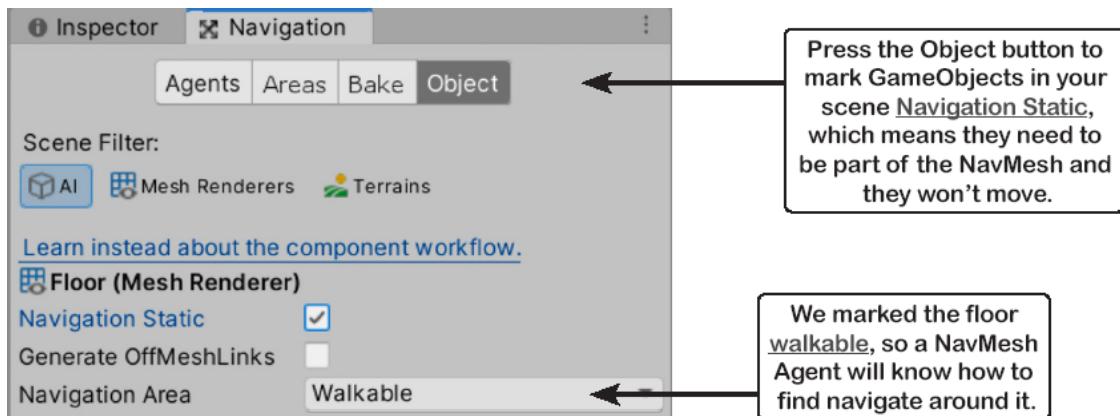
Unity provides a sophisticated AI navigation and pathfinding system that can move your GameObjects around a scene in real time by finding an efficient path that avoids obstacles.



Set up the NavMesh

Let's set up a simple NavMesh that just consists of the *Floor* plane. The way that we do this is to use 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.



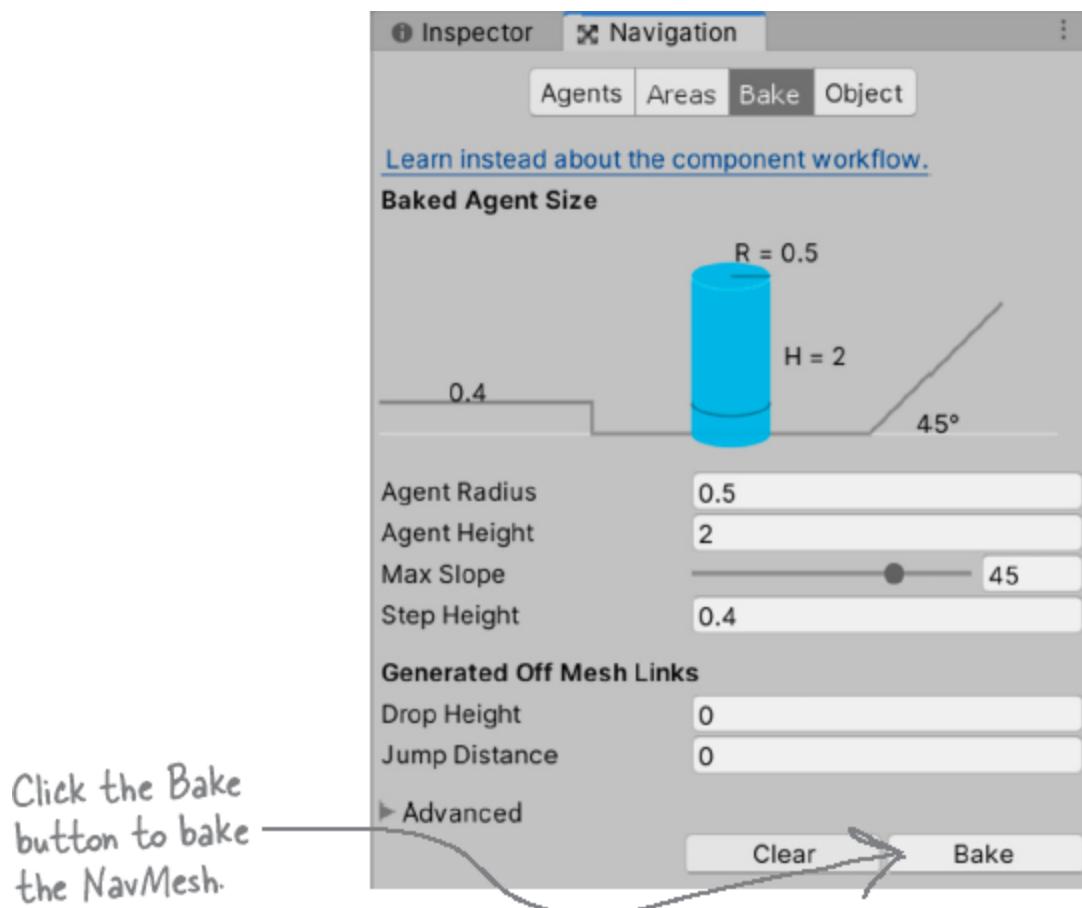
Since the only walkable area in this game will be the floor, we're done in the Object section. For a more complex scene with many walkable surfaces or non-walkable obstacles, each individual GameObject needs to be marked appropriately.

Click the Bake button at the top of the Navigation window. Now the Navigation window is showing you 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.

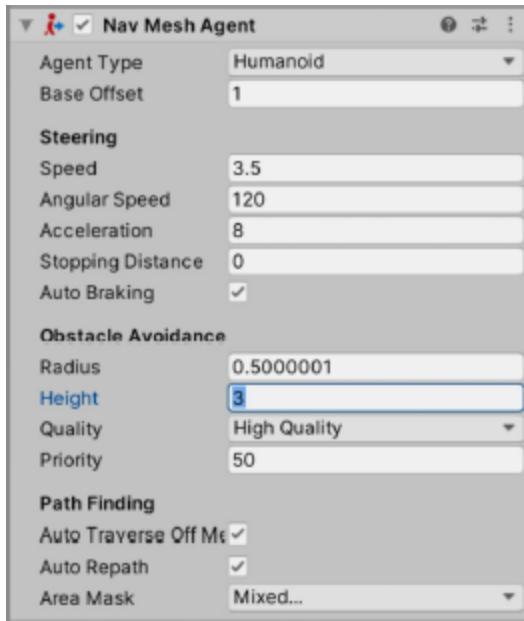


Make your player automatically navigate the play area

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. We 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 saves 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.GetMouseButtonDown(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 Unity loads the script. 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.GetMouseButtonDown 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, this catches all mouse clicks. That's all there is to it!

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. When you click on the plane, the NavMesh agent will move your player to the point that you clicked.



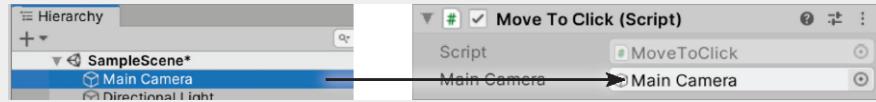
SHARPEN YOUR PENCIL

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 GameObject MainCamera;
```

Go back to the Hierarchy window, click on Player, and find the new Main Camera field in the Move To Click (Script) component, and **drag the Main Camera** out of the Hierarchy window and onto the Main Camera field in the inspector:



Now replace this line:

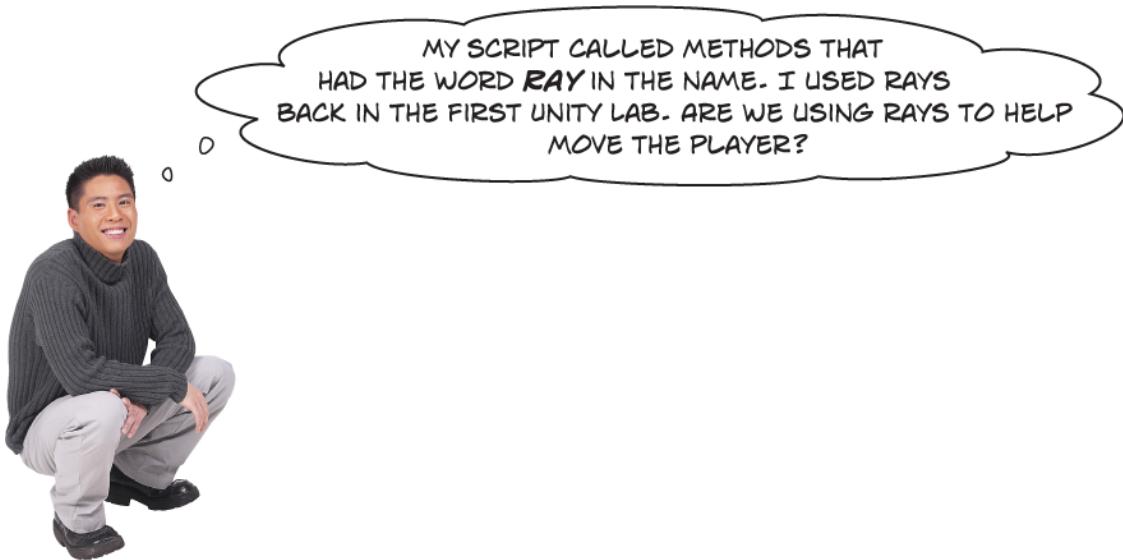
```
Camera cameraComponent = GameObject.Find("Main Camera").GetComponent<Camera>();
```

With this:

```
Camera cameraComponent =  
MainCamera.GetComponent<Camera>();
```

Run your game again. It still works! Why? Think about it, see if you can figure it out. Write down the answer:

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



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 and if the ray hit the floor**. If 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** towards that location.



RAYCASTING UP CLOSE

Your MoveToClick class 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. It then calls the Physics.Raycast method—its 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.

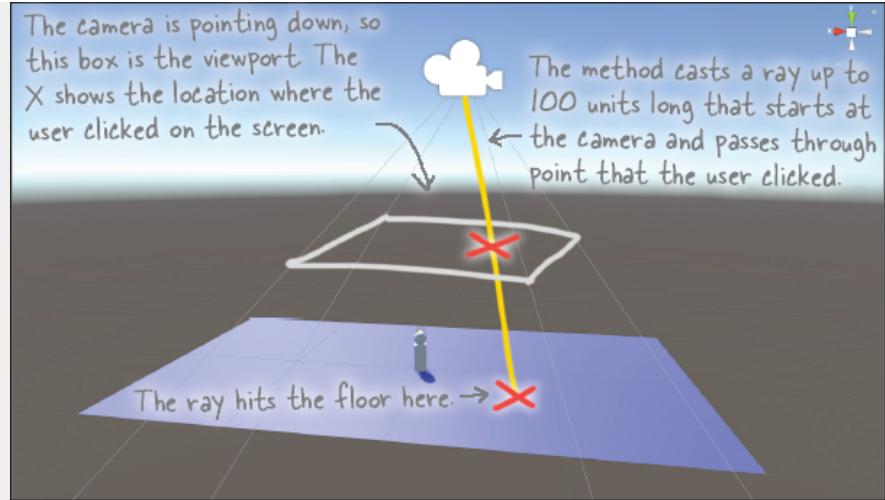
NOTE

We need to tell Physics.Raycast where to shoot the ray. So the first thing we need to do is to 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. So when you put them together, you get the ray to feed into Physics.Raycast:

```
Ray ray =  
cameraComponent.ScreenPointToRay(Input.mousePosition);
```



NOTE

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 towards the point where the ray hit.



SHARPEN YOUR PENCIL SOLUTION

We gave you a pencil-and-paper exercise to do. You modified the MoveToClick class to add a field for the main camera instead of using the Find and GetComponent methods and had you drag the Main Camera onto it, then we asked you some questions. Was your answer similar to ours?

Run your game again. It still works! Why? Think about it, 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 called 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 XZ 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 for 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 new path to a new position and start moving towards 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 “casts”) 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 the user clicked.

Chapter 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.

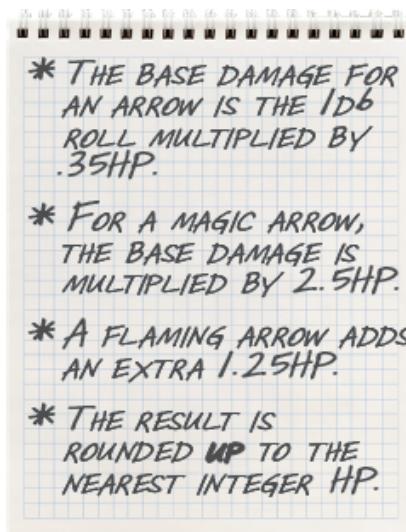
Ever run across an object that **almost** does exactly what you want **your** object to do? Found yourself wishing that if you could just **change a few things**, that object would be perfect? Well, that's just one reason that **inheritance** is one of the most powerful concepts and techniques in the C# language. Before you're through with this chapter, you'll learn how to **subclass** an object to get its behavior, but keep the **flexibility** to make

changes to that behavior. You'll **avoid duplicate code**, **model the real world** more closely, and end up with code that's **easier to maintain**.

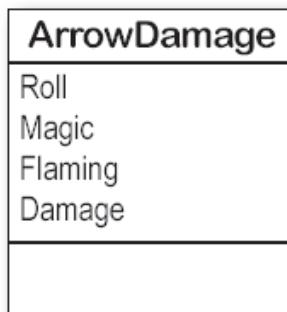
Calculate damage for MORE weapons

Do this!

The updated sword damage calculator was huge a hit on game night! Now Ryan 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 Ryan's game master notebook.



Most of the code in ArrowDamage will be **identical to the code** in the SwordDamage class. Here's what to do to get started building the new app:



- 1. 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.
- 2. 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.
- 3. Change the two 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.

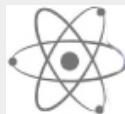
```
private const decimal BASE_MULTIPLIER = 0.35M;
private const decimal MAGIC_MULTIPLIER = 2.5M;
private const decimal FLAME_DAMAGE = 1.25M;
```

} Do you agree with us that these constants make the code easier to read? It's okay if you don't!

- 4. Modify the CalculateDamage method.** Now all you need to do to make your new ArrowDamage class work is to update 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.



BRAIN POWER

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?

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:

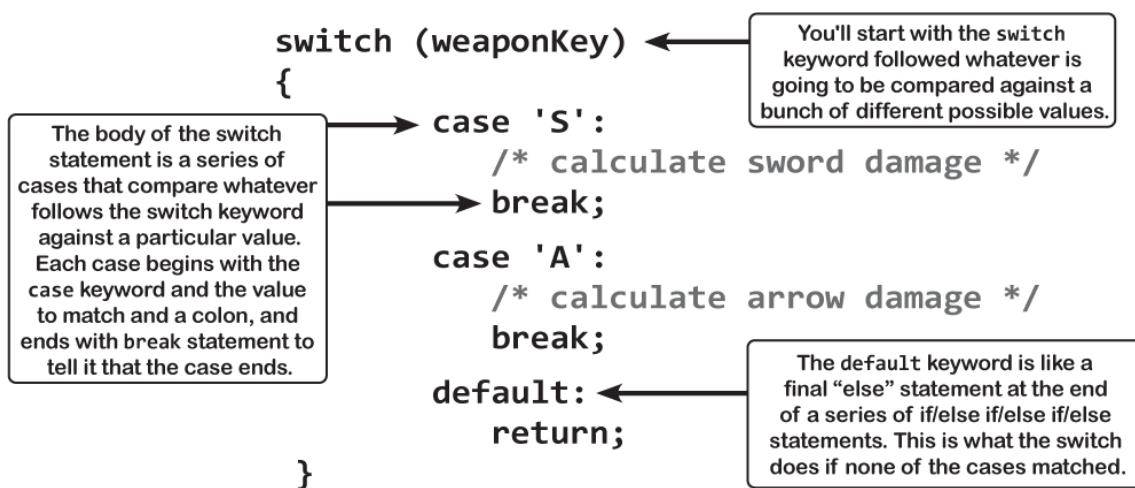
```
Console.WriteLine("nS for sword, A for arrow, anything else to quit: ");
weaponKey = Char.ToUpper(Console.ReadKey().KeyChar);
```

The `Char.ToUpper` method converts 's' and 'a' to 'S' and 'A'.

We **could** use if/else if/else statements:

```
if (weaponKey == 'S') { /* calculate sword damage */ }
else if (weaponKey == 'A') { /* calculate arrow damage */ }
else return;
```

That's how we've handled input so far. But 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 if/else statement 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 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 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;
        }
    }
}
```

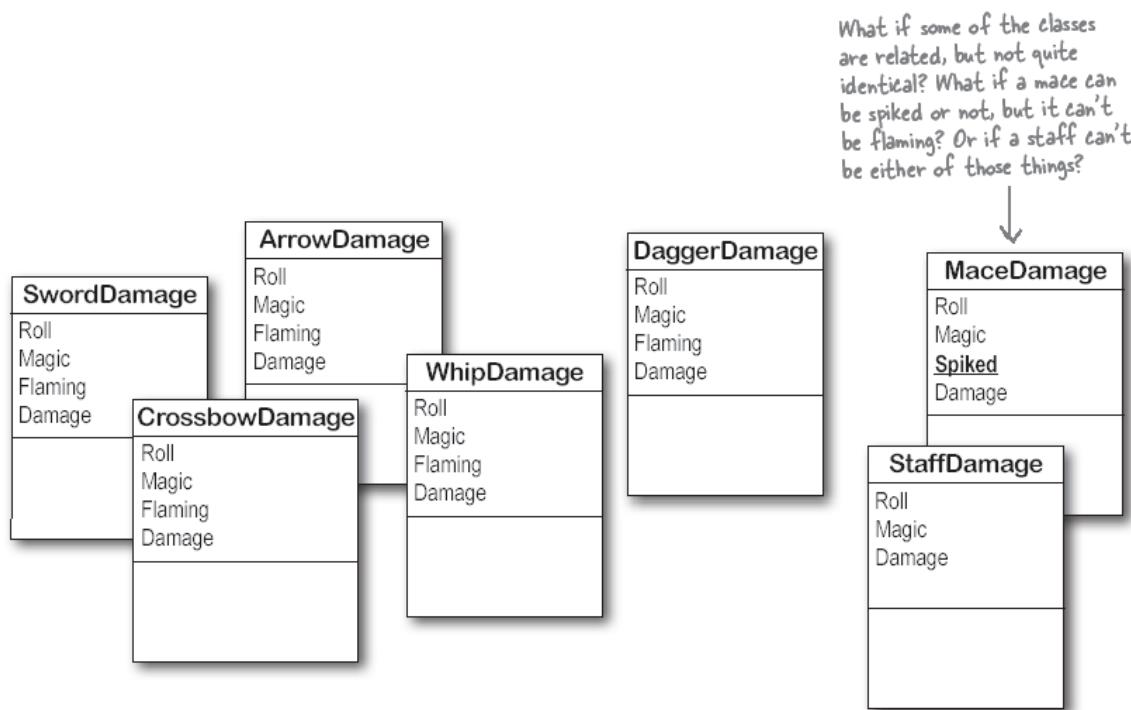
This block of code is almost identical to the program from the last chapter. But instead of using it in an if/else block, it's in a case in a switch statement (and it passes an argument to RollDice).

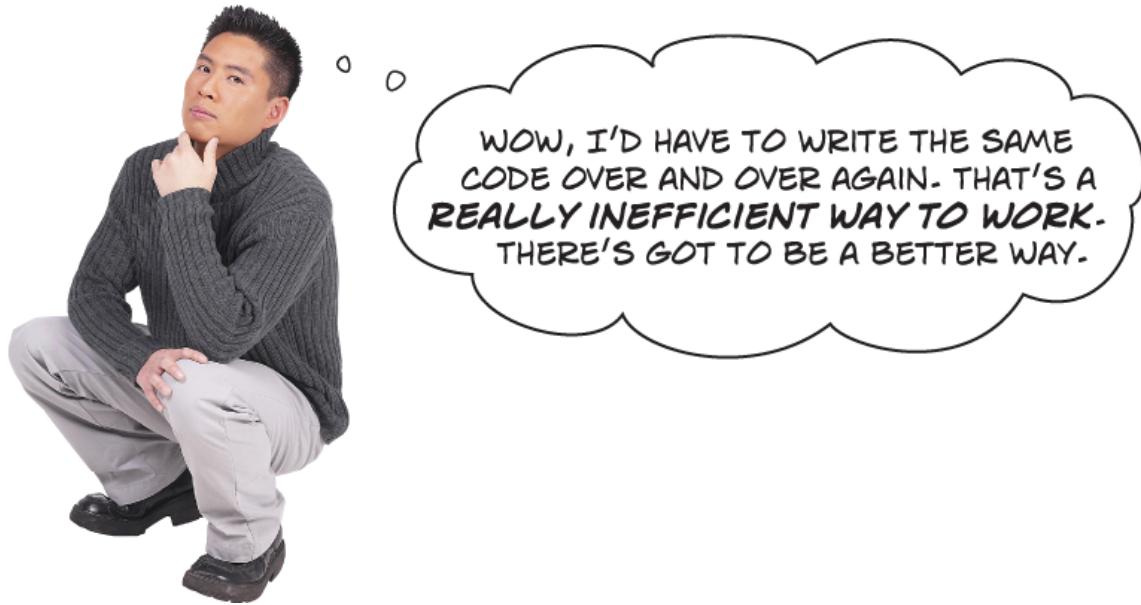
The code to use the instance of ArrowDamage to calculate damage is really similar to the code for SwordDamage. In fact, it's almost identical. Is there a way that we can reduce the duplication of code and make our program easier to read?

Set a breakpoint on the switch line, then use the debugger to step through the switch statement. That's a great way to really get a sense of how switch works. Then try removing one of the break lines and step 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...

It's easy enough to make the two classes for sword and arrow damage. But what happens if there are three other weapons? Or four? Or twelve? 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 have 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.



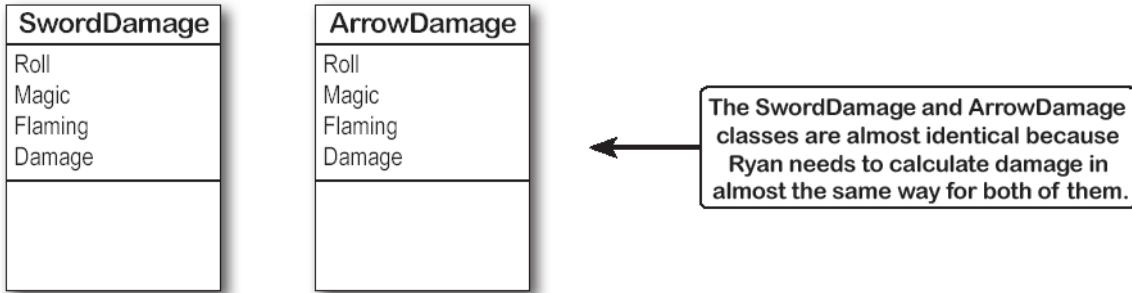


You're right! Having the same code repeated in different classes is inefficient and error-prone.

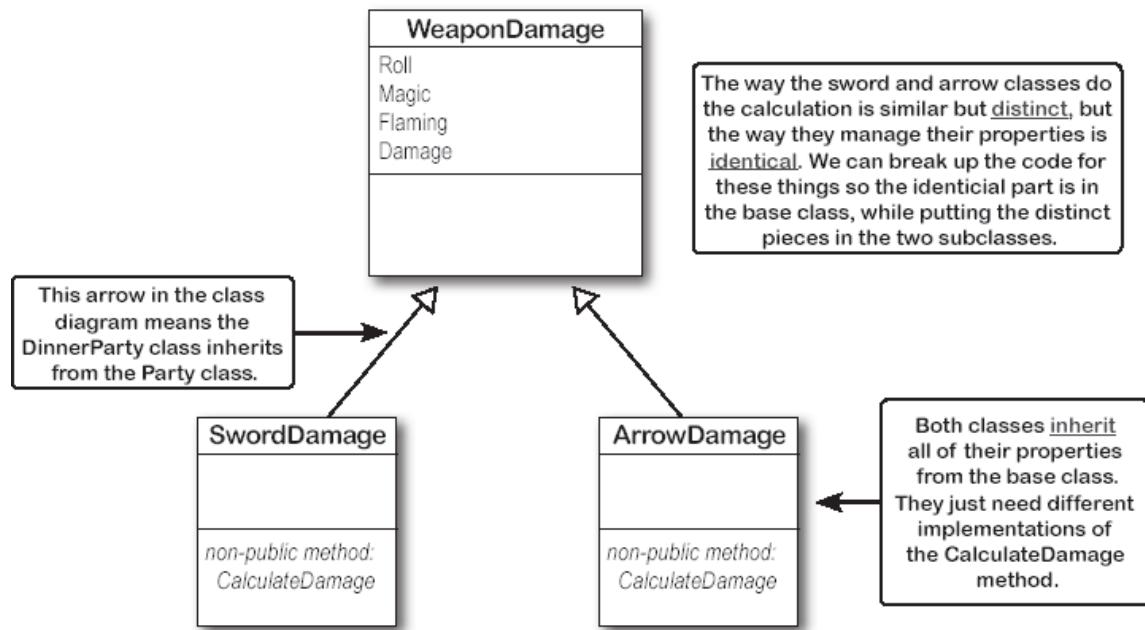
Lucky for us, C# gives us a better way to build classes that are related to each other and share behavior: **inheritance**.

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**.



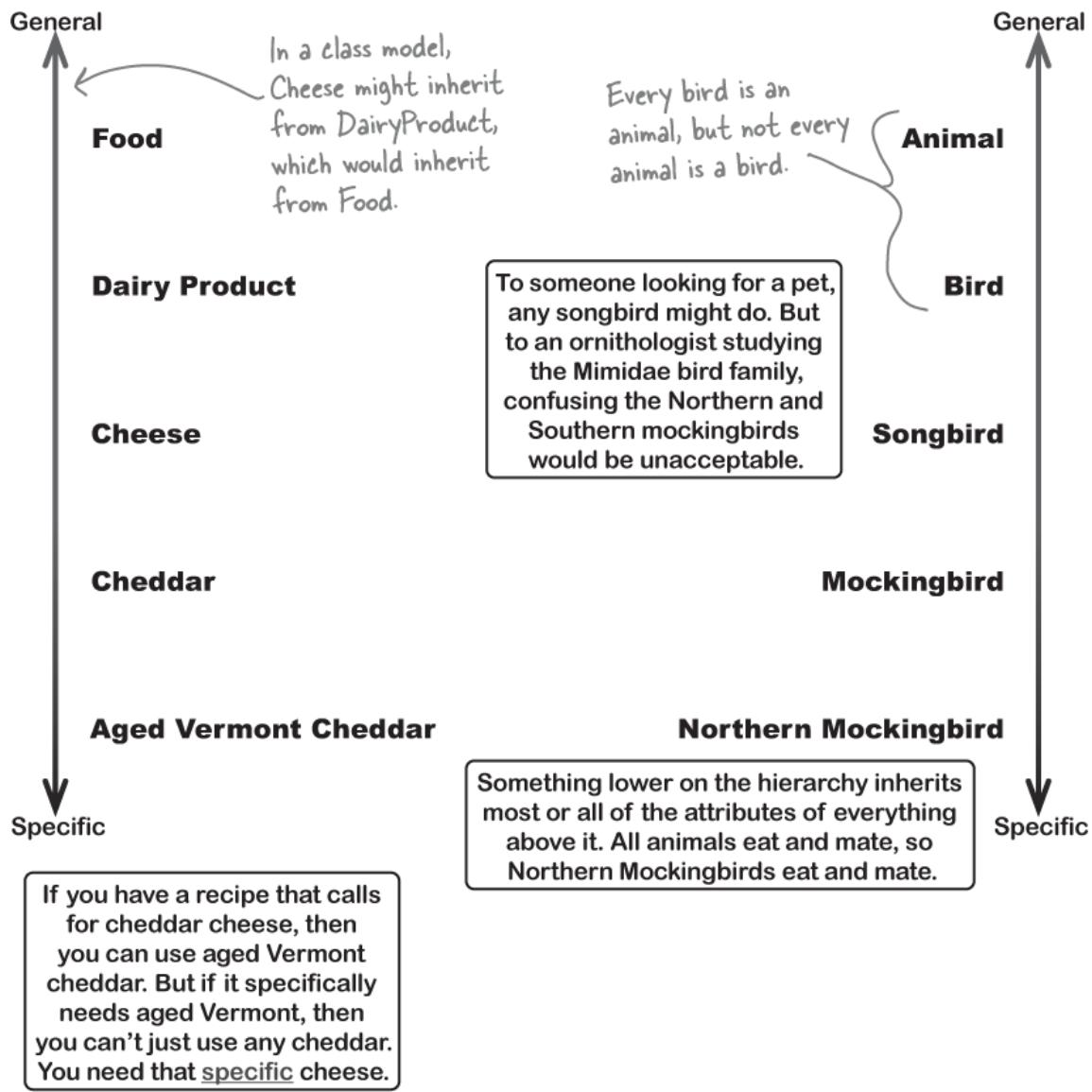
When you have two classes that are specific cases of something more general, you can set them up to **inherit** from the same class. When you do that, each of them is a **subclass** of the same **base class**.



Build up your class model by starting general and getting more specific

When you build a set of classes that represent things, it's called a **class model**. C# programs use inheritance because it mimics the relationship that the things they model have in the real world. 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 it.



in-her-it, verb.

to derive an attribute from one's parents or ancestors. *She wanted the baby to **inherit** her big brown eyes, and not her husband's beady blue ones.*

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 a game 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. But 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 class that all animal classes can inherit from.

NOTE

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.

NOTE

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...

Use inheritance to avoid duplicate code in subclasses

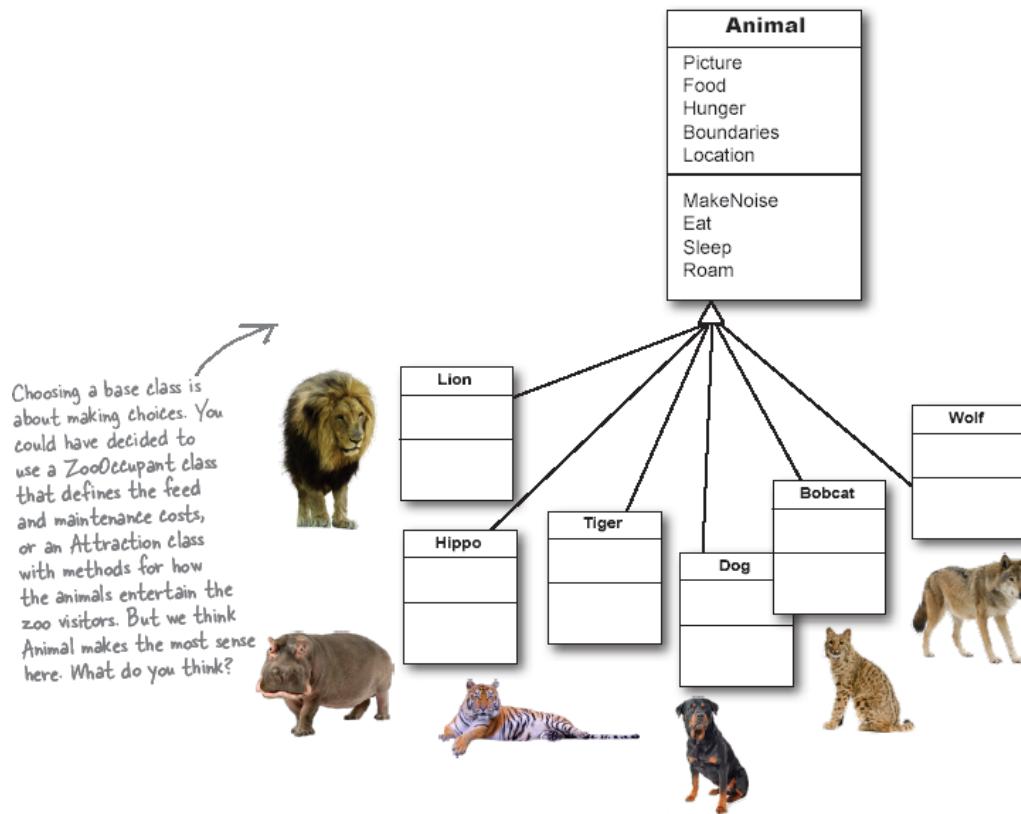
You already know that duplicate code sucks. It’s hard to maintain, and always leads to headaches down the road. So let’s choose fields and methods for an Animal base class that you **only have to write once**, and each of the animal subclasses can inherit from them. Let’s start with the public fields:

- **Picture:** an image that you can put into a PictureBox.

- **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:** the animals like to wander around their pens in the zoo.

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.



Different animals make different noises

Lions roar, dogs bark, and as far as *we* know hippos don't make any sound at all. Each 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.

NOTE

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!

3. Figure out what each animal does that the Animal class does differently—or not at all.

What does each type of animal do that all the other animals don't? Dogs eat dog food, so the dog's Eat method will need to override the Animal.Eat method. Hippos swim, so a hippo will have a Swim method that isn't in the Animal class at all.

Think about what you need to override

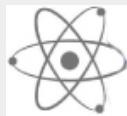
Every animal needs to eat. But a dog might take small bites of meat, while a hippo eats huge mouthfuls of grass. So what would the code for that behavior look like? Both the dog and the hippo would override the Eat method. The hippo's method would have it consume, say, 20 pounds of hay each time it was 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.

NOTE

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



BRAIN POWER

We already know that some animals will override the MakeNoise and Eat methods. Which animals will override Sleep or Roam? Will any of them? What about the properties—which animals will override some properties?

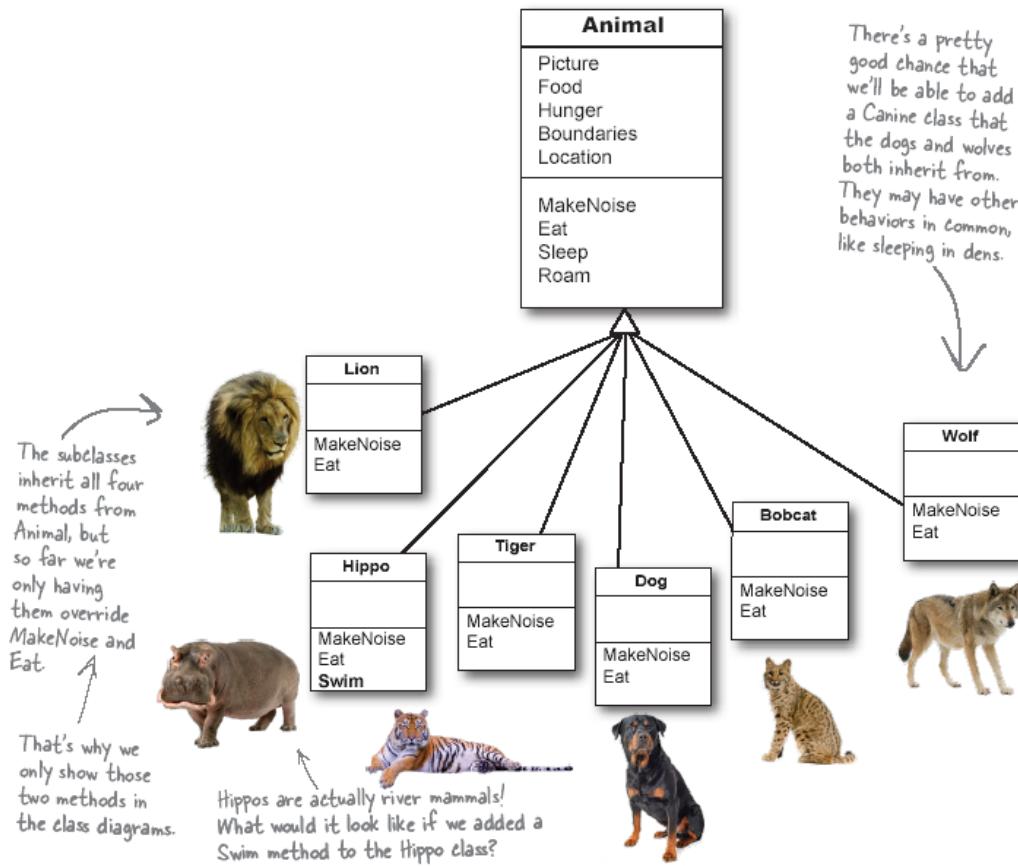
Think about how to group the animals

Aged Vermont cheddar is a kind of cheese, which is a dairy product, which is a kind of food, and a good class model for food would represent that. Lucky for us, C# gives us an easy way to do it. You can create a chain of classes that inherit from each other, starting with the topmost base class and working down. So you could have a Food class, with a subclass called DairyProduct that serves as the base class for Cheese, which has a subclass called Cheddar, which is what AgedVermontCheddar inherits from.

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 Feline class that lives between Animal

and those three feline classes that can help prevent duplicate code between them.

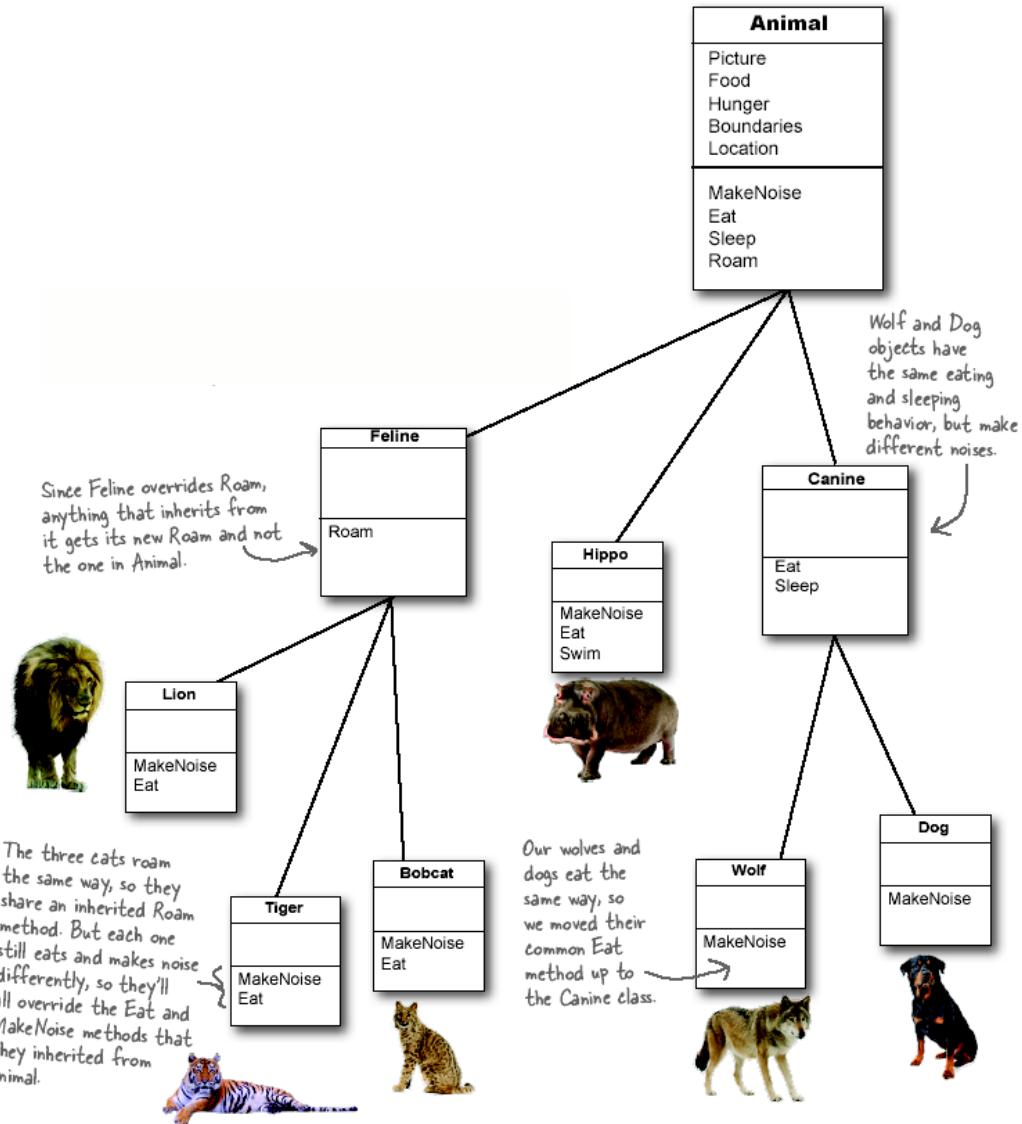


Create the class hierarchy

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. But when it comes down to it, the biggest benefit you'll get is that your code becomes really easy to understand and maintain. When you're looking at the zoo simulator code, when 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.

4. Finish your class hierarchy.

Now that you know how you'll organize the animals, you can add the Feline and Canine classes.

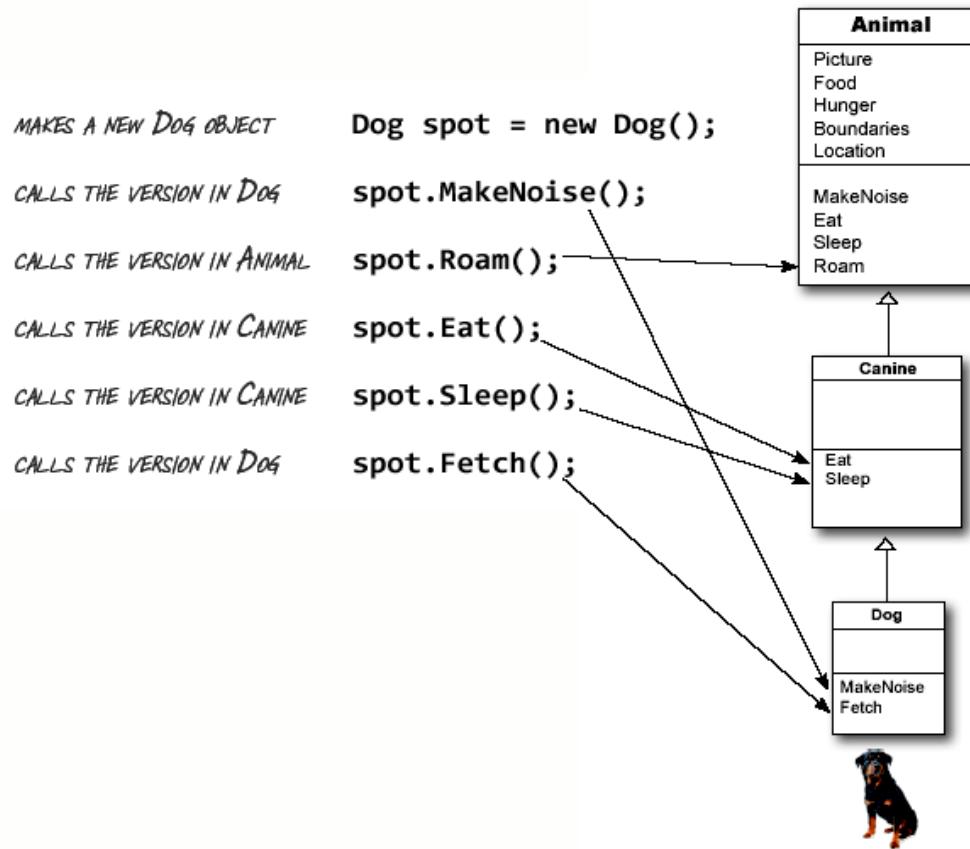


Every subclass extends its base class

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'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 add inheritance to a class, 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 wanted to add a Fetch method to Dog, that's perfectly normal. It won't inherit or override anything—only Dog objects will have that method, and it won't end up in Wolf, Canine, Animal, Hippo, or any other class.



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. But 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 do that thing. If it's in the Canine class, it's telling you how all canines do it. And 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 one, it'd then check Canine, and after that, it'd check Animal.

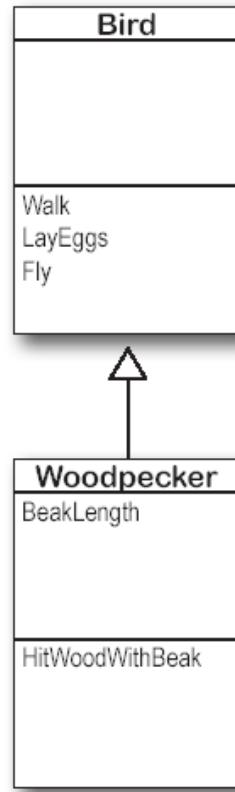
Any place where you can use a base class, you can use one of its subclasses instead

One of the most useful things you can do with inheritance is to **use a subclass in place of the base class it inherits from**. So if your method takes a Bird object, then you can pass an instance of Penguin. But if you're treating it like any bird, then you can only ask it to do things that all birds do. So you can ask it to Walk and LayEggs, but you can't ask it to Swim because only Penguins know how to swim—and your method doesn't know that it's specifically a Penguin, just that it's a more general Bird. It **only has access to the fields, properties, and 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.



So if you want to incubate some Woodpecker eggs, you can pass a Woodpecker reference to the Incubator 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(27);
    IncubateEggs(woody);
    woody.HitWoodWithBeak();
}

```

You can ***always move down*** the class diagram—a reference variable can always be set equal to an instance of one of its subclasses. But you ***can't move up*** the class diagram.

```

public void GetWoodpeckerEggs_Take_Two()
{
    Woodpecker woody = new Woodpecker(27);
    woody.HitWoodWithBeak();

    // 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
                                a kind of bird...

    // THE NEXT LINE WILL HAVE A COMPILER ERROR!!!
    Woodpecker secondWoodyReference = birdReference; ← ..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.

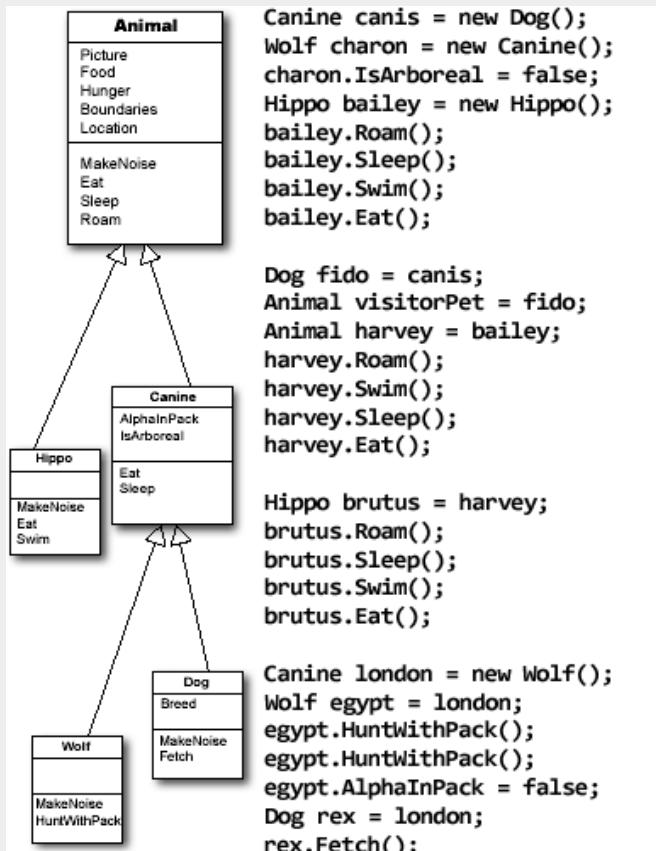
    secondWoodyReference.HitWoodWithBeak();
}

```



SHARPEN YOUR PENCIL

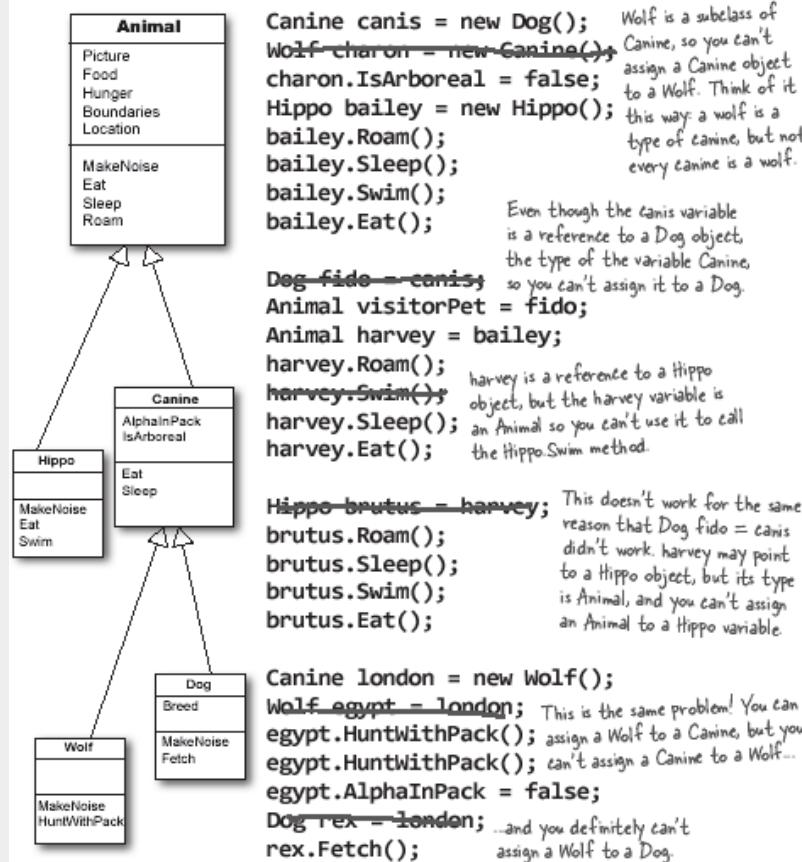
The code below is from a program that uses the class model that includes Animal, Hippo, Canine, Wolf, and Dog. Draw a line through each statement that won't compile, and write an explanation for the problem next to it.





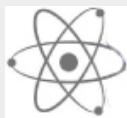
SHARPEN YOUR PENCIL SOLUTION

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.





THIS IS ALL GREAT... IN THEORY.
BUT HOW IS IT GOING TO HELP WITH MY
DAMAGE CALCULATOR APP?



BRAIN POWER

Ryan asked a really good question. Go back to the app you built for Ryan 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

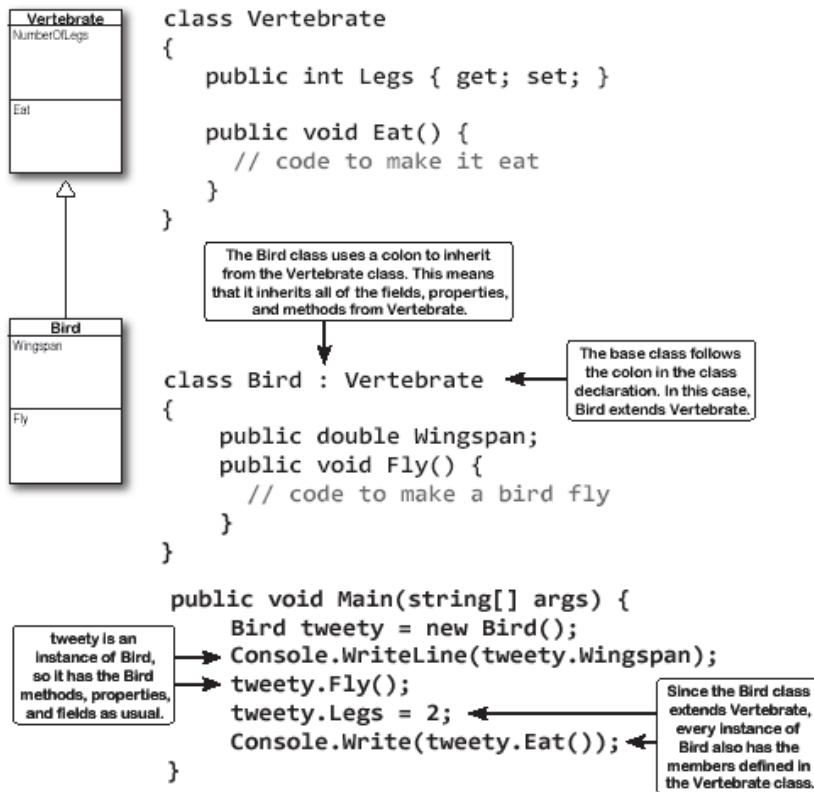
- A **switch statement** lets you compare one variable against many values. Each case executes code if its value matches. The default block runs if no cases match.
- **Inheritance** lets you build classes that are related to each other and share behavior. Use arrows to show inheritance in a class diagram.
- When you have two classes that are **specific** cases of something more **general**, you can set them up to inherit from the same class. When you do that, each of them is a **subclass** of the same **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 superclasses.
- 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, but **subclass** can also be used as a verb. We say that a subclass **extends** its base class.
- 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 **move down the class diagram**—a reference variable can always be set equal to an instance of one of its subclasses—but you can't move up the class diagram.

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:

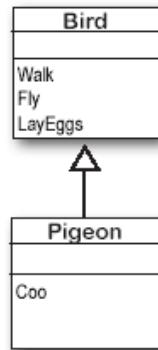
NOTE

When a subclass extends a base class, it inherits all of the fields, properties, and methods in the base class. They're automatically added to the subclass.



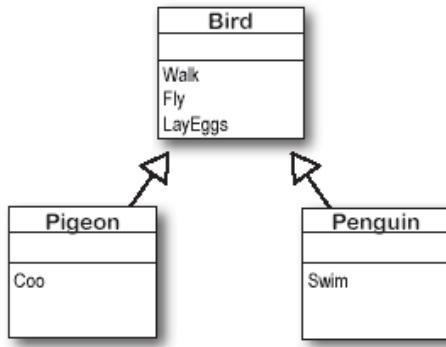
We know that inheritance adds the base class fields, properties, and methods to the subclass...

Inheritance is simple when your subclass needs to inherit **all** of the base class 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**?



NOTE

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() { ... }
}

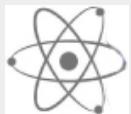
public void SimulatePigeon() {
    Pigeon Harriet = new Pigeon();

    // 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?
}

```



BRAIN POWER

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

o-ver-ride, verb.

To use authority to replace, reject, or cancel. Once she became president of Dynamco, she could **override** poor management decisions.

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 this case, that means it needs to be called Fly, return void, and have no parameters.

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 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.

You'll 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");  
    }  
}
```

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.

To override the `Fly()` method, add an identical method to the subclass and use the `override` keyword.

NOTE

Use the `override` keyword to add a method to your subclass that replaces one that it inherited. Before you can override a method, you need to mark it `virtual` in the base class.



EXERCISE

Mixed Messages

```
a = 6; → 56
b = 5; → 11
a = 5; → 65
```

Diagram showing three lines of code: a = 6;, b = 5;, and a = 5;. Arrows from each line point to the numbers 56, 11, and 65 respectively, with the first two arrows crossed out.

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.

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, ";
    }
}
```

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 we 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();
```

```
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)

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

NOTE

(Don't just type this into the IDE—you'll learn a lot more if you figure this out on paper!)



EXERCISE SOLUTION

Mixed Messages

```
a = 6; → 56  
b = 5; → 11  
a = 5; → 65
```

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 A, a2, and C 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();
```

```
class A {  
    public virtual string m1() { ... }  
    public virtual string m3() { ... }  
}  
  
class B : A {  
    public override string m1() { ... }  
}  
  
class C : B {  
    public override string m3() {  
        q += b.m1();  
        q += c.m2();  
        q += a.m3(); }  
        _____  
        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(); }
```

Annotations from right to left:

- 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

THERE ARE NO DUMB QUESTIONS

Q: A switch statement does exactly the same thing a series of if/else if statements, right? Isn't that redundant?

A: Not at all. There are many times that switch statements are much more readable than if/else if 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 ten different options. What would ten if/else if statements in a row look like? We think that 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 the 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 if statements are easier. It's up to you to write code in a 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 look better, 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. And that makes sense when you think about it from the perspective of the base class. 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 that inherited from it. 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's part of the subclass, and points to the base class that it inherits from.



EXERCISE

Here's chance to 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 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 called the methods from outside the object—like how the Main method in the code you just wrote calls LayEggs. But inheritance really shines when the base class **uses a method 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";
    }
}
```

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.

VendingMachine is the base class for all vending machines. It has code to dispense items, but those items aren't defined. And 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;
    }
}
```

NOTE

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:

Debug 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 through every line of code the debugger. 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. `AnimalVendingMachine` also overrides this property, it returns "a handful of animal feed".

NOTE

You've been using the Visual Studio debugger to sleuth out bugs in your code. But 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?



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 can be extended—without it, you can't override it at all. And 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.

NOTE

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. And 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 with valuable jewels—you'll build a class for some sneaky thieves to steal the jewels.



1. Create a new console application and add the Safe class.

We're going to be doing a little experiment with virtual and override. Go ahead and **create a new .NET console app**, then add this Safe class.

Do this!

```

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.Com combination = safeCombination;
    }
}

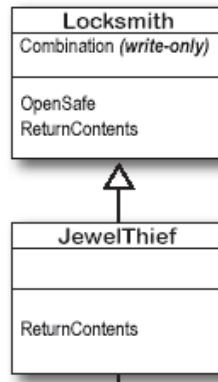
```

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.

We've got a forgetful safe owner who occasionally forgets his extremely secure safe password. Add a SafeOwner class to represent him.



```

class SafeOwner
{
    private string valuables = "";
    public void ReceiveContents(string
safeContents)
    {
        valuables = safeContents;
        Console.WriteLine($"Thank you for
returning my {valuables}!");
    }
}

```

```
    }  
}
```

3. The Locksmith class can pick the lock.

If a safe owner hires a professional locksmith to open your safe, they expect that locksmith to return the contents safe and sound. And 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. The 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 JewelTheif 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);
}

```



MINI SHARPEN YOUR PENCIL

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 superclass

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
```

NOTE

C# is supposed to call the most specific method, right? Then why didn't it call JewelThief.ReturnContents?

It looks like the JewelThief object acted just like a Locksmith object! So what happened?

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.

There's a big difference. When a subclass hides the 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 its own 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..**

NOTE

If a subclass just adds a method with the same name as a method in its superclass, it only hides the superclass method instead of overriding it.

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 program still won't act the way you expect it to!

It still calls the `ReturnContents` method defined in the `Locksmith` object. 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 the `ReturnContents` method, its own `ReturnContents` will never be called.

Use different references to call hidden methods

Now we know that the 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. And that means your class needs **two different ways to call it**.

And, in fact, we already have 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. But 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 the 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: What did you mean by only being able to move up the class diagram but not being able to move down?

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). When you think about it that way, it's easy to see how if all you need is a vehicle, a car or van or motorcycle will do. But 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. But if the method takes Motorcycle as a parameter, you can't pass any Vehicle object, because it may be a Van instance. Then C# wouldn'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, and it's called **overriding**. And it's very easy to get your class to do it. The first thing you need to do is use 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 easy error to fix! Just 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 what you should see the output we're looking for:

```
I'm stealing the jewels! I stole: precious jewels
```



SHARPEN YOUR PENCIL

Each of the following sentences describes a keyword used in the method declaration. Write down the keyword described by each sentence.

1. A method that can only be **accessed by an instance of the same class** is marked
 2. A method that **a subclass can replace** with a method of the same name is marked
 3. A method that can be **accessed by an instance of any other class** is marked
 4. A method that **hides another method in the superclass** with the same name is marked
 5. A method that **replaces a method in the superclass** is marked
 6. A method that can only be **accessed by a member of the class or its subclass** is marked
.....
-
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?



NOTE

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.

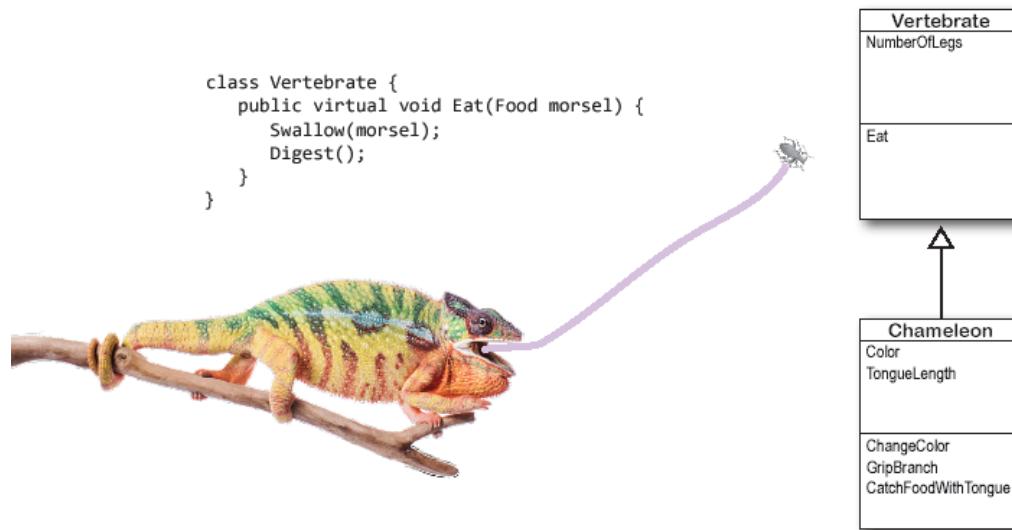
Exactly. Most of the time you want to override methods, but hiding them is an option.

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!

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.

1. All animals eat, so the `Vertebrate` class has an `Eat` method that takes a `Food` object as its parameter.



2. 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.

3. 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.

```

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 class. We want to add a BrokenEgg class that extends egg, and make 25% of the eggs that the Pigeon lays broken. So **replace the new statement** in Pigeon.LayEgg with this if/else that creates a new instance of a BrokenEgg subclass of Egg:

Add this!

```

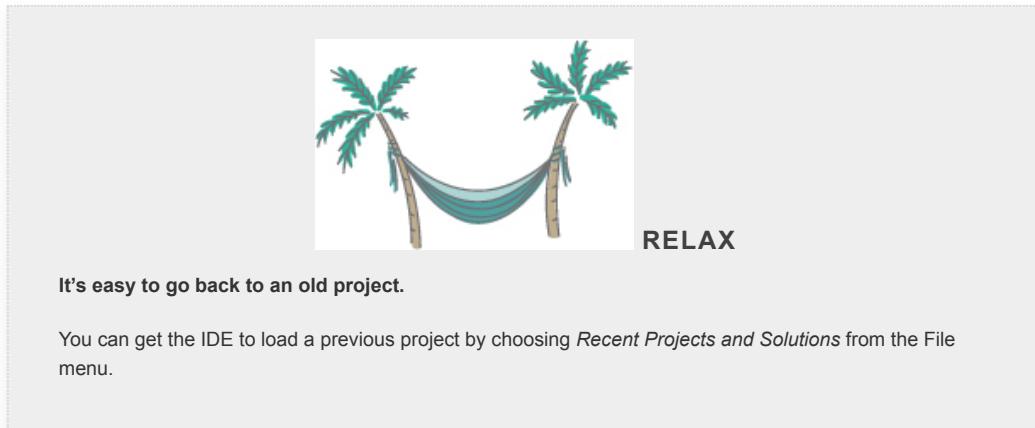
if (Bird.Randomizer.Next(4) == 0)
    eggs[i] = new BrokenEgg (Bird.Randomizer.NextDouble() * 2 + 1,
    "white");
else
    eggs[i] = new Egg(Bird.Randomizer.NextDouble() * 2 + 1,
    "white");

```

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. But what about the other error?

Let's break down what that error says.

- It's complaining about *Egg.Egg(double, string)* – that's talking about 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.

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. But 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:

Modify this!

```
if (Bird.Randomizer.Next(4) == 0)
    eggs[i] = new BrokenEgg("white");
else
    eggs[i] = new Egg(Bird.Randomizer.NextDouble() * 2 + 1,
"white");
```

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");
    }
}
```

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.

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 field, so 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?

HOW'S THE WEATHER UP THERE?



It's time to finish the job for Ryan

The first thing we did in this chapter was to modify the damage calculator you built for Ryan 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**. And we learned that having code repeated in different classes is inefficient and error-prone, especially if you want to keep extending this program to add more classes for different kinds of weapons. But now you have a new tool to solve this problem: **inheritance**. So it's time for you to finish the damage calculator app—and 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.





SHARPEN YOUR PENCIL

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 fill in the rest of the 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 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

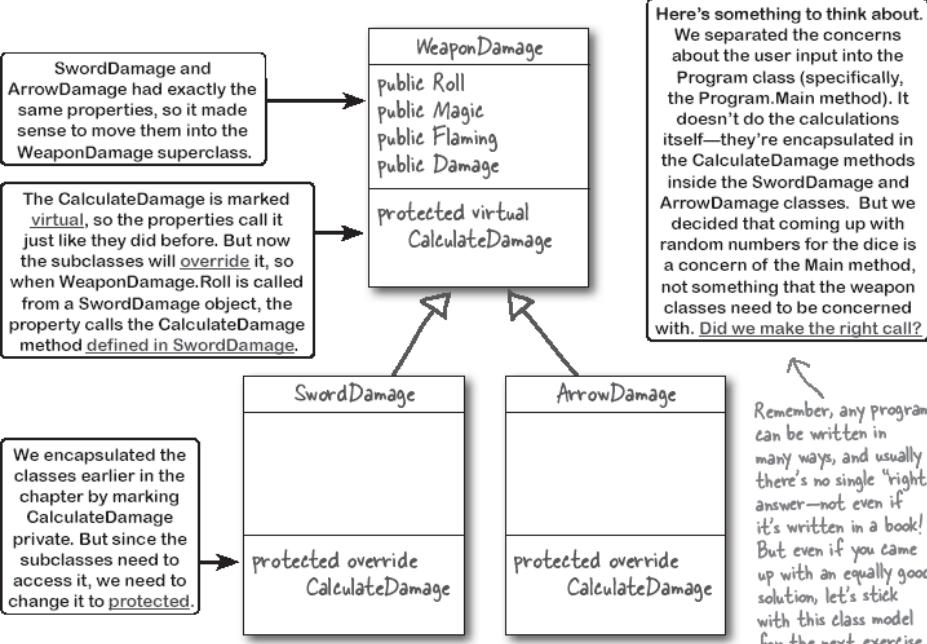
When you design your classes well today, they'll be easier to modify later. Imagine if you had a dozen different classes to calculate damage for different weapons. What if you wanted to change Magic from a bool to an int, so you could have weapons with enchantment bonuses (like a +3 magic mace or +1 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 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.



SHARPEN YOUR PENCIL SOLUTION





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 Ryan. You can re-open 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.

1. **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.
2. **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 in 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.
3. **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.
4. **Implement the ArrowDamage class.** It works exactly like SwordDamage, except that its CalculateDamage method does the arrow calculation and not the sword calculation.



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();
    }
}
```

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. But 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. So 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.



EXERCISE SOLUTION

The SwordDamage class extends WeaponDamage and overrides its CalculateDamage method to implement the sword damage calculation. Here's the code:

NOTE

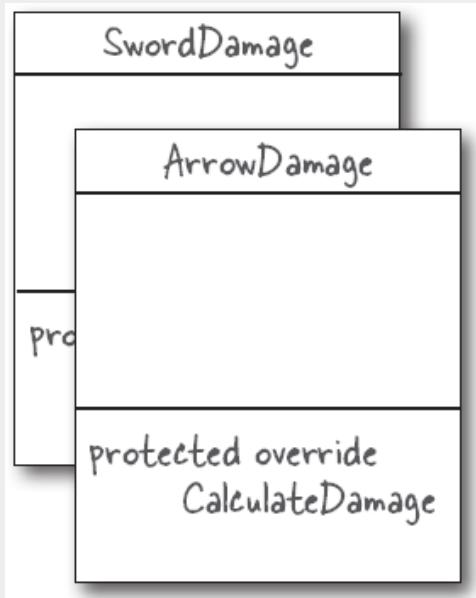
All the constructor needs to do is use the base keyword to call the superclass's constructor, using its startingRoll parameter as the argument.

```
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;
    }
}
```



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);
    }
}
```

NOTE

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.



GAME DESIGN... AND BEYOND

Dynamics

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 Ryan'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**, Ryan will need to change his campaigns. For example, some battles designed to be difficult may suddenly become too easy for the players. And 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*. Ryan 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. But 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.



NOTE

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.



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

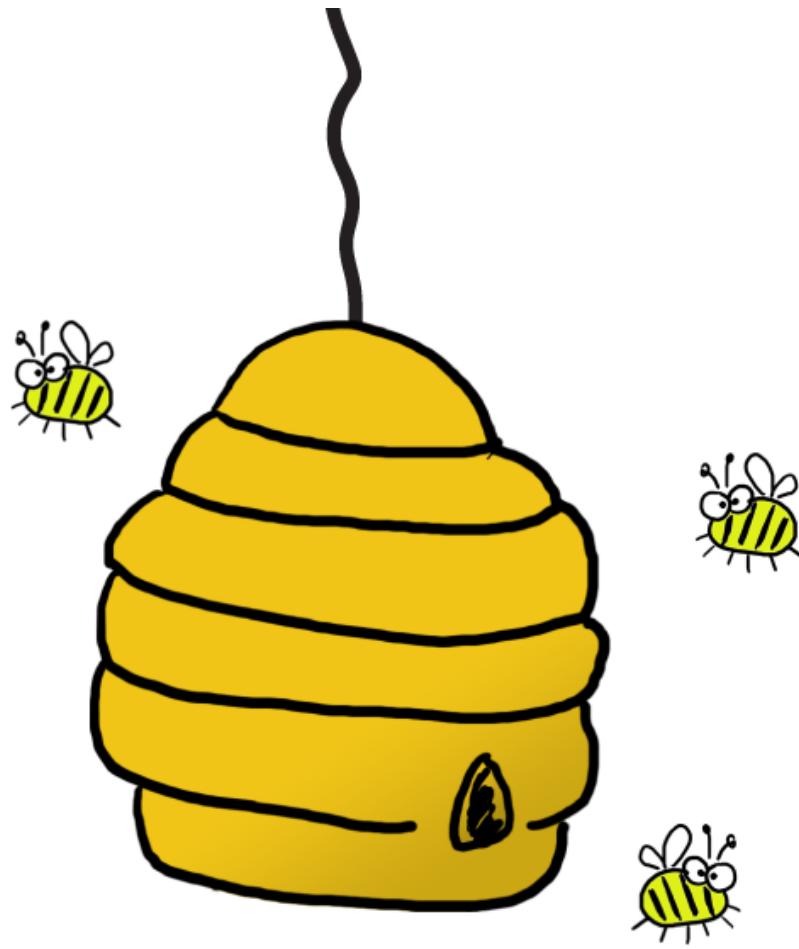
a talented game designer can break into! There's an entire ecosystem of **independent game developers** who build and sell games, either as individuals or on small teams.

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:

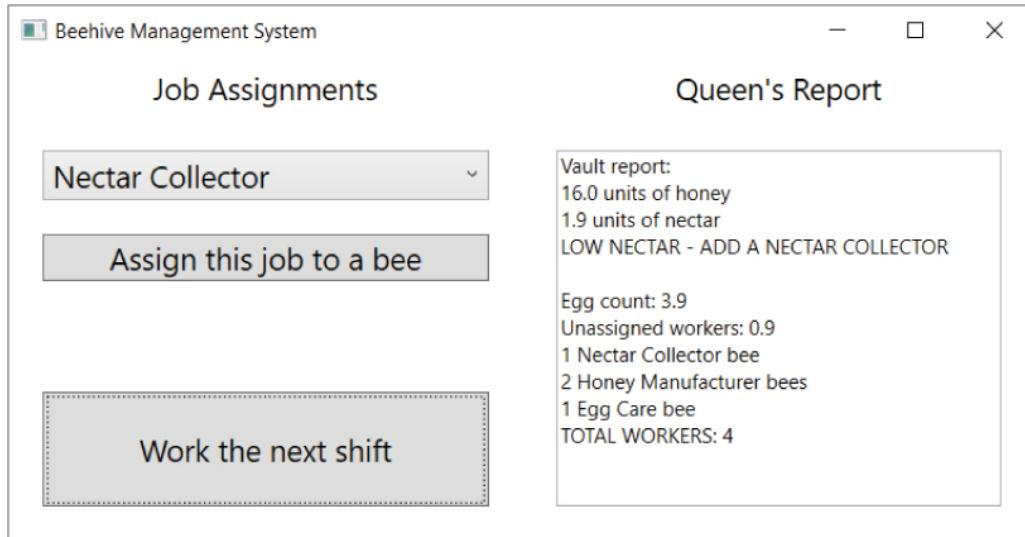


1. The queen assigns jobs to her workers.

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. And 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**.



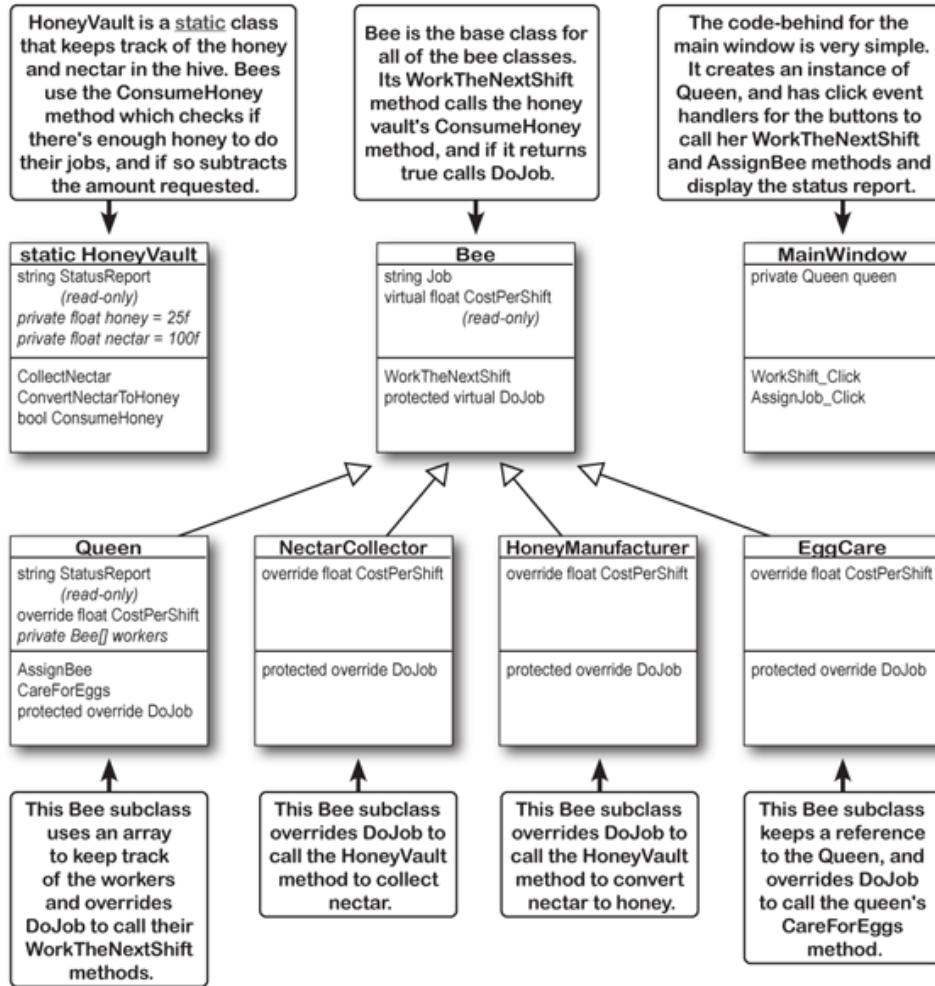
3. Help the queen grow her hive.

Like all business leaders, the queen is focused on **growth**. But the bee business is hard work! 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-nkrupcy?

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.

Examine this class model really carefully. It has a lot of information about the app you're about to build.



NOTE

We'll give you all of the details you need to write the code for these classes.

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. And the status report under Queen's Report is displayed in a **TextBox** control.

The grid has two columns with equal widths

It has three rows with widths (top) 3* 4* 1* (bottom)

Beehive Management System

Job Assignments

Nectar Collector

Assign this job to a bee

Queen's Report

Work the next shift

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. And 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"/>

```

The dropdown list is a **ComboBox** control. It's a container control (like **Grid** and **StackPanel**) 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 Property 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.

The Queen class: how she manages the worker bees

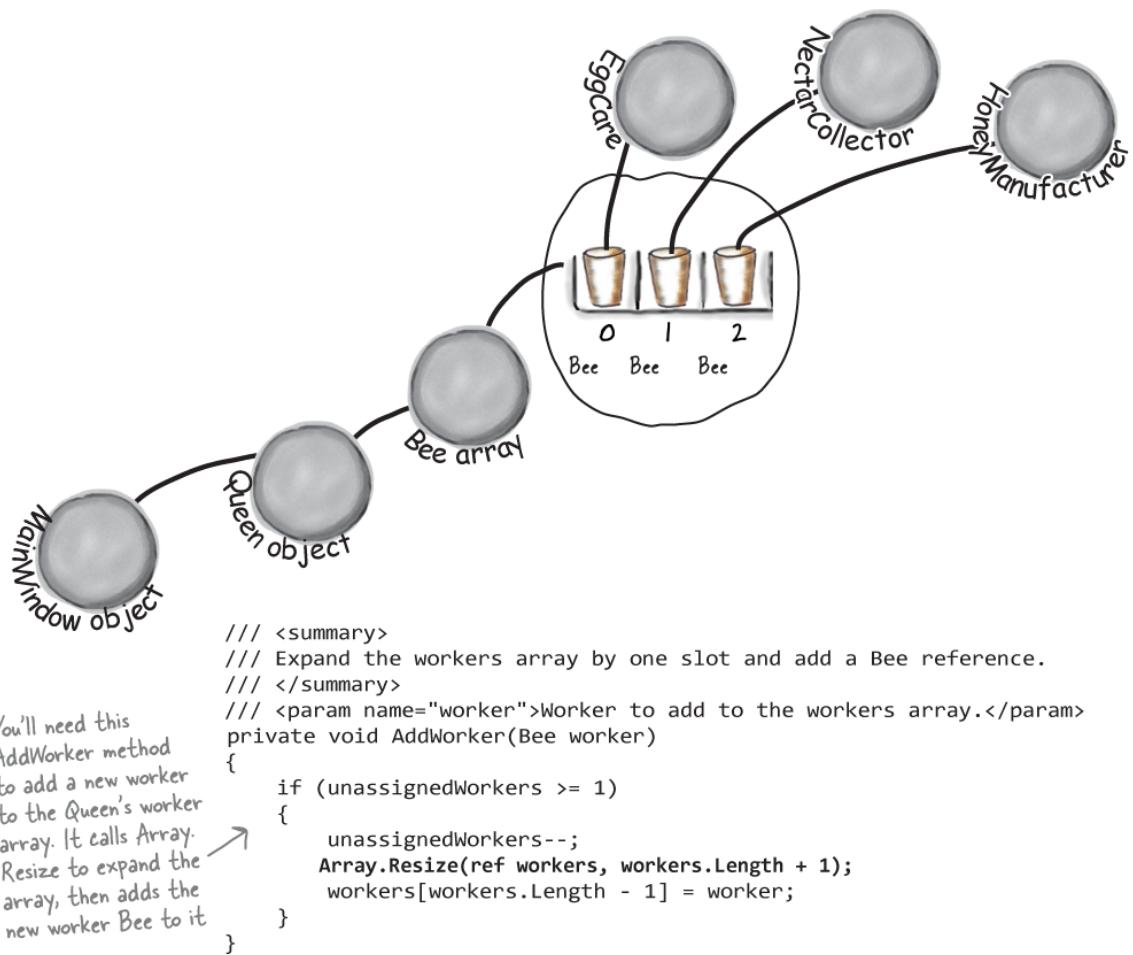
When you **press the button to work the next shift**, the button's Click event handler calls the Queen object's `WorkTheNextShift` method, which is inherited from the Bee base class. Here's what happens next:

- 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** to your Queen class.

NOTE

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.



NOTE

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 that 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 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 their 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.

NOTE

Every single part of this exercise is something you've seen before. You CAN do this!

This is a long exercise, **but that's okay!** Just build it and class by class. Finish building the Queen class first. Once you're done, you'll go back to the other Bee subclasses.

- She 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 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 3 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 button and combo box.
- Hook up the **event handlers**. The “assign jobs” button calls queen.AssignBee(jobSelector.Text). The “Work the next shift” button calls call 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**. But 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**. But 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; ← The constants in the HoneyVault class
    public const float LOW_LEVEL_WARNING = 10f; ← are really important. Try making the
    private static float honey = 25f; ← nectar conversion ratio bigger—that
    private static float nectar = 100f; ← adds lots of honey to the vault each
                                         shift. Try making it smaller—now the
                                         honey disappears almost immediately.

    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; } } The HoneyManufacturer
        nectar -= nectarToConvert;                                         bees do their jobs by calling
        honey += nectarToConvert * NECTAR_CONVERSION_RATIO;             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.
        return false; } } The Consumethoney method only
                                         returns true if there's enough honey
                                         for the bee to do its job.

    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;
        }
    }
}

```

It's okay if your code doesn't exactly match our code! There are many different ways that you can solve this program—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.

NOTE

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 helps you explore 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) ← 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.
    {
        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 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);
    }
}
```

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.

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" +
        $"Egg count: {eggs:0.0}\nUnassigned workers: {unassignedWorkers:0.0}\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. And 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 what to include here.

The EggCare bees call the CareForEggs method to convert eggs into unassigned workers.

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. But she's not a micromanager—she lets the individual bee instances do their own jobs and decide how much honey to consume.

```

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.

NOTE

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.

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 and 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.

NOTE

If you run into trouble writing the code, it's absolutely okay to look at the solution!



HEY, WAIT A MINUTE. THIS... THIS ISN'T
A SERIOUS BUSINESS APPLICATION.
IT'S A GAME!

YOU GUYS REALLY SUCK.

Okay, you got us. Yes, you're right. This is a game.

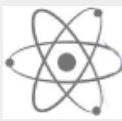
Specifically, it's a **resource management game**, or a game where the mechanics are focused on collecting, monitoring, and using resources. If you've played a simulation game like SimCity or strategy game like Civilization, you'll recognize resource management as a big part of the game, where you need 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 the 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.

NOTE

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?



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 your game works. The nectar conversion ratio has a big impact 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 Difficultly field on any of the classes. 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?

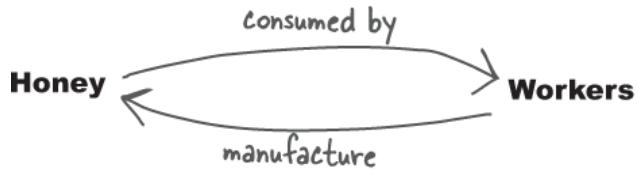
When you point a camera at a screen displaying its video output, you create a feedback loop that can cause these weird patterns.



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 **emerged** 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. Here's an example. Honey manufacturers add honey to the vault, which is consumed by honey manufacturers, who make more honey.

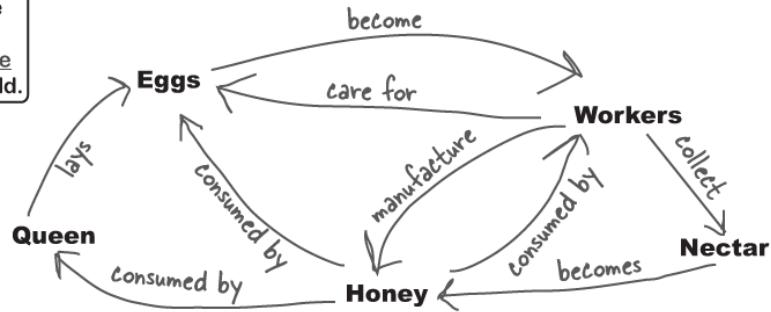


The feedback loop between the workers and honey is just one small part of the whole system that drives the game. See if you can spot it in the bigger picture below.

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*.

We'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"—and that paper actually very 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 understanding 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 of feedback loops.

- Start with mechanics of the game: the player 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, 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.

Your game is turn-based... now let's convert it to a real-time game

A **turn-based game** is a game where the flow is broken down into parts—in our case, 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. But 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. And since DispatcherTimer is in the System.Windows.Threading namespace, you'll need to add this **using** line to the top of your `MainWindow.xaml.cs` file:

```
using System.Windows.Threading;
```

NOTE

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 references.

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

```
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 simple 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.



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.

NOTE

There's a feedback loop here, too. As players feel more stress, they make worse decisions, changing the game... aesthetics feeds back into mechanics.

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.

This whole discussion of feedback loops seems pretty important - especially the part about **how behavior emerges**.



Feedback loops and emergence are important programming concepts.

We designed this project to give you practice with inheritance, but *also* let you explore and experiment with **emergent** 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. And that's all driven by the feedback loops between the eggs, worker, nectar, honey, and queen.

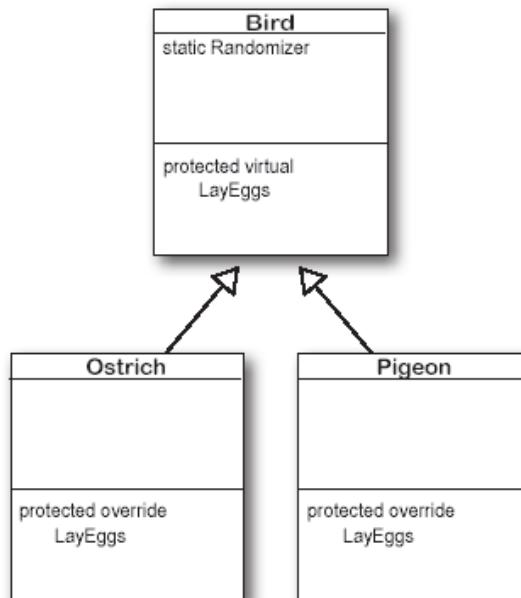
NOTE

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! The entire feel of the game changes just by making small changes to a few constants.

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. But 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 in [Chapter 5](#) that should never be instantiated.



```
class Bird
{
    public static Random Randomizer = new Random();
    public virtual Egg[] LayEggs(int numberOfEggs)
```

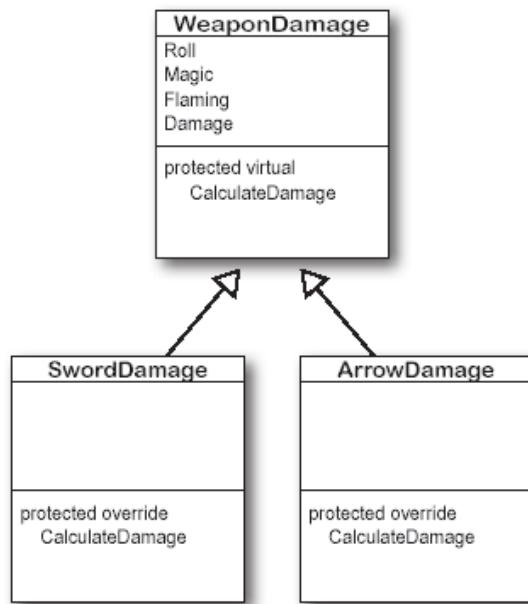
```

    {
        Console.Error.WriteLine
            ("Bird.LayEggs should never get called");
        return new Egg[0];
    }
}

```

NOTE

Your Bird class was tiny—it just had a shared instance of Random and a LayEggs method that only existed so the base classes could override it. Your WeaponDamage was a lot bigger—it had a lot of properties. It also a CalculateDamage class for the subclasses to override, which it called from its WeaponDamage method.



```

class WeaponDamage
{
    /* ... code for the properties ... */

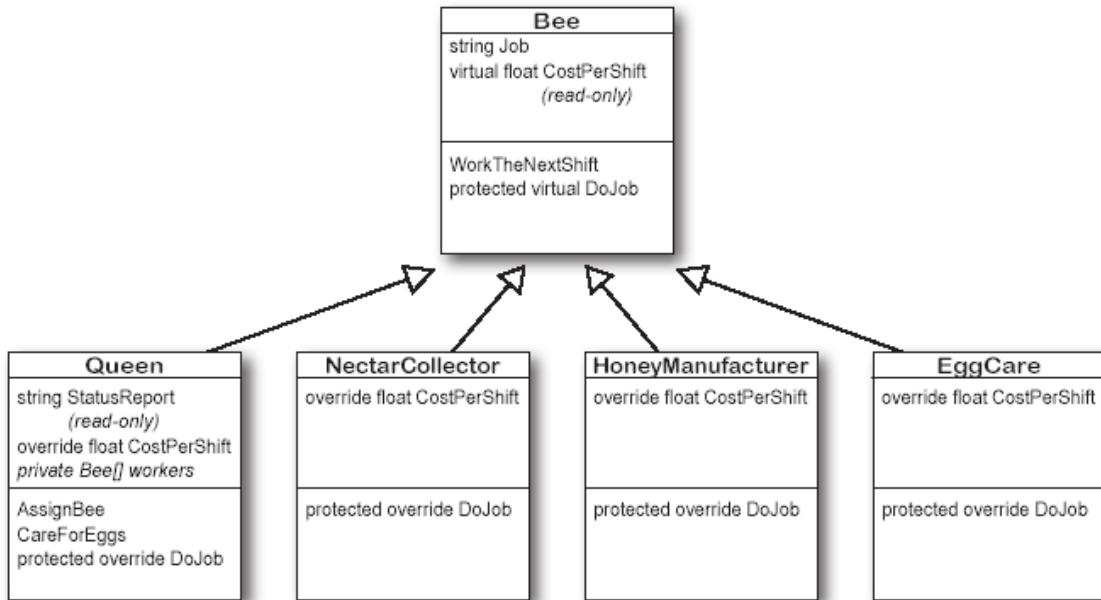
    protected virtual void CalculateDamage()
    {
        /* the subclass overrides this */
    }
}

```

```

public WeaponDamage(int startingRoll)
{
    roll = startingRoll;
    CalculateDamage();
}

```



NOTE

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;
    }
}

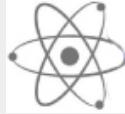
```

```
public void WorkTheNextShift()
{
    if (HoneyVault.ConsumeHoney(CostPerShift))
    {
        DoJob();
    }
}

protected virtual void DoJob() { /* the subclass overrides this */
}
```

NOTE

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.



BRAIN POWER

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. They 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. And all you need to do to make a class abstract is **add the abstract keyword to the class declaration.**

- **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 you can inherit from other classes, too, exactly like with a normal class. There's almost nothing new to learn

- **An abstract class is can have incomplete “placeholder” members.**

Abstract classes 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 it 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:

The compiler won't let you instantiate an abstract class because abstract classes are not meant to be instantiated.



 CS0144 Cannot create an instance of the abstract class or interface 'MyAbstractClass'



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...***

Do this!

```
class PlanetMission
{
    protected float fuelPerKm;
    protected long kmPerHour;
    protected long kmToPlanet;

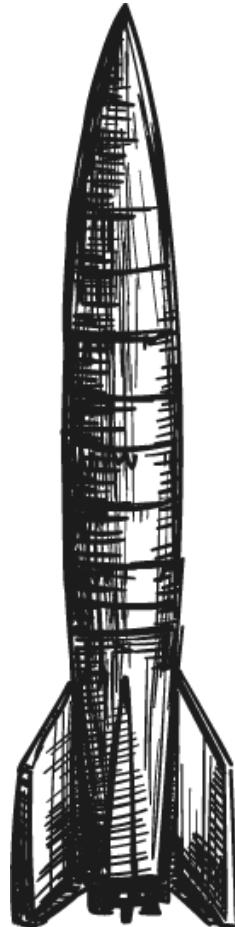
    public string MissionInfo()
    {
        long fuel = (long)(kmToPlanet * fuelPerKm);
        long time = kmToPlanet / kmPerHour;
        return $"We'll burn {fuel} units of fuel in {time} hours";
    }
}

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());
    }
}
```



NOTE

Before you run this code, can you figure out what it will print to the console?

Like we said, some classes should never be instantiated

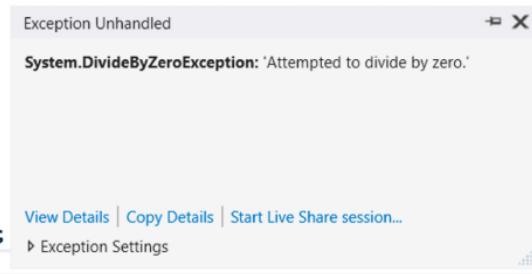
Try running the PlanetMission console app. Did it do what you expected? It printed two lines to the console:

```
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. But when they aren't, they get their default values—zero. And 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 4](#). Making some members private doesn't change the behavior. It just prevents your code from building if you break the encapsulation. And the `abstract` keyword works the same way, keeping your code from building try to instantiate an abstract class.

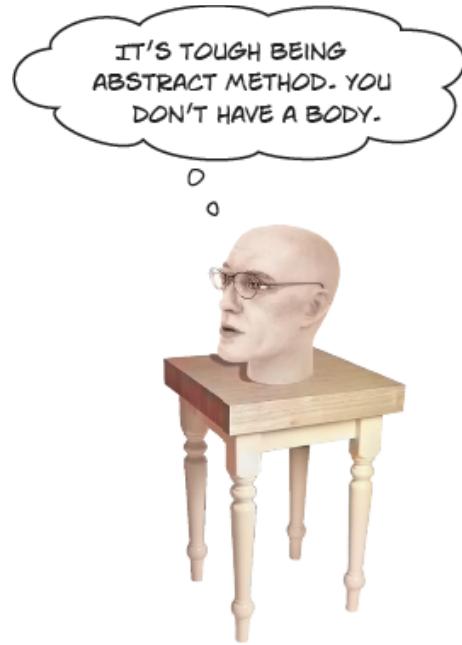
NOTE

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 in [Chapter 5](#) 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 number_of_eggs)
    {
        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**, because we never want them to leave it out and inherit the version of LayEggs that

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 [Chapter 5](#) 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

NOTE

If an abstract class has virtual members, every subclass must override all of those members.

Abstract properties work just like abstract methods

Let's go back to the Bee class from [Chapter 5](#). 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 abstract 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 4](#) that properties are really just methods that are called like fields. And we can use the **abstract keyword to create an abstract property** just like we do with a method:

```
public abstract float CostPerShift { get; }
```

Abstract properties can have either a get accessor, set accessor, or both get and set accessors. But 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 an 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 a Bee class with `new Bee()` and you'll get a compiler error. And 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.

At the end of [Chapter 5](#) 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 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 this is just to keep my code from compiling if I try to instantiate a certain kind of 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 all out the Errors window. 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 right now than it is to track down the error that it prevents. If you have 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 easy to fix.

NOTE

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

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.

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. And 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. So 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 that you've written 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 feeling 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 and refactor your code. And 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. BUT 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?

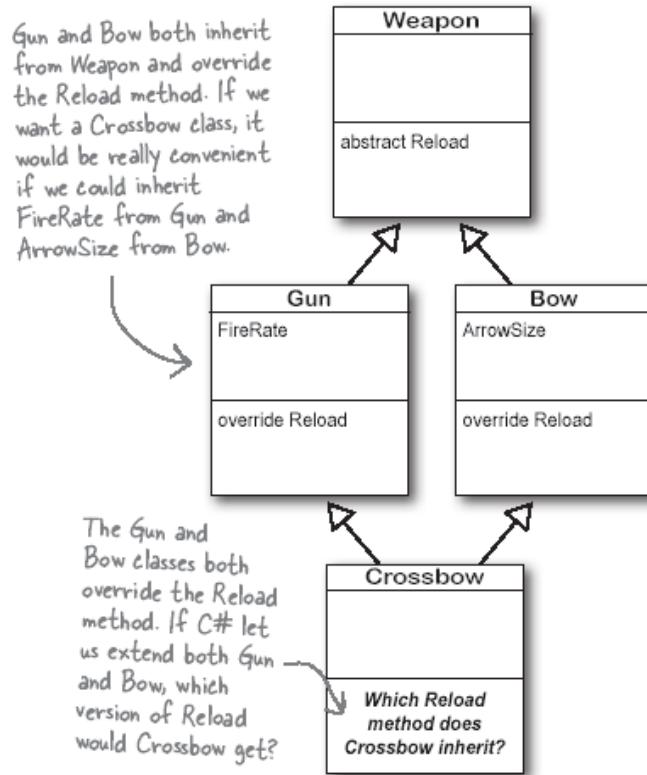
That sounds great! But there's a problem.

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**. And by giving you interfaces instead, C# saves you from a big fat class conundrum called...

The Deadly Diamond of Death

NOTE

That's its real name! Some developers just call it the "diamond problem."



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 Weapon that had an abstract method called Reload?

And what if... it had two subclasses: Gun with a FireRate property, and Bow with an Arrow Size property?

And what if... you wanted to create a Crossbow class that inherited both FireRate and ArrowSize?

And what if... C# supported multiple inheritance, so you could do that?

Then there's only one more question...

Which Reload does Crossbow inherit?

Does it get the version from Gun? Or does it get the version from Weapon?
There's no way to know!

**And that's why C# doesn't allow multiple inheritance. But
wouldn't it be convenient if we could have something like an
abstract class, but gets around the diamond problem so C# lets us
extend more than one of them at the same time?**



BULLET POINTS

- A subclass can **override** methods to change or replace members it inherited, which replaces them with a new method with the same name.
- To override a method or property, add the **virtual keyword** to the base class, then add the **override keyword** to 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 superclass, it only **hides** the superclass 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** comes from the 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 superclasses to use.



WOULDN'T IT BE DREAMY IF THERE WERE
SOMETHING LIKE AN ABSTRACT CLASS, BUT LETS US
EXTEND MORE THAN ONE OF THEM IN A WAY THAT GETS
AROUND THE DIAMOND PROBLEM?

BUT IT'S PROBABLY
NOTHING BUT A
FANTASY...

Part VI. 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 this lab is to get 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.

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 next Unity Lab you'll pick up where the last Lab 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.

The MoveToClick script uses raycasting to find the point on the floor that the player clicked and uses it to set the NavMesh Agent's destination.



NOTE

If you're using our book because you're preparing to be a professional developer, being able to go back and read the code in your old projects and 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 that your player can move around 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 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. If the mouse button is down, 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.GetMouseButtonUp 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 which point in the scene the player clicked.

- **Tell the NavMesh Agent to set that point as a destination**

The NavMeshAgent.SetDestination triggers the agent to calculate a new path and start moving towards the new destination.

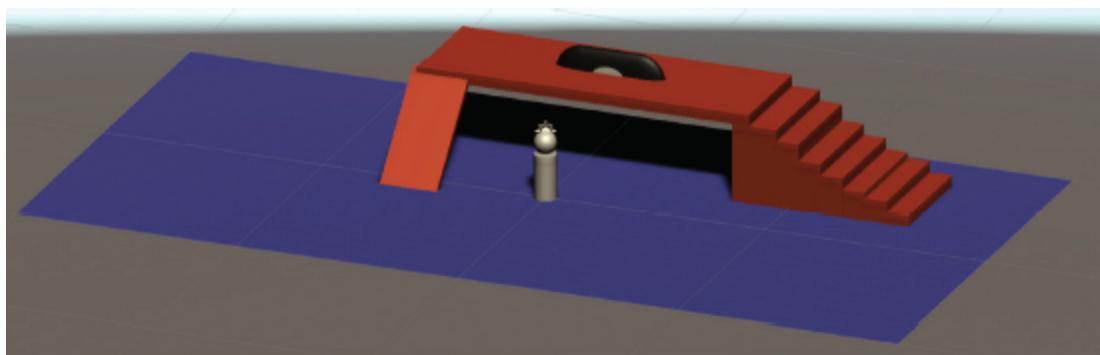
The MoveToClick method was adapted from code on the Unity manual page for the NavMeshAgent.SetDestination method. Take a minute and read it right now—choose Help >> Scripting Reference from the menu, then search for NavMeshAgent.SetDestination.

Add a platform to your scene

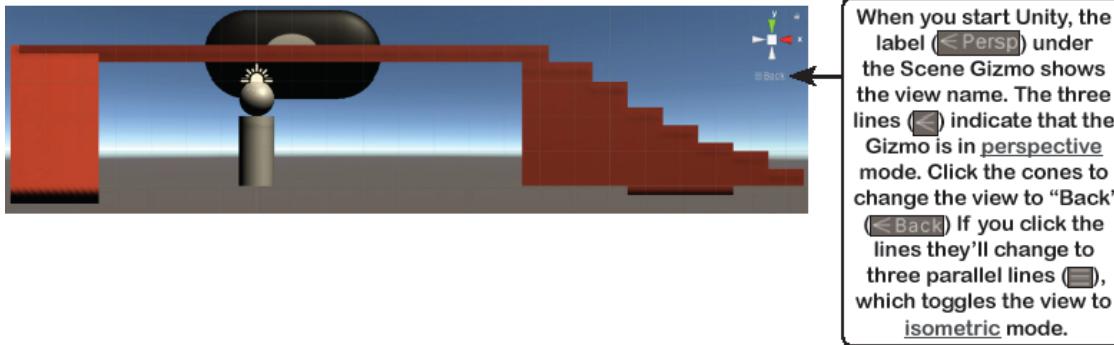
NOTE

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.

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:



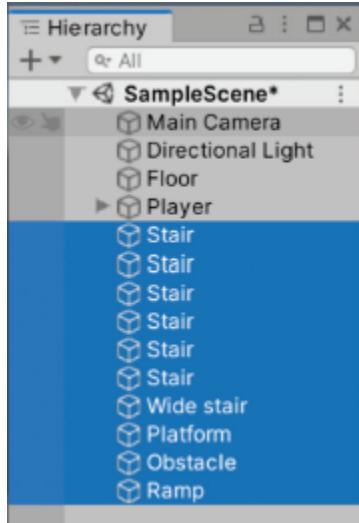
It's a little easier to see what's going on we switch to an **isometric** view, or a view that doesn't show perspective. In a **perspective** view, objects further away look small, while close objects look large. In an isometric view, objects are always the same size no matter how far away they are from the camera.



Add ten 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:

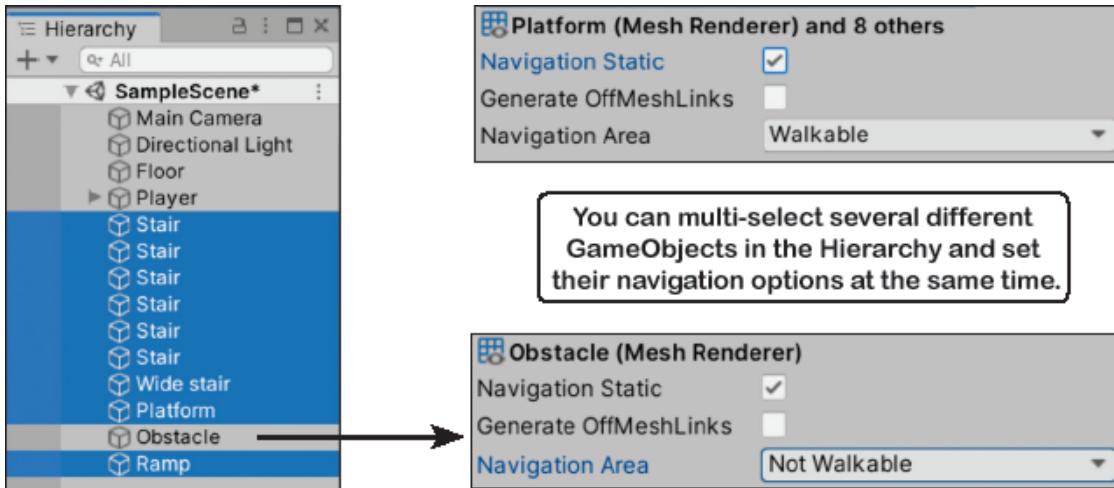
Name	Type	Position	Rotation	Scale
Stair	Cube	(15, 0.25, 5)	(0, 0, 0)	(1, 0.5, 5)
Stair	Cube	(14, 0.5, 5)	(0, 0, 0)	(1, 1, 5)
Stair	Cube	(13, 0.75, 5)	(0, 0, 0)	(1, 1.5, 5)
Stair	Cube	(12, 1, 5)	(0, 0, 0)	(1, 2, 5)
Stair	Cube	(11, 1.25, 5)	(0, 0, 0)	(1, 2.5, 5)
Stair	Cube	(10, 1.5, 5)	(0, 0, 0)	(1, 3, 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)

A Capsule is like a cylinder that has spheres on its ends.

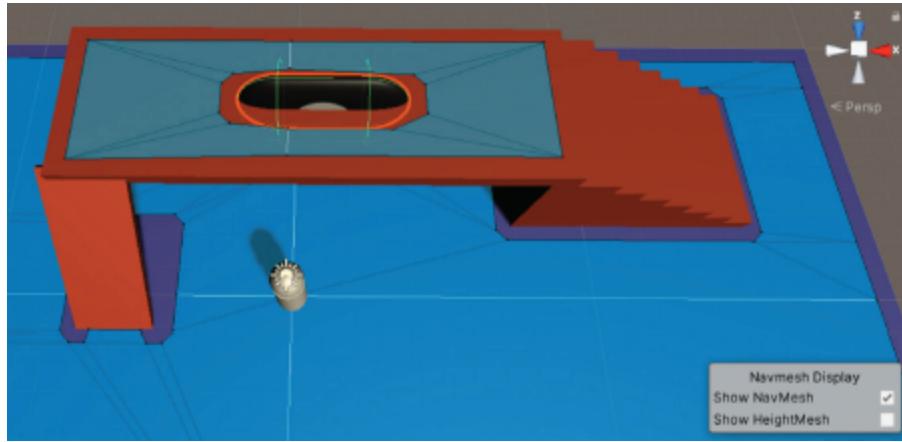


Use bake options to make the platform walkable

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.



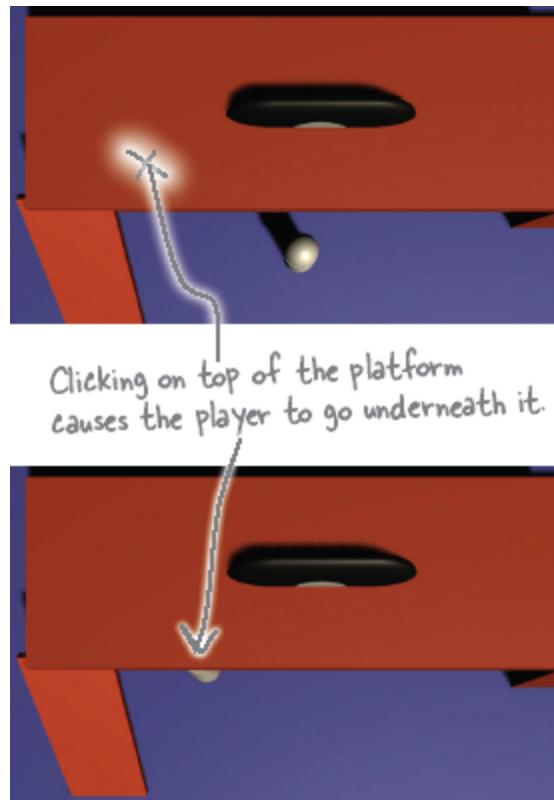
Now follow the same steps that you used before to **bake the NavMesh**: click the Bake button at the top of the Navigation window to switch to the Bake view, then click the Bake button at the bottom.



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. And if you look closely at the NavMesh that's

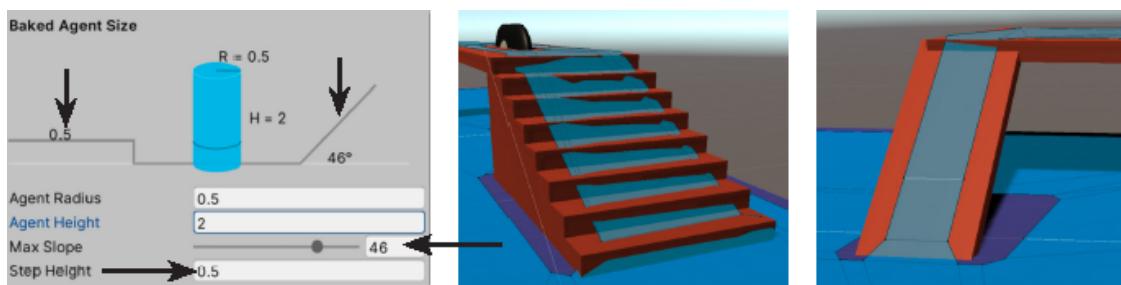
displayed when you're viewing the Navigation window, you'll see that it also has space around the stairs and ramp, but doesn't actually include either of them in the NavMesh. The player has no way to get to the point you clicked on, so the AI gets it as close as it can.



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 increase, 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, too. 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 that with at most a 45 degree incline, so **change Max Slope to 46**. Now **bake the NavMesh again**. Now it will include the ramp and stairs.



Now the NavMesh includes the ramp. Start your game and test out your new NavMesh changes.



EXERCISE

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!*

- **Create a new script called MoveCamera and drag it onto the camera.** It should have a GameObject field called Player. Drag the Player GameObject out of the hierarchy and onto the Player field in the Inspector.
- **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.GetMouseButtonUp` 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 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. And reset the Main Camera's transform to position (0, 1, -10) and rotation (0, 0, 0).

NOTE

Don't forget – it's not cheating to peek at the solution!

NOTE

It's okay 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 GameObject 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.transform.position, Vector3.up, Angle);
    }

    if (Input.GetKey(KeyCode.LeftArrow))
    {
        transform.RotateAround(Player.transform.position, Vector3.down, Angle); ←
    }
    if (Input.GetKey(KeyCode.UpArrow)) ← This is just like how you used transform.RotateAround in
        Vector3.zero (0,0,0) you're rotating around the player.
    {
        transform.RotateAround(Player.transform.position, Vector3.right, Angle);
    }

    if (Input.GetKey(KeyCode.DownArrow))
    {
        transform.RotateAround(Player.transform.position, Vector3.left, Angle);
    }

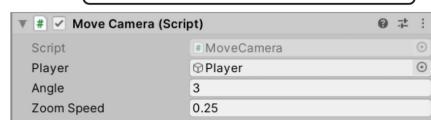
    transform.LookAt(Player.transform.position); ←
}

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.
```

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.

Use the arrow keys to move the camera so it's looking up at the player. You can see right through the floor plane!



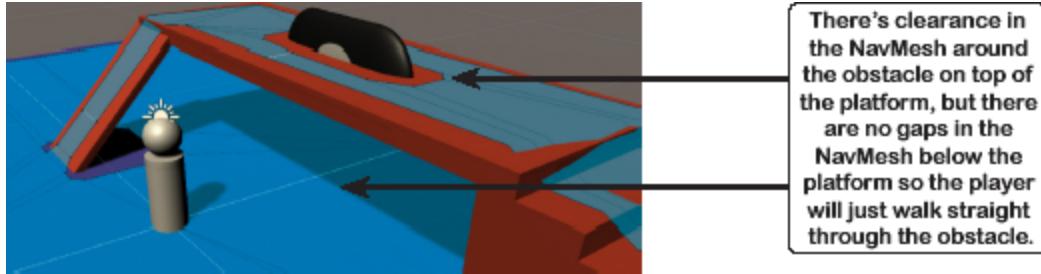
Fix height problems in the NavMesh

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 end of the ramp, too.

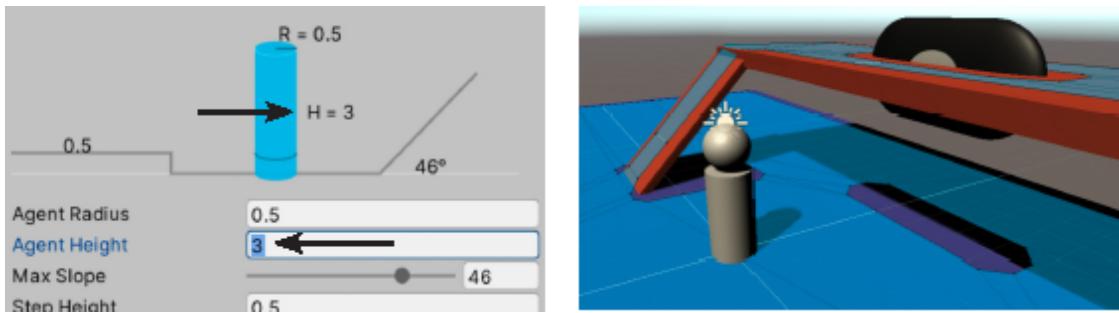


But if you move the player back to the top of the platform, it avoids the obstacle just fine. What's going on?

Look closely at the parts of the NavMesh above and below the obstacle. Notice any differences between them?



Go back to the earlier part of this Lab to the part where you set up the NavMesh Agent component—specifically, the part where you set the Height to 3. Now we 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.**

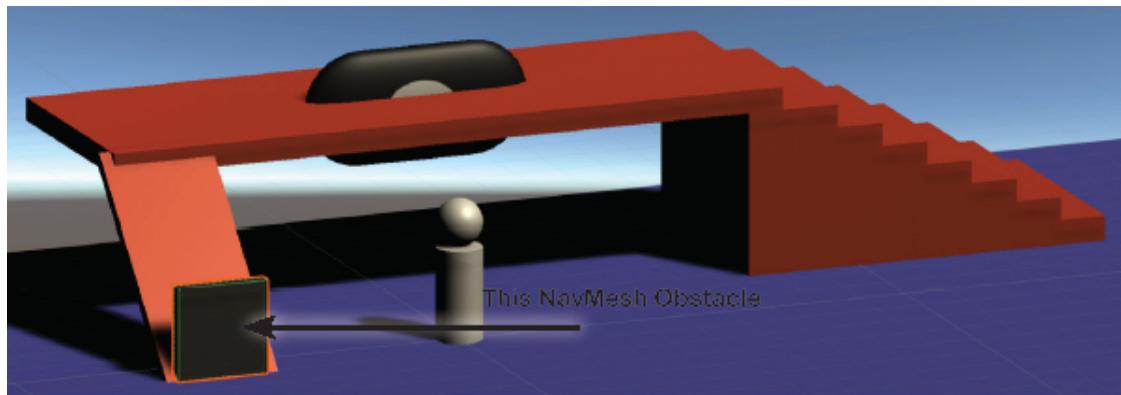


This created a gap in the NavMesh under the obstacle and expanded the gap under the ramp. Now the Player doesn't hit either the obstacle or the ramp when moving around under the platform.

Add a NavMesh obstacle

You added a static obstacle in the middle of your platform: you created a stretched out capsule and marked it nonwalkable, and when you baked your NavMesh it had a hole around the obstacle so the player has to walk around it. But 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. And 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.

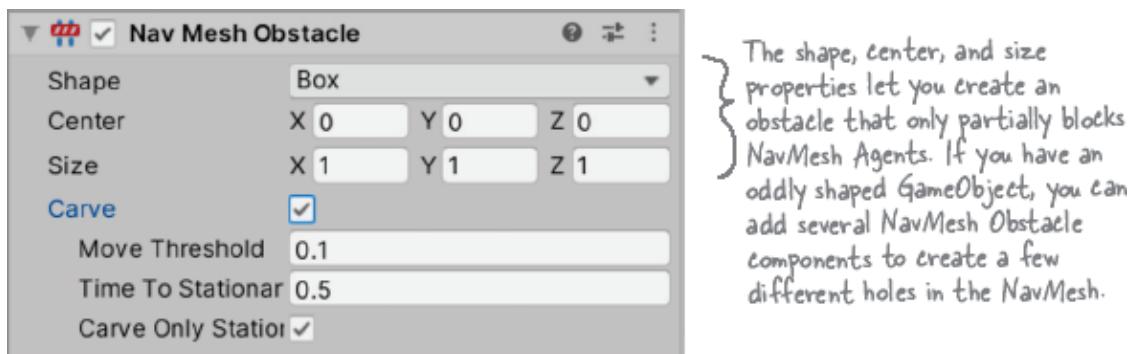
Let's do that right now. **Add a Cube to your scene** with position (-5.75, 1, -1) and scale (2, 2, 0.25). Create a new material for it with a dark gray color (333333) and name your new GameObject *Moving Obstacle*. This will act as a kind of gate that at the bottom of the ramp that can move up out of the way of the player or down to block it.



NOTE

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

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 obstacle GameObject.



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.

NOTE

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 component in the Inspector to open up the manual page. Take a minute to read it—it does a great job of explaining the options.

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 OnMouseDrag()
    {
        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.

NOTE

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.



Then **modify your MoveToClick script** to check for the tag:

```
hit.collider      if (Physics.Raycast(ray, out hit, 100))  
contains a       {  
reference to    if (hit.collider.gameObject.tag != "Obstacle")  
the object that {  
the ray hit.     agent.SetDestination(hit.point);  
}                }  
}
```

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. In the next Unity Lab we'll pick up where this one leaves off, but you don't have to wait. 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 developer.

NOTE

This is your chance to experiment. Using your creativity is a really effective way to quickly build up your coding skills.



BULLET POINTS

- Use the icon underneath the Scene Gizmo to toggle between perspective and isometric views.
- When you bake the NavMesh, you can specify **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).
- 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.

Chapter 7. Interfaces, Casting, and “is” Making Classes keep their Promises



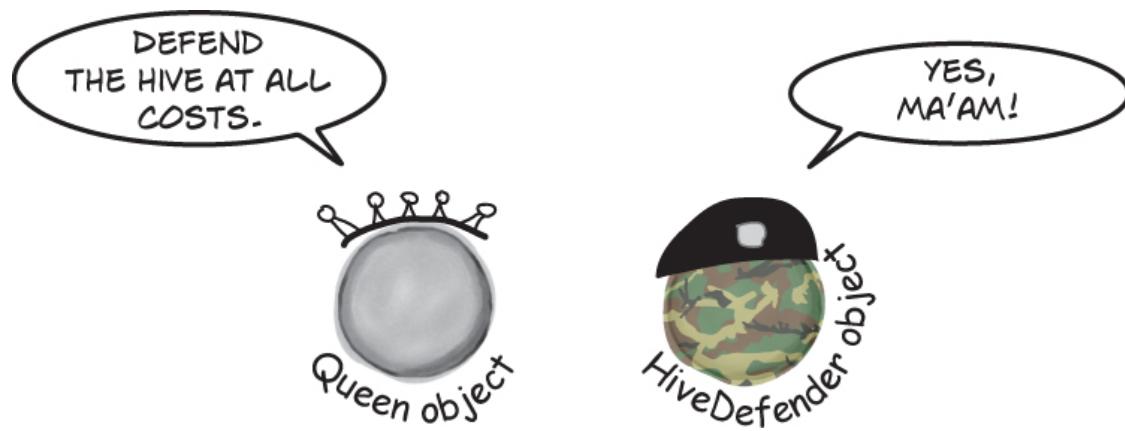
Actions speak louder than words.

Sometimes you need to group your objects together based on the **things they can do** rather than the classes they inherit

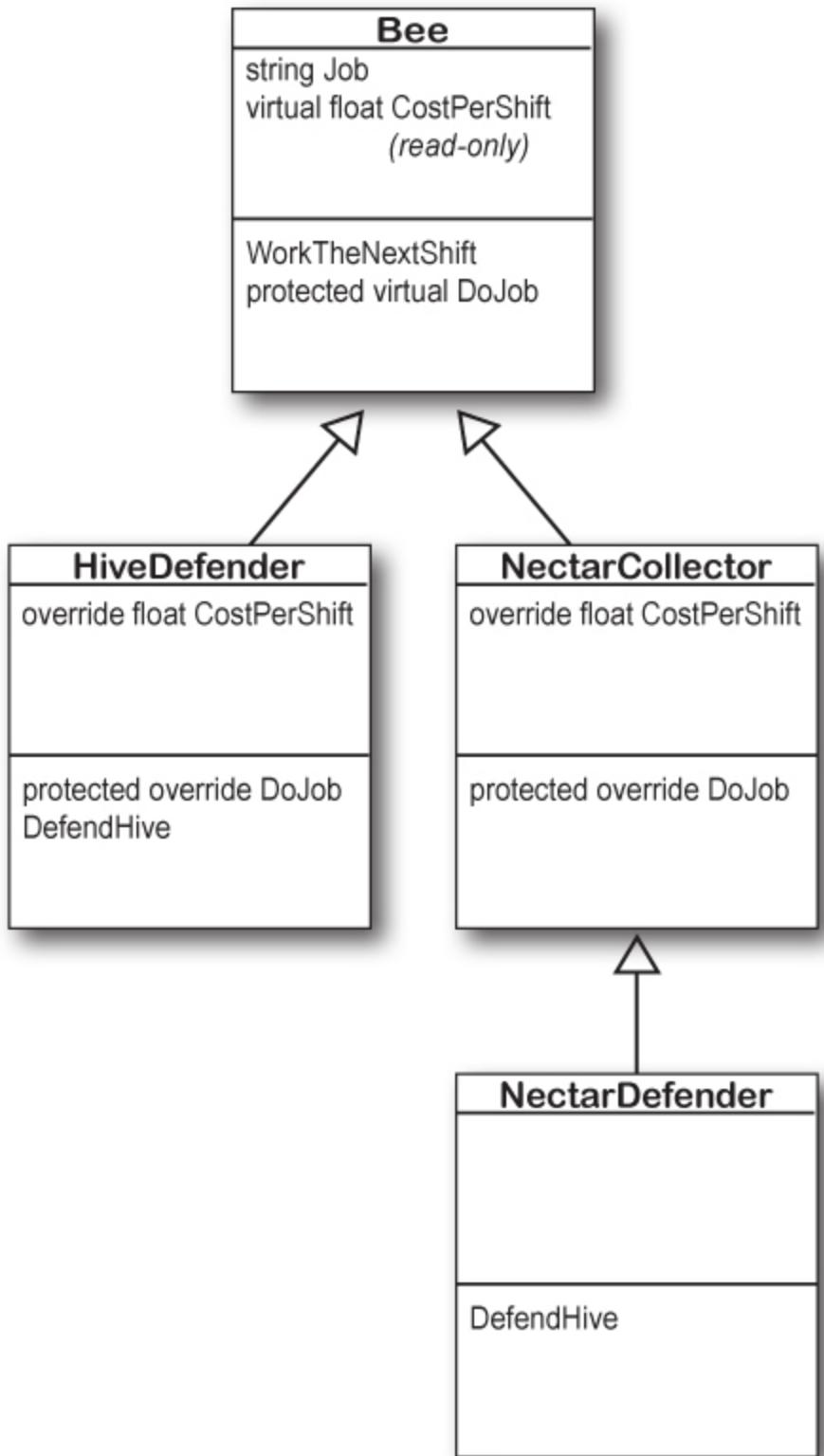
from. That's where **interfaces** come in—they let you work with any class that can do the job. But with **great power comes great responsibility**, and any class that implements an interface must promise to **fulfill all of its obligations**...or the compiler will break their kneecaps, see?

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.



But the rest of the worker bees are loyal to their queen, and the ones who can defend the hive will take up arms. The Nectar Collectors are expert navigators, and can spot enemies while they're making their rounds collecting nectar from flowers. They may not be the elite warriors that their HiveDefender sisters are, but they've still got stingers, and they'll defend their hive.



So we need a DefendHive method, because enemies can attack at any time

We can add a HiveDefender 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 to any of her 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.

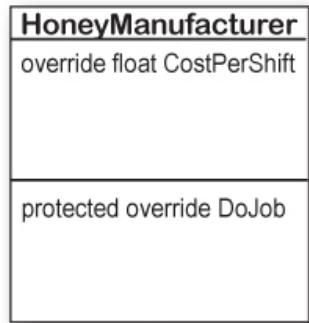
We can use casting to call the DefendHive method...

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 method. 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();
            }
        }
    }
}
```

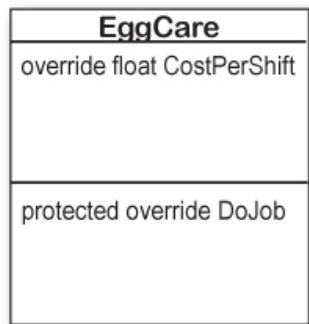


protected override DoJob



DefendHive

↑
The honey manufacturer and egg
care bees what to help defend the
hive, too. Does the queen need to
have an if/else for each subclass?
↓



protected override DoJob

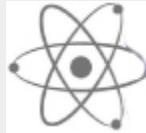


DefendHive

... 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. But 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. But 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?***



BRAIN POWER

Is it possible to modify the bee class model so the HiveUnderAttack method can be as simple as the Queen.DoJob method?

An interface defines methods and properties that a class must implement...

An **interface** contains a set of methods and properties that a class must implement. It works just like an abstract class: you define the interface using 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, but 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 when we talked about abstract classes 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 on interfaces that a class can implement

We just said that you use a colon (:) to make a class implement an interface. But hold on—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 */
    }
}
```

```
    }  
}
```

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 it can also defend the hive. NectarCollector extends Bee, so you **use it from a Bee reference** it acts like a Bee:

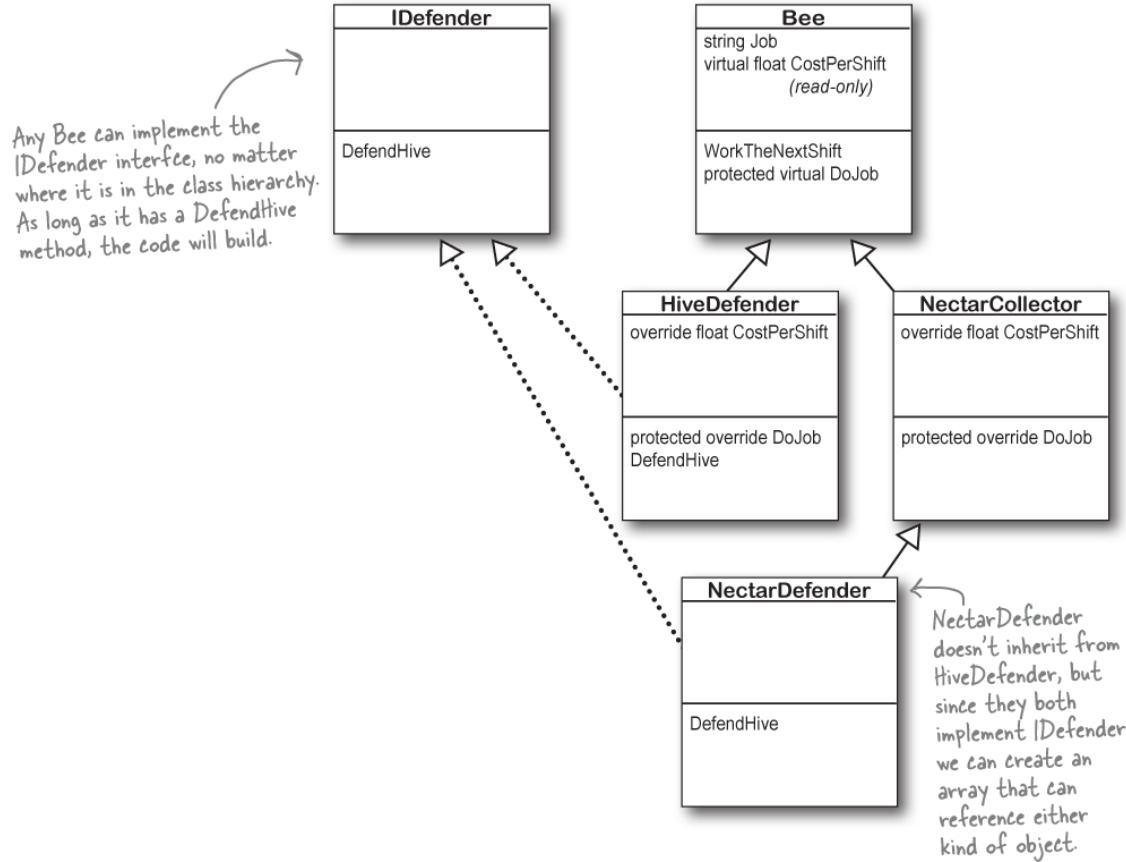
```
Bee worker = new NectarCollector();  
Console.WriteLine(worker.Job);  
worker.WorkTheNextShift();
```

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 as easy to understand and build as possible. Start by thinking about **specific jobs that classes need to do**, because that's what interfaces are all about.



So how does this help the queen? The **IDefender** interface lives entirely outside the Bee class hierarchy. So we can add a **NectarDefender** class that knows how to defend the hive, and **it can still extend NectarCollector**. The Queen can keep an array of all of her defenders:

```

IDefender[] defenders = new IDefender[2];
defenders[0] = new HiveDefender();
defenders[1] = new NectarDefender();
  
```

That makes it easy for her to rally her defenders:

```
private void DefendTheHive() {  
    foreach (IDefender defender in defenders)  
    {  
        defender.Defend();  
    }  
}
```

And since it lives outside of the Bee class model, we can do this ***without modifying any existing code***.



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 them is to start using them. So **create a new Console Application** project—let's get started.

Do this

- 1. Add the Main method.** Here's the code for the method. Here's the TallGuy class, and the code for the Main method that instantiates it using an object initializer and calls its TalkAboutYourself method. 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.** You already know that everything inside an interface has to be public, but don't take our word for it. Add a new `IClown` interface to your project, just like you would add a class: right-click on the project in the Solution Explorer, **select Add >> New Item... 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 type "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 okay 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.

Now **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()'
```

5. 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.

6. 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();
}
```



SHARPEN YOUR PENCIL

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...

1)

```
interface Foo { }
class Bar : Foo { }
```

2)

```
interface Vinn { }
abstract class Vout : Vinn { }
```

3)

```
abstract class Muffie : Whuffie { }
class Fluffie : Muffie { }
interface Whuffie { }
```

4)

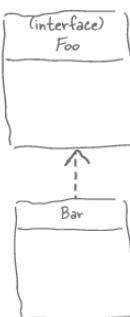
```
class Zoop { }
class Boop : Zoop { }
class Goop : Boop { }
```

You'll need a little
more room for #5.

3)

4)

2)



3)



5)

```
class Gamma : Delta, Epsilon { }
interface Epsilon { }
interface Beta { }
class Alpha : Gamma, Beta { }
class Delta { }
```

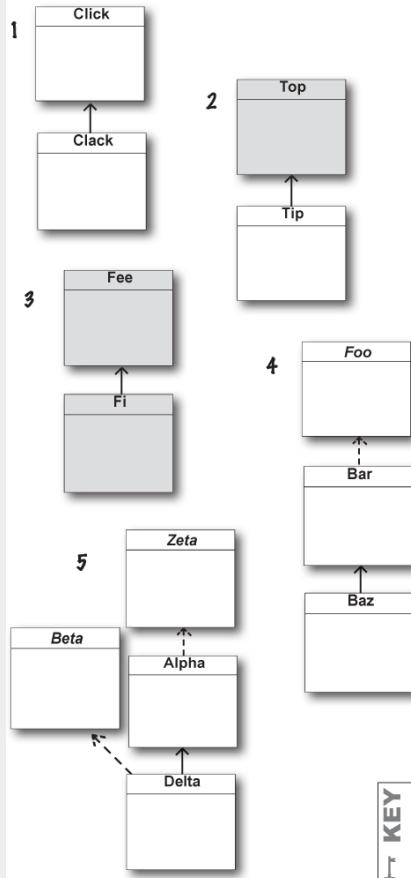
5)



SHARPEN YOUR PENCIL

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!)

If you're given...



What's the declaration?

1) public class Click { }

public class Clack : Click { }

2)

3)

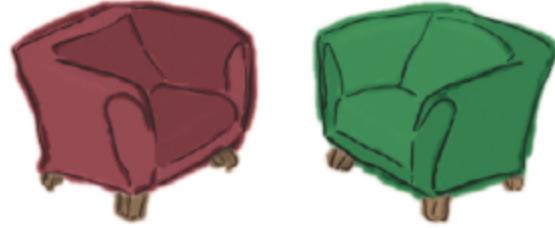
4)

5)

KEY

↑	extends	Clack	class
↑	implements	Clack	interface
↔		Clack	abstract class

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.

Interface:

Nice. This oughta be good.

You can't really think you're more important than me. You don't even use real inheritance —you only get implemented.

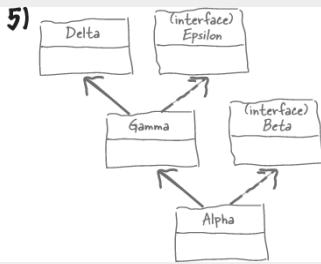
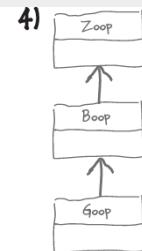
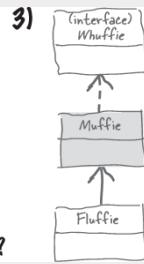
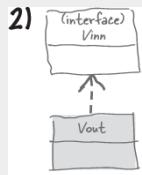
Great, here we go again. Interfaces don't use real inheritance. Interfaces only implement. That's just plain ignorant. Implementation is as good as inheritance. In fact, it's better!

Better? You're nuts. I'm much more flexible than you. Sure, I can't be instantiated—but then, neither can you. But 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!

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. And 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.

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.

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 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!

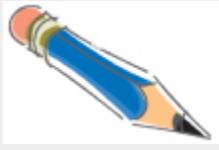
Nine times out of ten, a programmer wants to make sure an object has certain properties and methods, but doesn't really care how they're implemented.

Really? I doubt that—programmers always care what's in their properties and methods.

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!

Yeah, sure, tell a coder they can't code.

Ugh you're ***so frustrating!***



SHARPEN YOUR PENCIL SOLUTION

2) abstract class Top { }
class Tip : Top { }

3) abstract class Fee { }
abstract class Fi : Fee { }

4) interface Foo { }
class Bar : Foo { }
class Baz : Bar { }

5) interface Zeta { }
class Alpha : Zeta { }
interface Beta { }
class Delta : Alpha, Beta { }

What's the declaration?

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. We 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. But 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?

...but 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();
```

Even though this object can do more, when you use an interface reference you only have access to the methods in the interface.

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.



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") { }

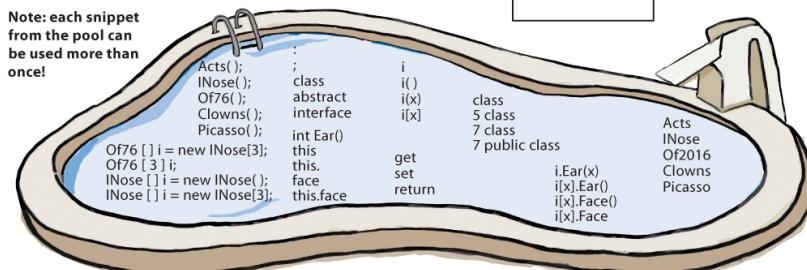
        public override _____ {
            return 5;
        }

        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 +=
                    $"{{_____{x}}} {{_____{x}}}\n";
            }
            Console.WriteLine(result);
            Console.ReadKey();
        }
    }
}

```

Here's the entry point—this is a complete C# program.

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**

```
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;
    }
}
```

```
Acts();
INose();
Of76();
Clowns();
Picasso();
;
class abstract interface
int Ear()
this.
face.
this.face
i()
i(x)
i[x]
class
5 class
7 class
7 public class
i.Ear(x)
i[i].Ear()
i[x].Face()
i[x].Face
→Acts.
-INose-
-Of76-
-Clowns-
-Picasso-
```

```
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();
}
```

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.

Interface references are ordinary object references

You already know all about how objects live on 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.

1. Objects are created as usual.

Let's create some objects. We'll create 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();
```

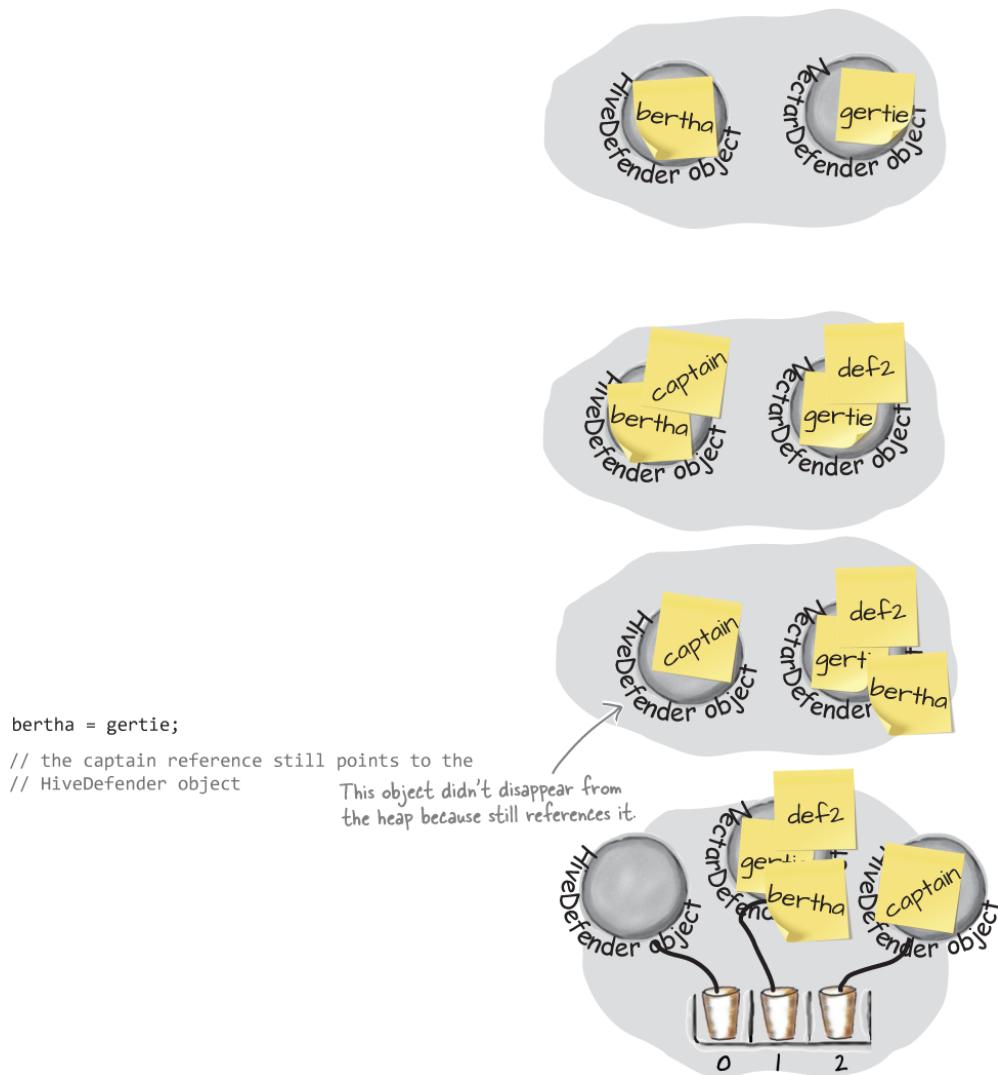
2. Add IDefender references.

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. But 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.



4. Use an interface like any other type.

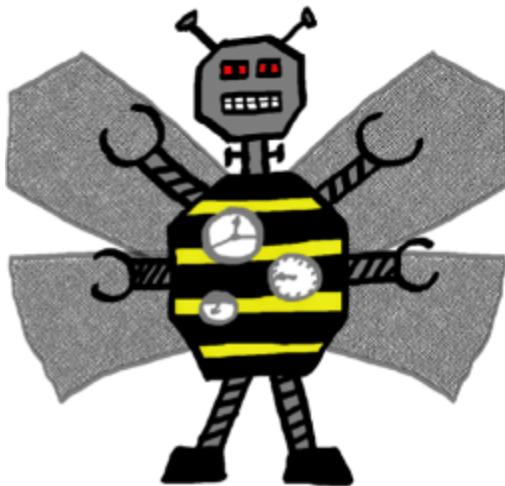
You can create a new object with a new statement and assign it straight to an interface reference variable in a single line of code. And you can **use interfaces to create arrays** that can reference any object that implements the interface.

```

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-suiness 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?

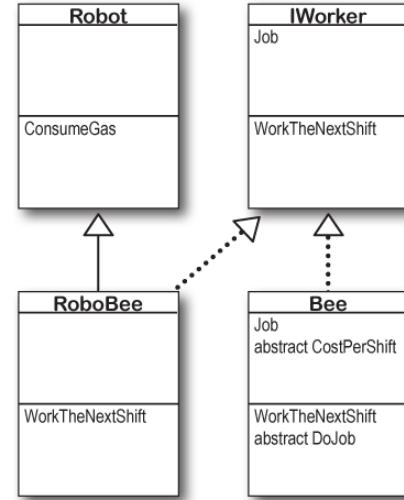
```

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!
    }
}

```



Take a close look at the class diagram to see how this would work.

- We'll start with a basic Robot class—everyone knows all robots run on gasoline, so it has a ConsumeGas method.
- The RoboBee class inherits from Robot and implements IWorker. That means it's a robot, but can do the job of three worker bees. Perfect!
- The RoboBee class implements all of the IWorker interface members. We don't have a choice in this—if RoboBee4000 class doesn't implement everything in the IWorker interface, the code won't compile.

And now you all we'd need to do is modify the Beehive Management System to use the IWorker interface instead of

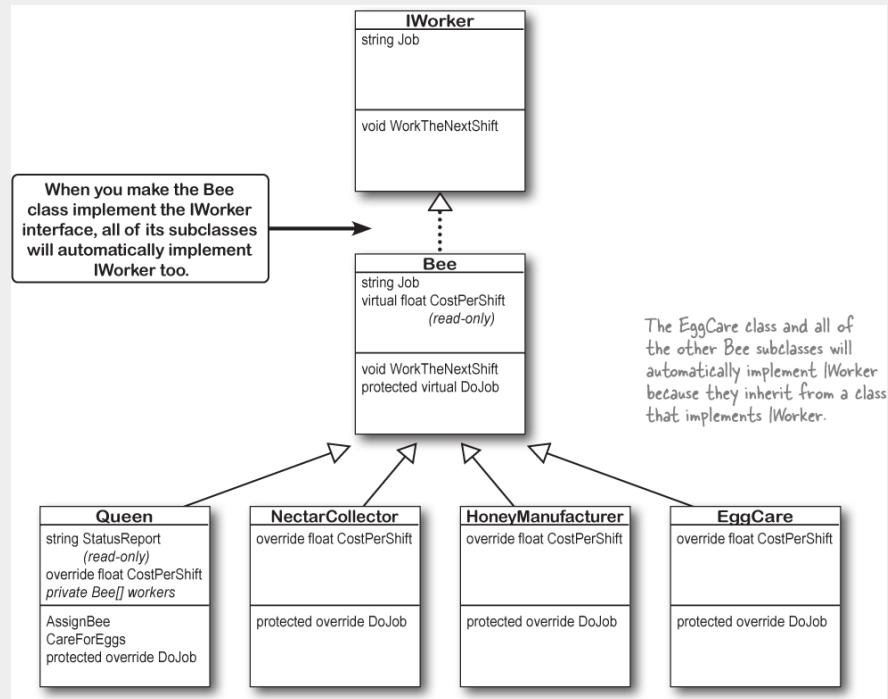
the abstract Bee class any time it needs to reference a worker.



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 referene 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 `Been` and `Queen` classes. This didn't take a lot of code—that's because interfaces are really easy to use.

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 */
}
```

Any class can implement ANY interface as long as it keeps the promise of implementing the interface's methods and properties.

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 */  
}
```

Try modifying `WorkerStatus` to change the `IWorker` variable in the `foreach` loop back to a `Bee` variable, then run your code—it works just fine. Now try changing it to a `NectarCollector` variable. When you run the game, 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). But you could also implement it like this:

```
public Job {  
    get {  
        return "Egg Care";  
    }  
}
```

and the compiler will be perfectly happy with that, too. 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 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. And 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.

The `IWorker`'s `Job` property is a hack

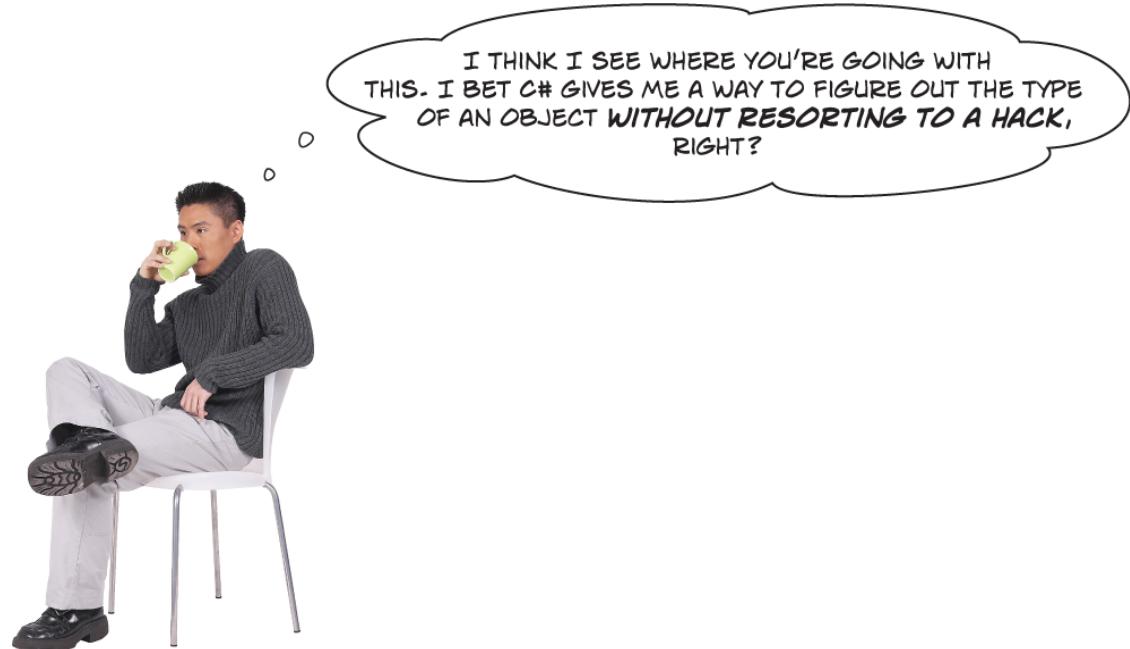
Does something seem a little odd about the fact that every worker has a property called `Job`?

Think about all of the classes you've made so far. You didn't include a property that returns the string "Pigeon" or "Ostrich" or "SwordDamage" or "ArrowDamage"—in fact, this is the only time in the book that you've done anything like the `Job` property.

So why do we think the **Job property is a hack**? Well, think about how much work you've put into **preventing bugs**. You've used access modifiers to prevent your classes from accidentally being misused. You've used abstract classes so you don't accidentally create an instance of a class that should never be instantiated. And this **all happens at compile time**—the C# compiler does all of those checks for you when it builds your code, so they'll never cause exceptions later on. So

it would be a real shame to do all that work, only to have **a simple typo break your code**:

```
class EggCare : Bee {  
    public EggCare(Queen queen) : base("Egg Crae") ← We misspelled "Egg Care" – that's a  
    // Oops! Now we've got a bug in the EggCare class,  
    // even though the rest of the class is the same.  
}
```



That's right! C# gives you tools to work with types.

You don't ever need a property like `Job` to keep track of the type of a class with strings like "`Egg Care`" or "`Nectar Collector`". C# gives you tools that let you check the type of an object.

hack, noun.

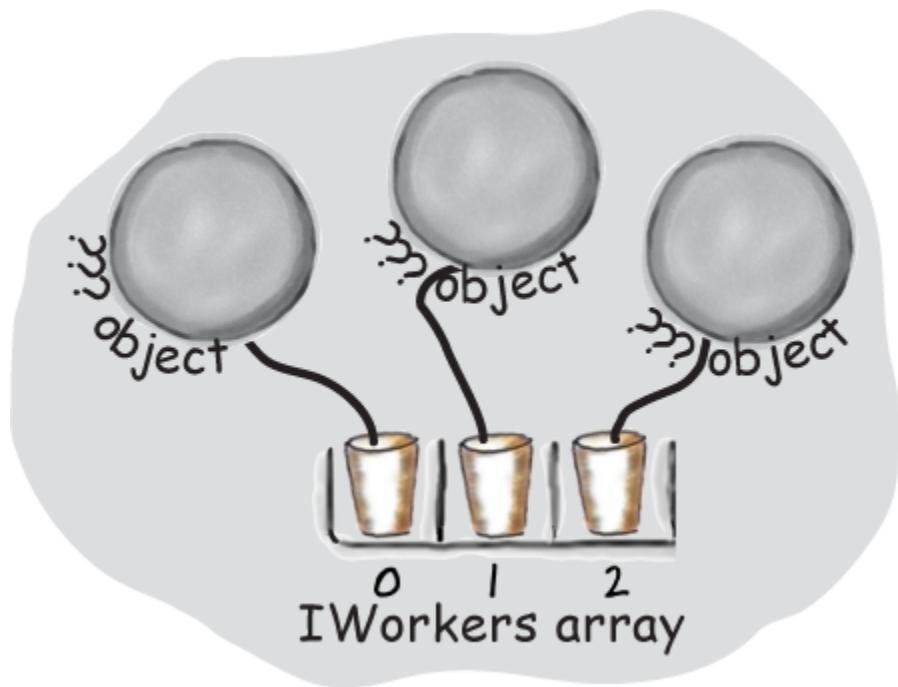
in engineering, an ugly, clumsy, or inelegant solution to a problem that will be difficult to maintain. *Lila spent an extra hour refactoring the **hack** in her code so she wouldn't have to deal with bugs later.* (synonym: kludge [klooj])

Use “is” to check the type of an object

The Queen needs more workers! She wants to find all of the EggCare workers and have them work extra nighttime shifts to make eggs into workers more quickly. But all she has is her workers array, which means that all she can get is an IWorker reference. She could use the hacky Job property:

```
foreach (IWorker worker in workers) {  
    if (worker.Job == "Egg Care") {  
        WorkNightShift((EggCare)worker);  
    }  
  
    void WorkNightShift(EggCare worker) {  
        // Code to work the night shift  
    }  
}
```

But 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 we could get the compiler to 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 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 optionally declares a variable of the type it checked that references the object.

Use the “is” keyword to access methods in a subclass

Do This

Let’s pull together everything we’ve talked about so far into a new project. Here’s what we’ll do:

- We’re going to implement a class model. At the top of the hierarchy is the abstract class `Animal`.

- The Hippo subclass overrides the abstract MakeNoise method, and adds its own Swim method that has nothing to do with the Animal class.
- Animal is also extended by an abstract class called Canine, which has its own abstract property AlphaInPack. The Wolf class extends Canine, and adds its own HuntInPack method.

Create a Console App and **add these Animal, Hippo, Canine, and Wolf classes** to it:

```
abstract class Animal
{
    public abstract void MakeNoise();
}

class Hippo : Animal
{
    public override void MakeNoise()
    {
        Console.WriteLine("Grunt.");
    }

    public void Swim()
    {
        Console.WriteLine("Splash! I'm going for a
swim!");
    }
}

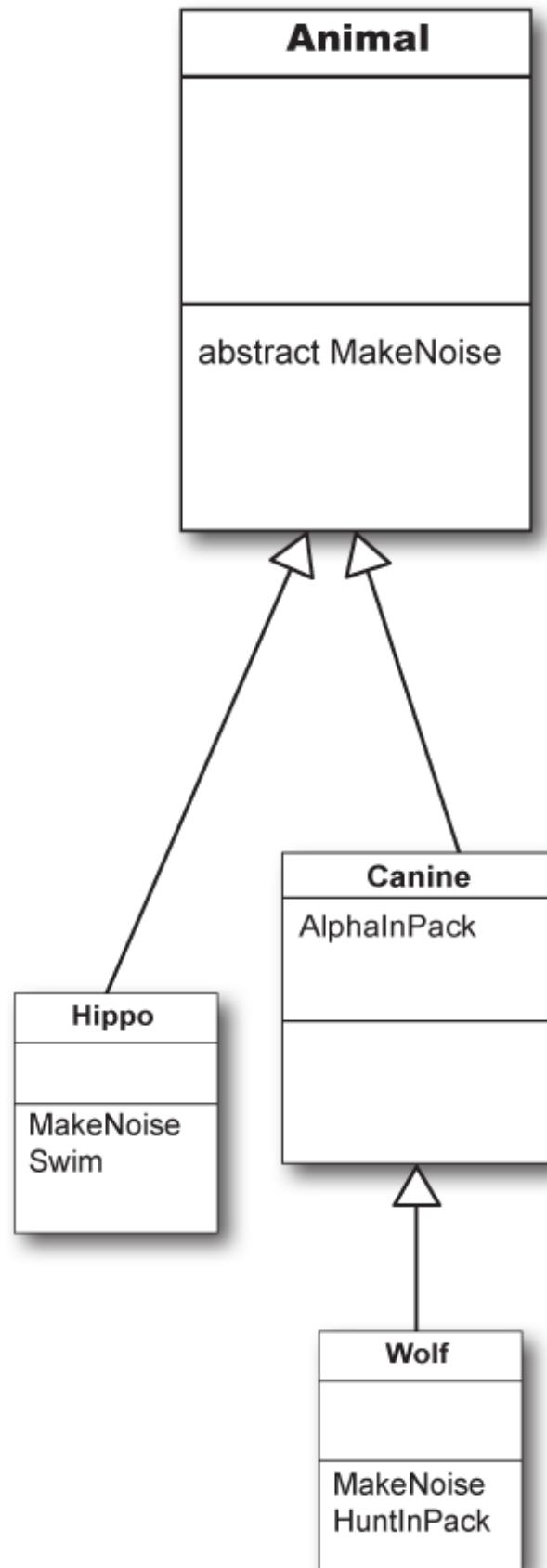
abstract class Canine : Animal
{
    public bool AlphaInPack { get; protected set; } =
```

```
        false;
    }

    class Wolf : Canine
    {
        public Wolf(bool alphaInPack)
        {
            AlphaInPack = alphaInPack;
        }

        public override void MakeNoise()
        {
            if (AlphaInPack) Console.WriteLine("I'm the alpha!");
        );
            Console.WriteLine("Aroooooo!");
        }

        public void HuntInPack()
        {
            if (AlphaInPack) Console.WriteLine("Let's go
hunting. Follow me!");
            else Console.WriteLine("I'm following the
alpha.");
        }
    }
}
```



In Chapter 5 we learned that we could use different references to call different methods on the same object. When you didn't use the `override` and `virtual` keywords, if your reference variable had the type `Locksmith` it called `Locksmith.ReturnContents`, but if it was a `JewelThief` type it called `JewelThief.ReturnContents`. We're doing something similar here.

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`, we want to call its `Hippo.Swim` method.
- If it's a `Wolf`, we want to call its `Wolf.HuntInPack` method.

The problem is that if 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 an easy 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 Swim method.

```
Hippo hippo = (Hippo)animal;  
hippo.Swim(); // <-- It's the same object, but now  
you can call its 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 uses the `is` keyword to check if the animal reference is a Hippo or Wolf, and then safely cast 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?

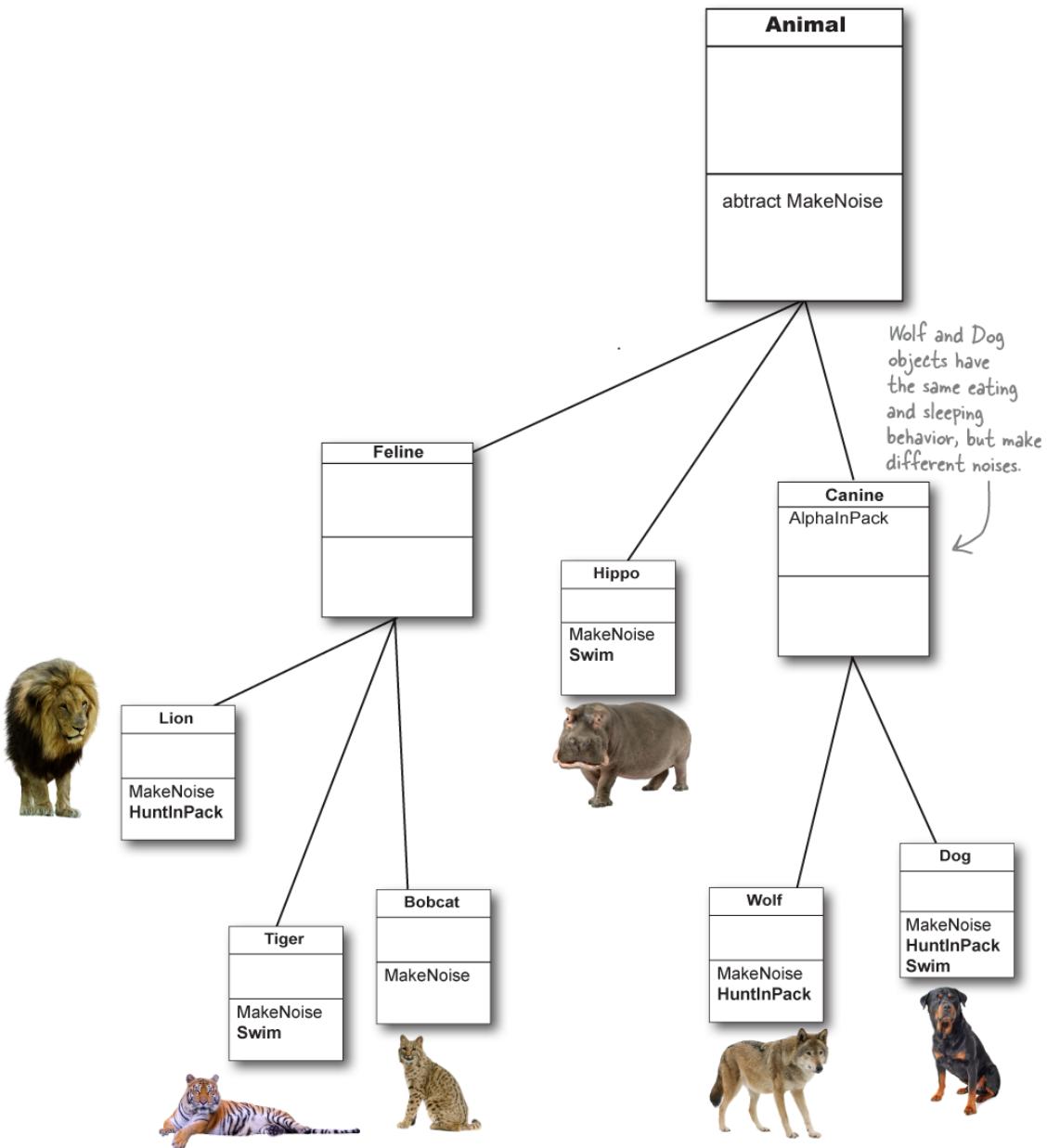
Did you know that lions are pack hunters? Or that tigers can swim? And what about dogs, which hunt in packs AND swim? If we want to add the `Swim` and `HuntInPack` methods to all of the animals in our zoo simulator model, the `foreach` loop is just going to get longer and longer.

But it's not just about saving code. 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 you work with classes that do the same job

Class that swim have a `Swim` method, and classes that hunt in packs have a `HuntInPack` method. Okay, that's a good start. But now we want to write code that works with object 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:

Add this!

```
interface ISwimmer {
    void Swim();
}

interface IPackHunter {
    void HuntInPack();
}
```

Next **make the Hippo and Wolf classes implement the interfaces** by adding an interface to the end of the 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)

```

foreach (Animal animal in animals)
{
    animal.MakeNoise();
    if (animal is ISwimmer swimmer) ←
    {
        swimmer.Swim();
    }
    if (animal is IPackHunter hunter)
    {
        hunter.HuntInPack();
    }
    Console.WriteLine();
}

```

What would your code look like if you had twenty different Animal subclasses that swim? You'd need twenty different `(if animal is ...)` statements that cast `animal` to each individual subclass to call the `Swim` method. By using an `ISwimmer` we only have to check it once.

C# helps you safely navigate your class hierarchy

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 IllegalCastException 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 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** because it lets you check that the type matches, only does the cast if it true.

```
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# also 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. But 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 it 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];           1.  
(bees[i] is IDefender)  
bees[0] = new HiveDefender();  
bees[1] = new NectarCollector();  
.....  
bees[2] = bees[0] as IWorker;  
bees[3] = bees[1] as NectarCollector;        2.  
(bees[i] is IWorker)  
//bees[4] = IDefender;  
.....  
bees[5] = bees[0];                         3.  
(bees[i] is Bee)  
bees[6] = bees[0] as IDefender;  
//bees[7] = new IWorker();  
.....
```

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 on the diagram, the more abstract it is; the lower the class is on 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.

When you convert a reference to a different type, you're **moving up or down the class hierarchy**. So if you start with this:

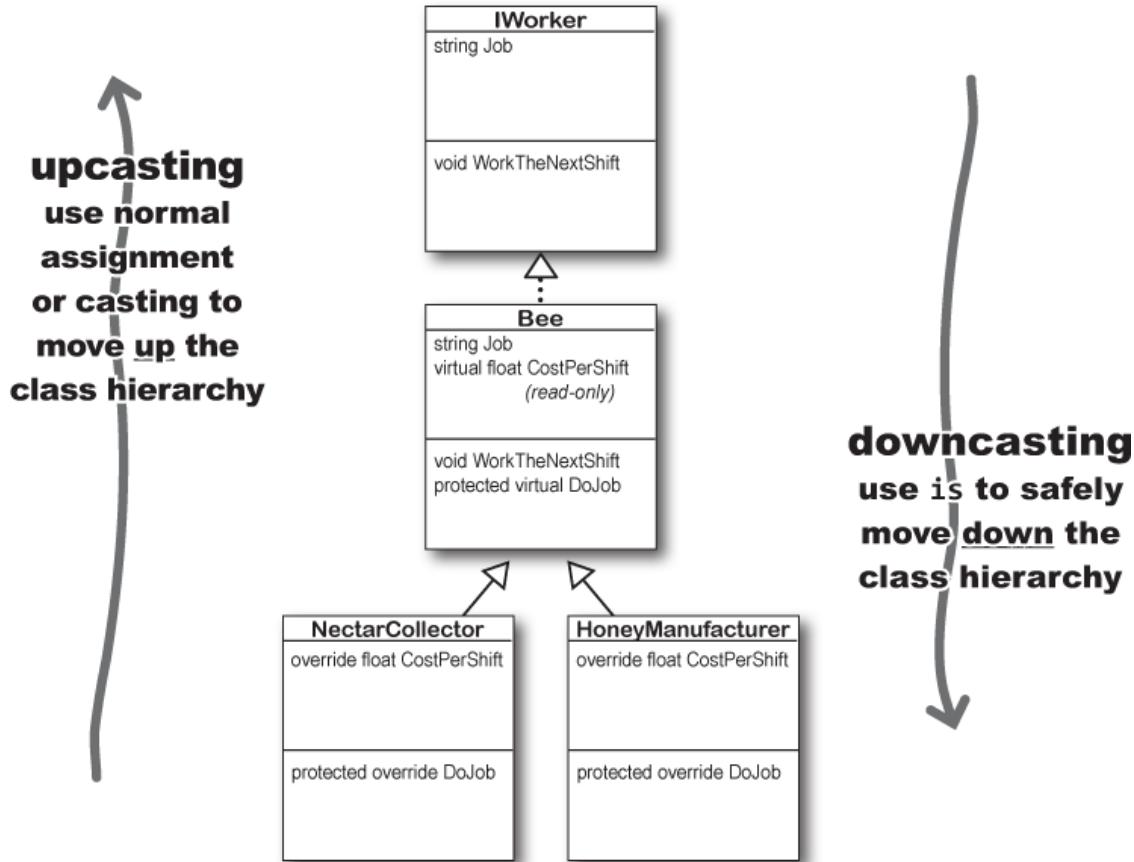
```
NectarCollector ida = new NectarCollector();
```

You can use the same normal assignment with the = operator (for superclasses) or casting (for interfaces) to **move 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 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 */ }
```



A CoffeeMaker is also an Appliance

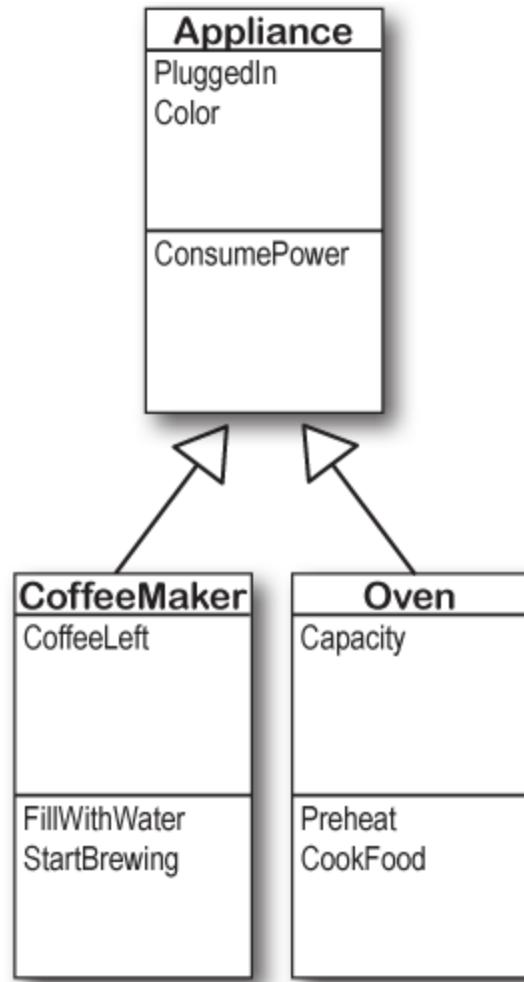
If you're trying to figure out how to cut down your energy bill each month, you don't really care what each of your appliances does—you only really care that they consume power. So if you were writing a program to monitor your electricity consumption, you'd probably just write an **Appliance** class. But if you need 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 want to use that method to monitor the power consumption for a coffee maker, you can create an instance of CoffeeMaker and pass its reference directly to the method:



```
CoffeeMaker mrCoffee = new CoffeeMaker();
MonitorPower(misterCoffee);
```

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] as IDefender;
bees[5] = bees[0];
bees[6] = bees[0] as IDefender;
//bees[7] = new IWorker();
```

1. `(bees[i] is IDefender)`
.....
`O, 2, and b`

2. `(bees[i] is IWorker)`
.....
`O, 1, 2, and b`

3. `(bees[i] is Bee)`
.....
`O, 1, 2, and b`

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 that you get 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 coffee maker like an appliance, you can't tell it to make coffee or fill it with water. But 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();
```

You don't need to add this code to an app—just read through the code and start to get a sense of how upcasting and downcasting work. You'll get lots of practice with them later in the book.

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. But 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 coffee makers 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:

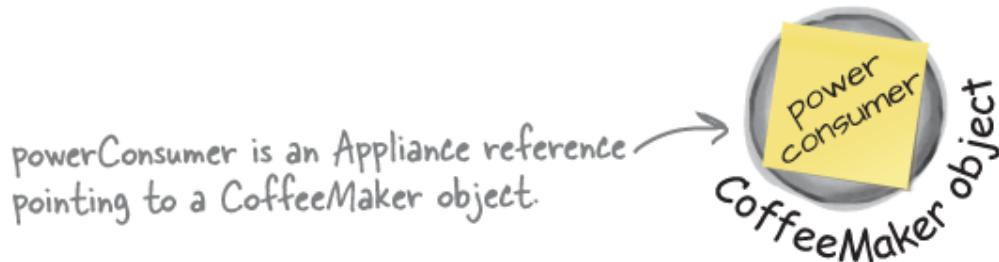
```
Appliance powerConsumer = new CoffeeMaker();  
powerConsumer.ConsumePower();  
  
But as soon as you try to use it like a CoffeeMaker:  
powerConsumer.StartBrewing();
```

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?)

because once you upcast from a subclass to a base class, then you can only access the methods and properties that **match the reference** that you're using to access the object.



Downcasting turns your Appliance back into a CoffeeMaker

Upcasting is a great tool, because it lets you use a coffee maker 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. And 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. But what if we want to turn the Appliance back into a CoffeeMaker?

We can't just use our Appliance reference to call CoffeeMaker method:

```
powerConsumer.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.

Here's our Appliance reference that points to a CoffeeMaker object. You can only use it to access members of the Appliance class.



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 `as` to downcast it. And that lets you use the `CoffeeMaker` class's methods and properties. And since `CoffeeMaker` inherits from `Appliance`, it still has its `Appliance` methods and properties.

```
if (powerConsumer is CoffeeMaker  
javaJoe) {  
    javaJoe.StartBrewing();  
}
```

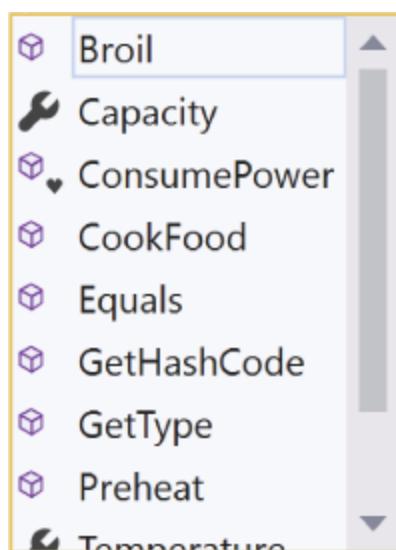
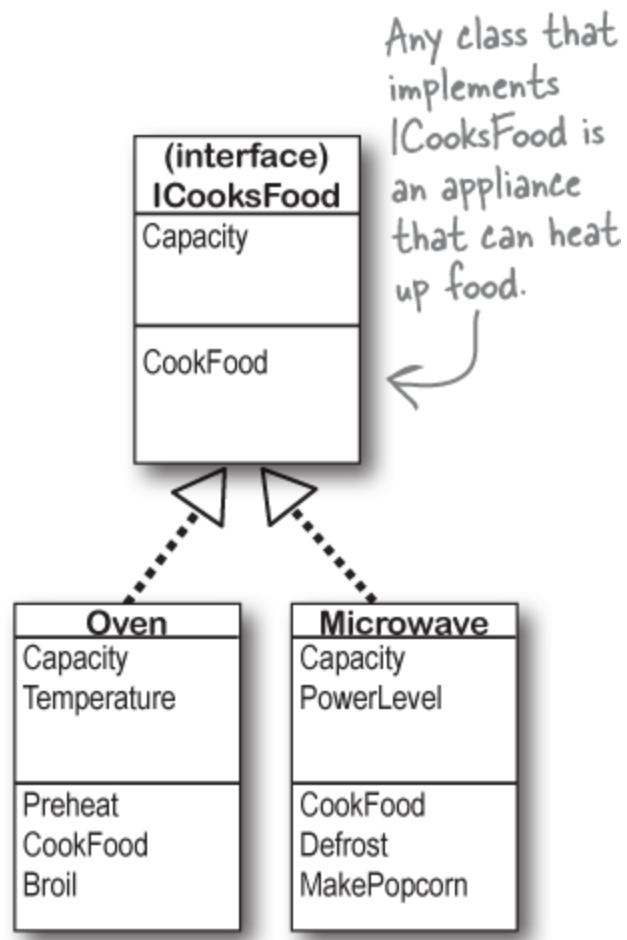


The `javaJoe` reference points to the same `CoffeeMaker` object as `powerConsumer`. But it's a `CoffeeMaker` reference, so it can call the `StartBrewing` method.

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. And we'll add a Microwave class—both Microwave and Oven implement the ICooksFood interface. Now there are three different ways that you can access an Oven object. 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:

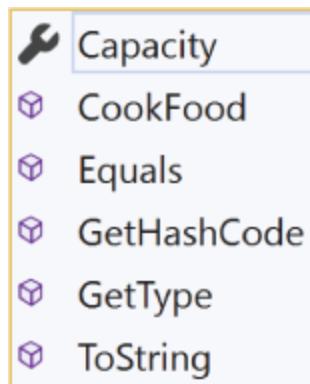
```
Oven misterToasty = new Oven();  
misterToasty.
```



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. But it's the most specific type, so you can only point it at `Oven` objects.

If you want to access `ICooksFood` interface members, convert it to an `ICooksFood`:

```
if (misterToasty is ICooksFood cooker) {  
    cooker.
```

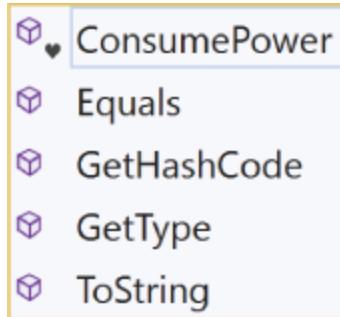


`cooker` is an `ICooksFood` reference pointing to that same `Oven` object. It can only access `ICooksFood` members, but it can also point to a `Microwave` object.

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.
```

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—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 on 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. And 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 a class that implements it. And 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. But 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, then 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. Now you could 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

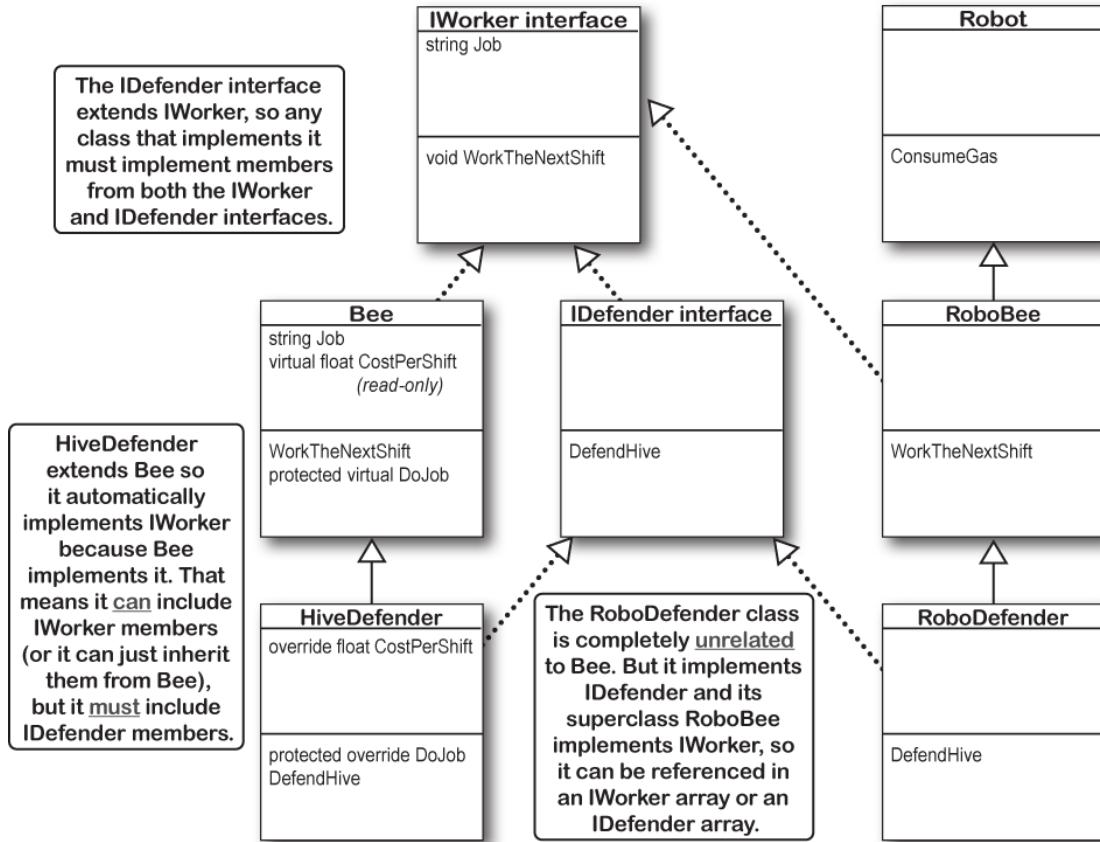
When one class inherits from another, it gets all of the methods and properties from the base class. **Interface inheritance** is simpler. Since there's no actual method body in any interface, you don't have to worry about calling base constructors or methods. The inherited interfaces **accumulate all 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 include every property and method in that interface. And 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 must implement all of the IDefender members, but must also implement 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 application with classes that implement the IClown interface. Can you figure out how to get the code at the bottom to build?

1. Start with the IClown interface you created earlier:

```
interface IClown {  
    string FunnyThingIHave { get; }  
    void Honk();  
}
```

2. 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.

3. 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.

4. Here’s 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();

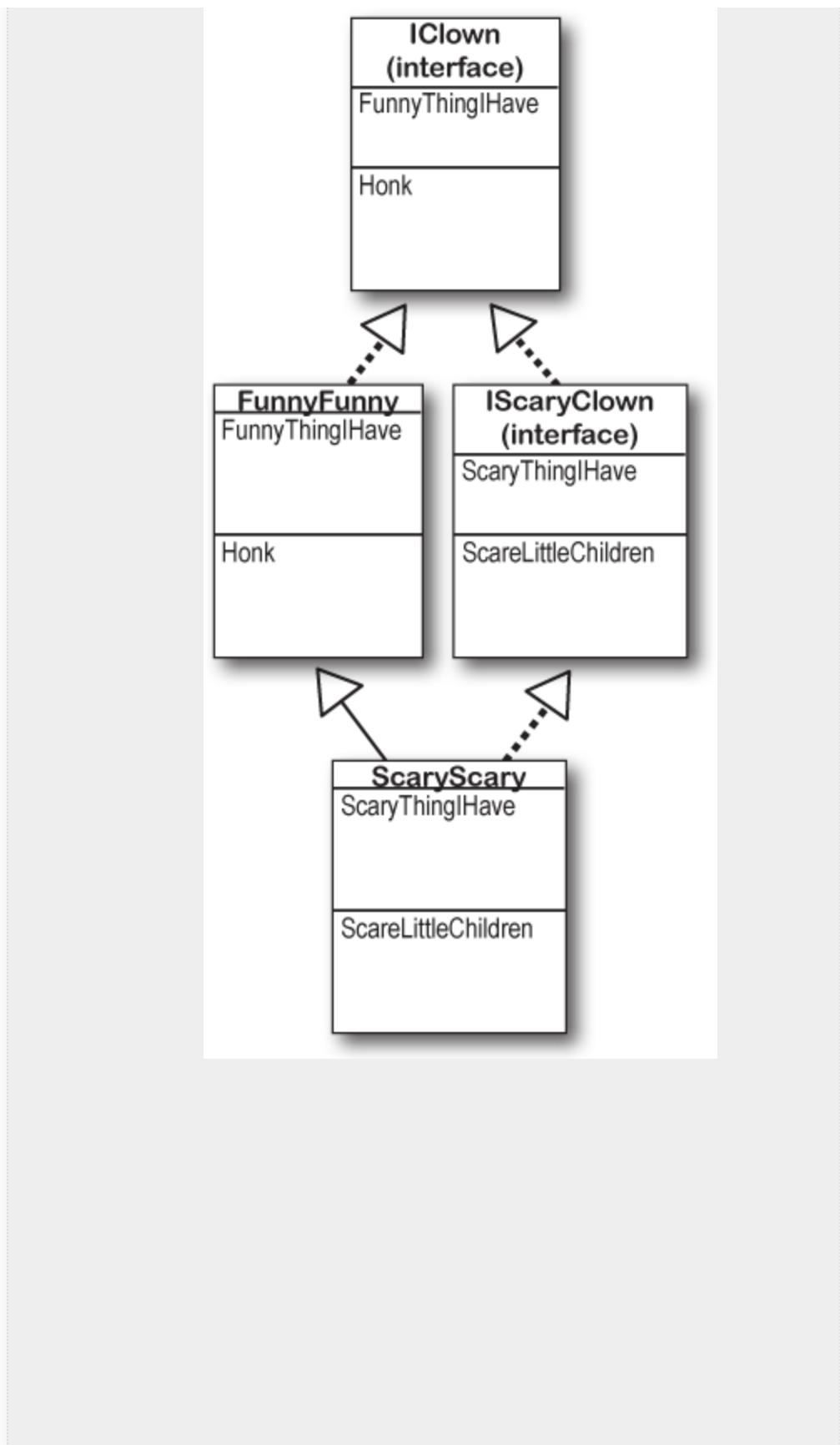
}
```

Before you run the code, **write down the output** that the Main method it will print to the console (once you fix it):

.....

.....

Then run the code and see if you got the answer right.





EXERCISE SOLUTION

Create a new Console application 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}.");
    }
}
```

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.

FunnyFunny.funnyThingIHave is a private field, so **ScaryScary** can't access it —it needs to use the **base** keyword to call the **FunnyFunny** constructor.

```
class ScaryScary : FunnyFunny, IScaryClown
{
    private int scaryThingCount;

    public ScaryScary(string funnyThing, int
scaryThingCount) : base(funnyThing)
    {
        this.scaryThingCount = scaryThingCount;
    }

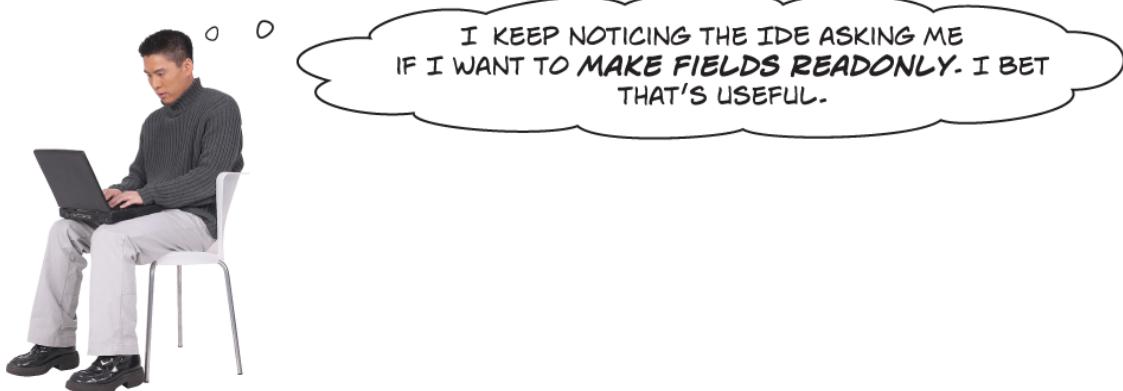
    public string ScaryThingIHave { get { return $""
{scaryThingCount} spiders"; } }

    public void ScareLittleChildren()
    {
        Console.WriteLine($"Boo! Gotcha! Look at my
{ScaryThingIHave}!");
    }
}
```

To fix the Main method, replace lines 3 and 4 of the method with these lines that use the `is` operator:

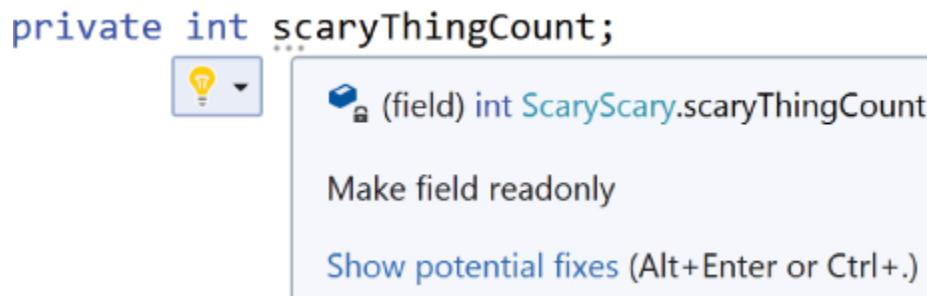
```
if (fingersTheClown is IScaryClown
iScaryClownReference)
{
    iScaryClownReference.ScareLittleChildren();
}
```

You can set a FunnyFunny reference equal to a ScaryScary object because ScaryScary inherits from FunnyFunny. But 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.



Absolutely! Making fields readonly 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:



Press **Ctrl-period** 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 in 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. But 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 does it to get the job done.

Here's an example: you can have a Truck class and a Sailboat class that implements 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 that 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. But 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.

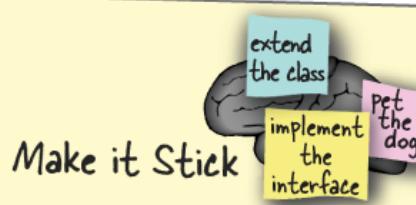
Interface references only know about the methods and properties that are defined in the interface.



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 diamond problem 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, but 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, and optionally declares a variable of the type it checked that references the object.
- **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 **readonly keyword** it can only be set in the field initializer or constructor.



Make it Stick

To remember how interfaces work is take note of the language: 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)

Look up “implement” in the dictionary—one definition is putting a decision, plan, or agreement into effect. →



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. But it can call methods implemented by the object, as long as they're part of the interface.



WATCH IT!

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 old versions of Visual Studio.

Interfaces static members work exactly like in classes

Everybody loves it when way too many clowns pack themselves into a tiny clown car! So let's update the IClown class 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 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:

IClown (interface)
FunnyThingIHuve static CarCapacity protected static Random
Honk static ClownCarDescription

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();

    protected static Random random = new Random();

    private static readonly 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";
    }
}
```

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).

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
}
```

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.

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.

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. But the virtual method is **attached to the interface**. And like any other interface implementation, it's not inherited. And 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 multi-line strings that include line breaks. And 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 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();
    }
}
```

↗ Add this!

↑
We used a verbatim literal here. We could have used a normal string literal instead and added \n's for the line breaks. But this way is a lot easier to read

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 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();
}
```



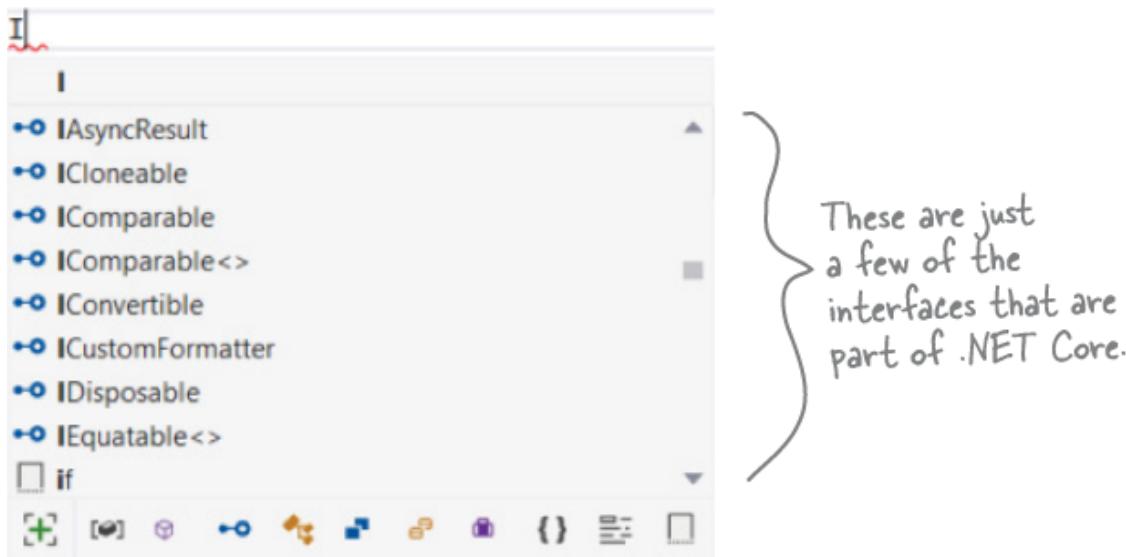
C# developers use interfaces all the time, especially when we use libraries, frameworks, and APIs.

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. But 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:

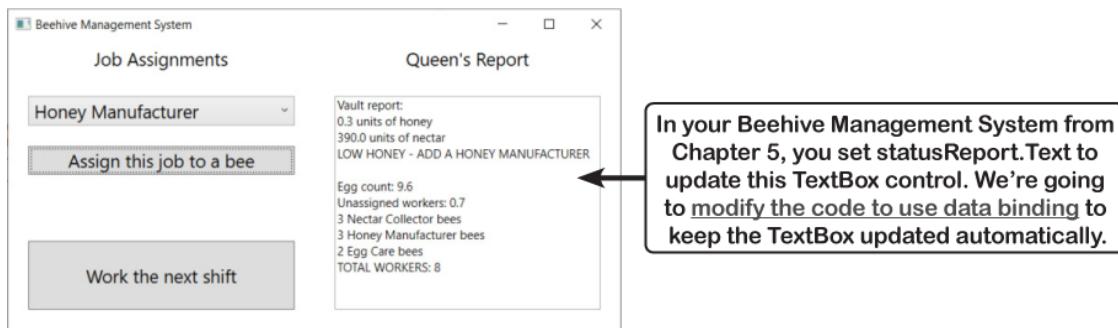
- You've used **frameworks** like .NET Core and WPF.
- You've used **APIs** like the Unity scripting API.
- The frameworks and APIs contain **class libraries** that you access with `using` directives at the top of your code.

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.



Data binding updates WPF controls automatically

Here's a great example of a real-world use of an interface: you can use one to take advantage of a really useful feature in WPF: **data binding**, which lets you set up your controls so their properties are automatically set and kept up to date by your objects.



Here's what we'll do to modify your Beehive Management System code to use data binding:

- 1. Modify the Queen class to implement the `INotifyPropertyChanged` interface.**

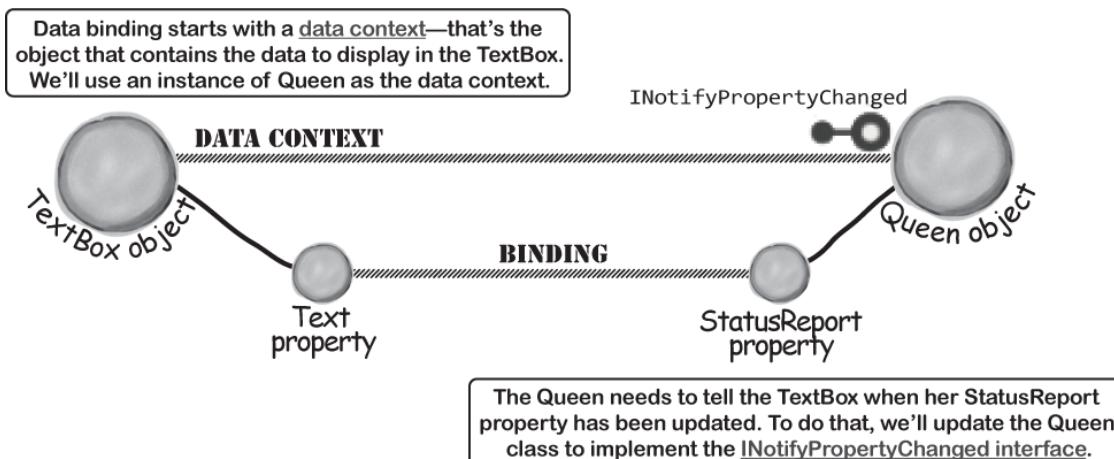
This interface lets the Queen announce that the status report has been updated.

- 2. Modify the XAML to create an instance of Queen.**

We'll bind the `TextBox.Text` property to her `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.



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.

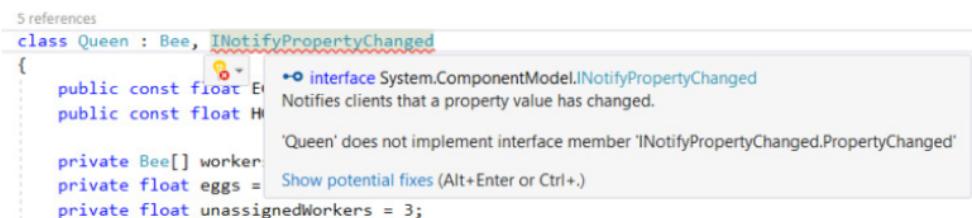
Do this!

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. It the **event keyword**, which you haven't seen yet:

```
public event  
    PropertyChangedEventHandler  
    PropertyChanged;
```

Guess what? You've used events before! The DispatchTimer 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 Click its 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{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 add a method that uses the ?. operator to invoke the event. That's all you need to know about events for now—you'll learn all about them in Chapter 13.

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>
```

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.

Next, **modify the <Grid> tag** to add a DataContext attribute:

```
<Grid DataContext="{StaticResource  
queen}">
```

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: `Resources["queen"]` as Queen – this code uses a **dictionary** called Resources. (*This is a sneak peek at dictionaries! You'll learn about them in the next chapter.*)

```
public MainWindow()  
{  
    InitializeComponent();  
    queen = Resources["queen"] as Queen;  
    //statusReport.Text = queen.StatusReport; ← Now that the WPF app uses data  
    //timer.Tick += Timer_Tick; binding we don't need to use the  
    timer.Interval = TimeSpan.FromSeconds(1.5); report TextBox, so go ahead and  
    timer.Start(); comment out or delete this line.  
}
```

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. But can you go over it again, just in case I missed something?

A: Absolutely. The Beehive Management System app you built in [Chapter 5](#) 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 updated 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. And 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 your own event handlers to your app Button click events.

Q: I noticed that when I open the window in the designer, 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. And 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 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.



WATCH IT!

Data binding only works with properties.

You can only bind an attribute of a control to a public property. If you try to bind to a public field, 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 mockingbird in place of an animal or 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 all the time—we just didn't use that word to describe it. So while you're writing code over the next few chapters, be on the lookout for many different ways that you use polymorphism.

Here's a list of four typical ways that you'll use polymorphism. We gave you an example of each of them (you won't see these particular lines in the exercise, though). As soon as write

similar code in an exercise, 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.

You're using polymorphism when you take an instance of one class and use it in a statement or a method that expects a different type, like a parent class or an interface that the class implements.



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.

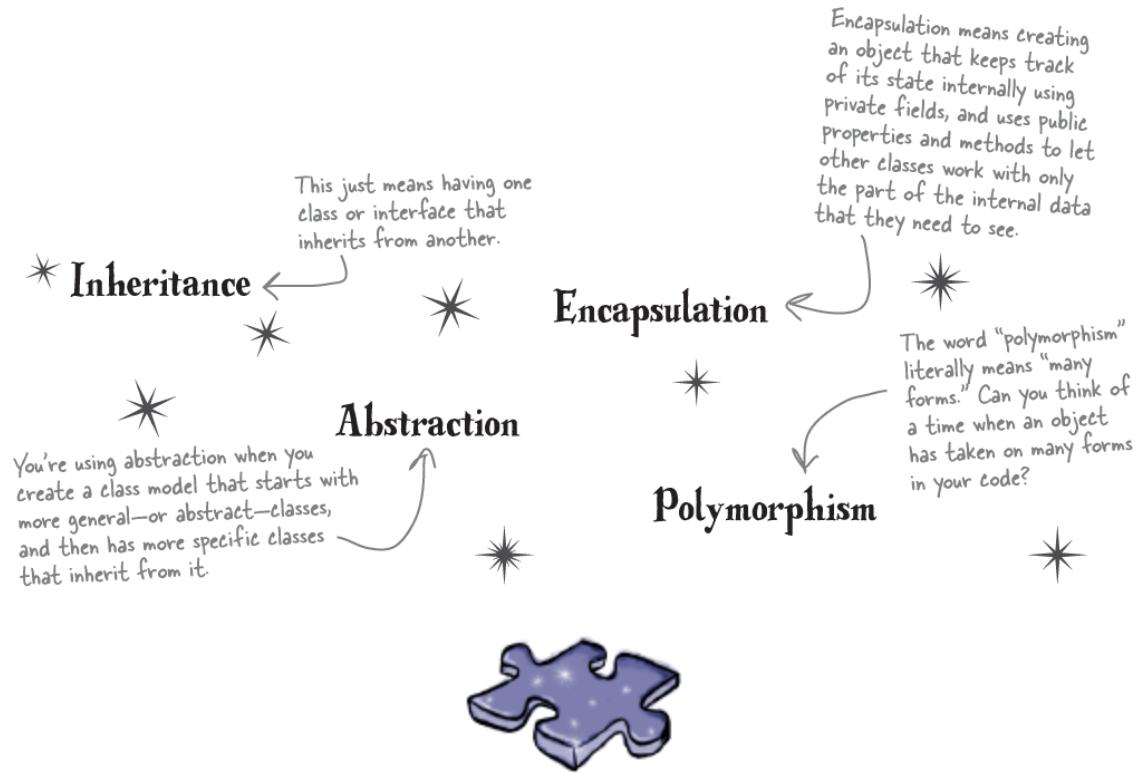
You're an object-oriented programmer.

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-OOP 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#.

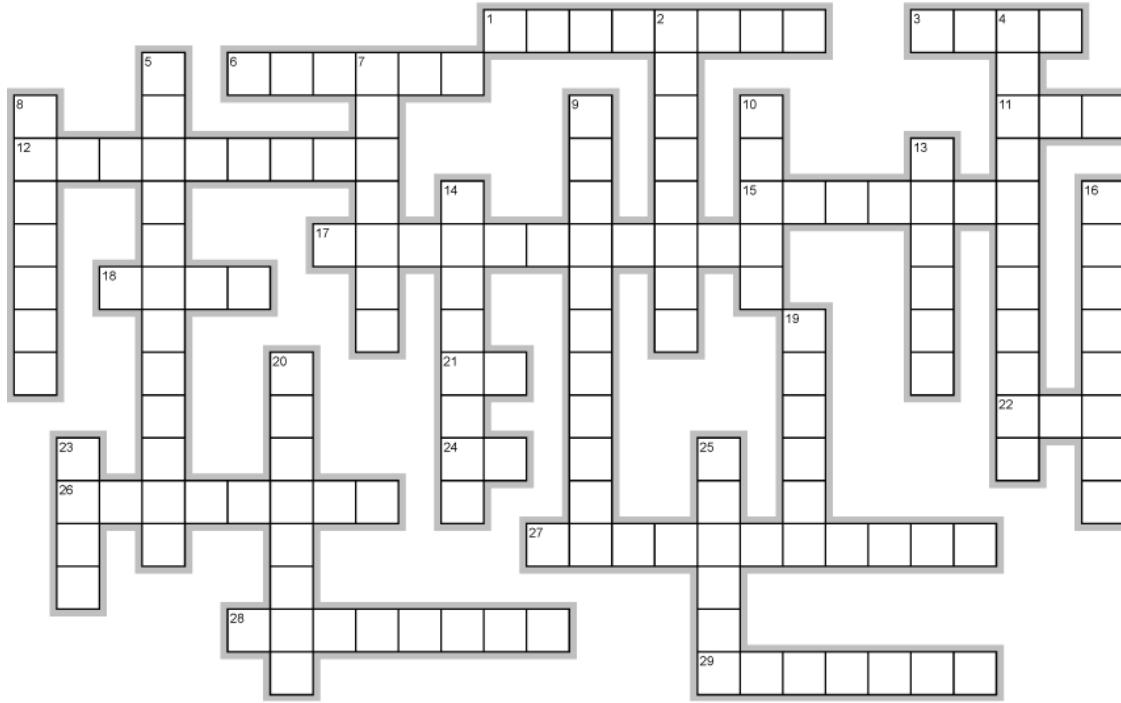
The four core principles of object-oriented programming

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.

You'll recognize the first three principles just from their names: **inheritance**, **abstraction**, and **encapsulation**. The last one's called **polymorphism**. It sounds a little odd, but it turns out that you already know all about it too.



OOPcross



Across

1. The keyword you use in a subclass if you're extending a member in its superclass
3. A really inelegant solution to a problem
6. What the colon (:) operator does with a class
11. The keyword you use when you're hiding a method in a superclass
12. What the colon (:) operator does with an interface
15. The kind of implementation that gives a body to an interface method

17. A subclass gets members from its superclass because of this OOP principle

18. What an abstract method doesn't have

21. A safe way to cast that returns null instead of throwing an InvalidCastException

22. What you're doing when you create programs that combine data and code together into classes and objects

24. The keyword you use to safely downcast

26. Every method in an interface is automatically _____ by default

27. What you can't do with an abstract class

28. The opposite of abstract

29. What your code won't do if you fail to implement all of the members of an interface in a concrete class

Down

2. What to mark a field whose value doesn't change after the class is instantiated

4. If a superclass has this, the subclass must call it

5. When you pass a subclass to a method that expects its base class, you're using this OOP principle
7. If your class implements an interface that _____ another interface, you need to implement all of its members too
8. You can only extend members that are _____ or abstract
9. The OOP principle that sometimes has you move common methods from specific classes to a more general class
10. A set of classes that work together to represent things in the real world
13. What an interface member is by default if you don't add an access modifier
14. Use @ at the beginning of a _____ string literal
16. C# doesn't allow _____ inheritance
19. When you use the = operator to assign a reference to a variable with its superclass as the type
20. Modifying the structure of code without changing its behavior
23. Use this keyword in a subclass to access its superclass
25. What any private field in an interface must be

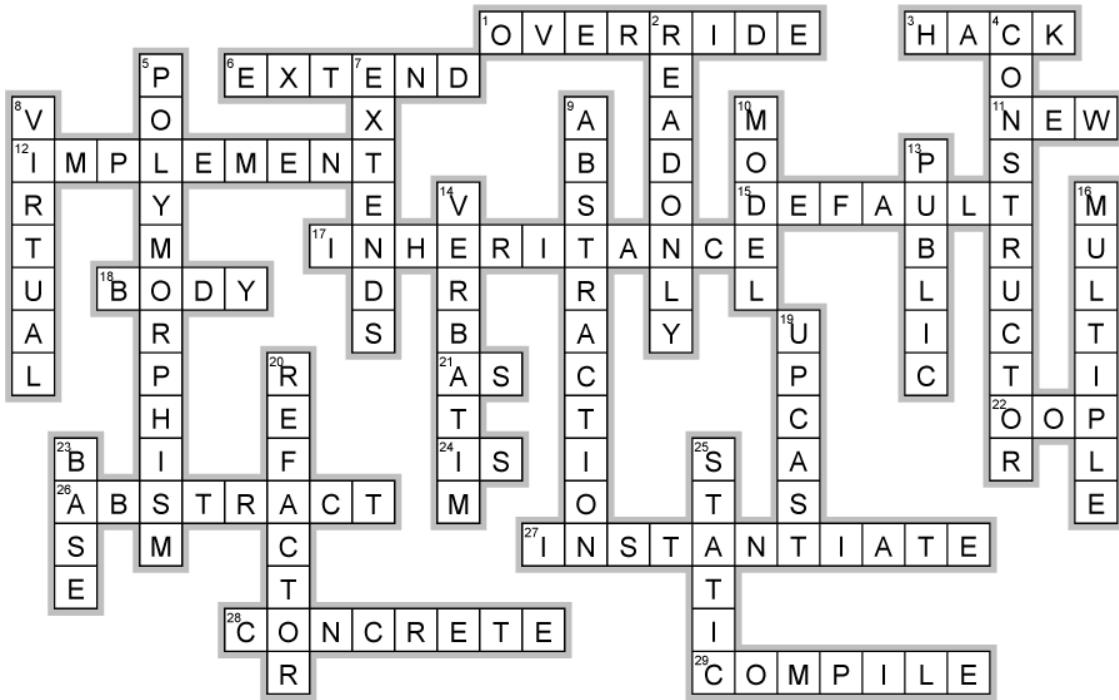


BULLET POINTS

- You can **add code to your interfaces** by including static methods and default implementations.
- **Static members**—methods, properties, and fields—in interfaces work exactly like static members in classes.
- A **default implementation** is a static method that's part of the interface, and can only be called from an interface reference.
- A default implementation **can call members** of the class that implements the interface.
- Interfaces can contain **private members** as long as those members are static.
- Put a @ in front of a string literal to make it a **verbatim string literal**.
- A verbatim string literal can **contain newlines and slashes**, so \n in a verbatim string literal is a slash and an n, not a newline.
- **Data binding** lets you automatically populate data in WPF controls by hooking them up to properties of an object.
- C# developers use **interfaces** all the time! They're not just theoretical, they're really, really useful.
- When you create code that can accept using different related types—like subclasses or classes that implement an interface—you're using **polymorphism**.
- Encapsulation, abstraction, inheritance, and polymorphism are the four core principles of **object oriented programming** (or OOP).



OOPcross Solution



Chapter 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 get to handle 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 **collections** come in. They let you **store, sort, and manage** all the data that your programs need to pore through. That way, you can think about

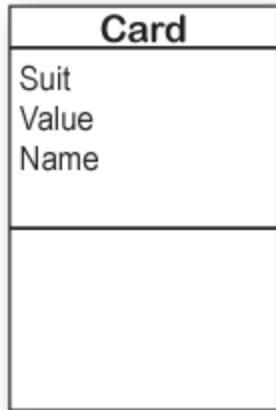
writing programs to work with your data, and let the collections worry about keeping track of it for you.

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. So let's 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;
    }
}
```



↑
This Card class uses
string properties for
suits and values.

That looks pretty good. We can create a card object and use it, no problem.

```

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 dutchessOfWeasels = new Card("Dutchess", "Weasels");
Card fourteenOfBats = new Card("Fourteen", "Bats");
Card nonceOfOxen = new Card("Nonce", "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 that would let us handle bad input and throw an exception. But wouldn't it be better if we could find a way to prevent that bad input in the first place?

If we had a way to only allow certain values in the first place, then we could prevent those bugs and avoid those messy, unnecessary exceptions

What we need is a way to say, “Hey, there are only certain values that are allowed here.” We need to **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.



The rarely-played nonce 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. Each member is 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 to end with a comma, 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**.

- 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.");  
    }  
}
```

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.

- **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);
```

Here's the error you get from the compiler if you try to use a value that's not part of the enum:

 CS0117 'Suits' does not contain a definition for 'Oxen'

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 at the bottom of this page—and you can cast those back to their type.

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. And 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 = 2500000000025  
}
```

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'.

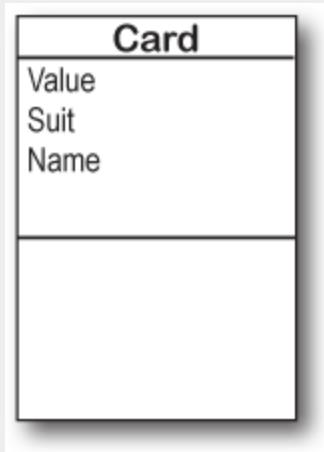


EXERCISE

Use what you've learned about enums to build a class that holds a playing card.

Create a new .NET Core Console App project and add a Card class.

You'll need two public properties: Suit (which will be Spades, Clubs, Diamonds, or Hearts) and Value (Ace, Two, Three...Ten, Jack, Queen, King). And you'll need a read-only property, Name (Ace of Spades, Five of Diamonds).



Add two enums to define the suits and values in their own *.cs files.

Use the familiar Add→'Class feature in the IDE to add each enum. Then **replace class with enum** in each of the files you just added. You can **use the Suits enum that we just showed you**, so you just need to 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.

When you use "Add File" in Visual Studio for Mac, you can choose the "Empty Enumeration" option.

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 and another random number between 1 and 13 as a Values. 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.

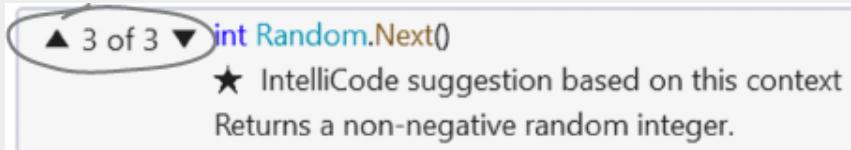
```
Random random = new Random();
int numberBetween0and3 = random.Next(4);
int numberBetween1and13 = random.Next(1, 14);
int anyRandomInteger = random.Next();
```

You did this back in Chapter 3. It tells Random to return a value 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 Random.Next, 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 doing it, make sure you choose the right overloaded Next method! But don’t worry too much now—we’ll talk a lot about overloading later on in the chapter.





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 (`Season`) and plural (`Days`) names for enums:
<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 `Values` and `Suits` properties private because they only ever need to be called from the constructor. That way we'll never accidentally change them.

The `get Name` property 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 want 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 `Card` 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);
    }
}
```

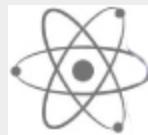
We used two for loops to iterate through all of the possible suit and value combinations.

...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.

Take another look at the `AddWorker` method from the Beehive Management System exercise in Chapter 6. To

insert a new worker in the middle of the array, you had to resize it at exactly the right index.



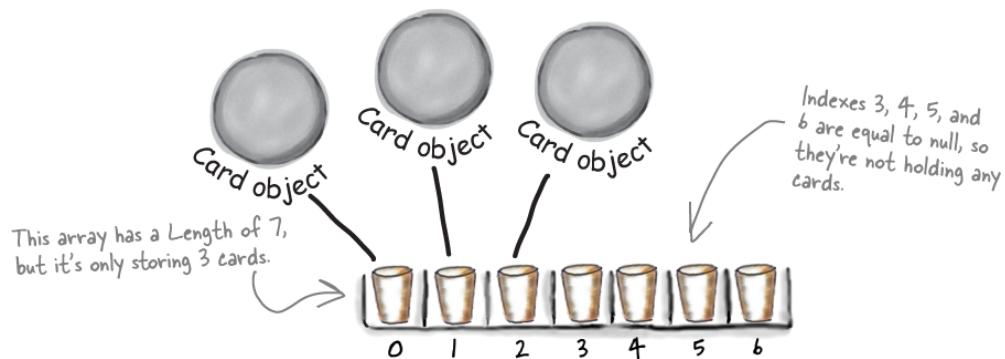
BRAIN POWER

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?

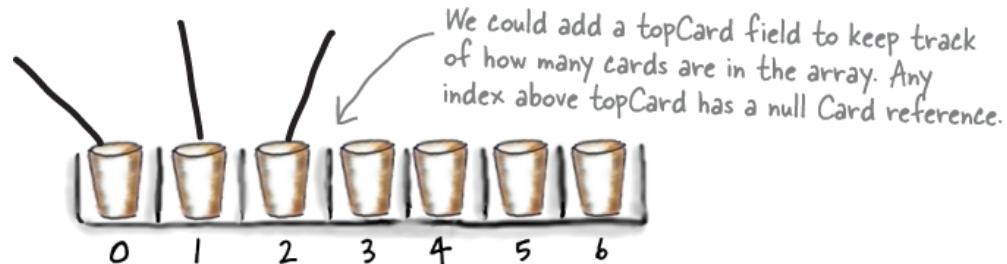
Arrays can be annoying to work with

An array is fine for storing a fixed list of values or references. But once you need to move array elements around, or add more elements than the array can hold, things start to get a little sticky.

1. Every array has a length, and you need to know the length to work with it. You could use null references to keep some array elements empty:



2. You'd need to keep track of how many cards are being held. So you'd need an int field, which we could call topCard, that would hold the index of the last card in the array. So our three-card array would have a Length of 7, but we'd set topCard equal to 3.



The AddWorker method from Chapter 6 used the Array.Resize method to do exactly this.

3. But 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. But what if you want to add a card? If topCard is less than the array's Length, you can just put your card in the array at that index and add 1 to topCard. But if the array's 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 topCard, you'll need to make sure to set the removed card's array index back to null. And 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 back...wow, what a mess!

Lists make it easy to store collections of... anything

C# and .NET have a bunch of **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:

1. **First you create a new instance of `List<T>`.** 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 way. 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.

We'll sometimes leave the <T> off because it can make the book a little hard to read. When you see List, think List<T>.



RELAX

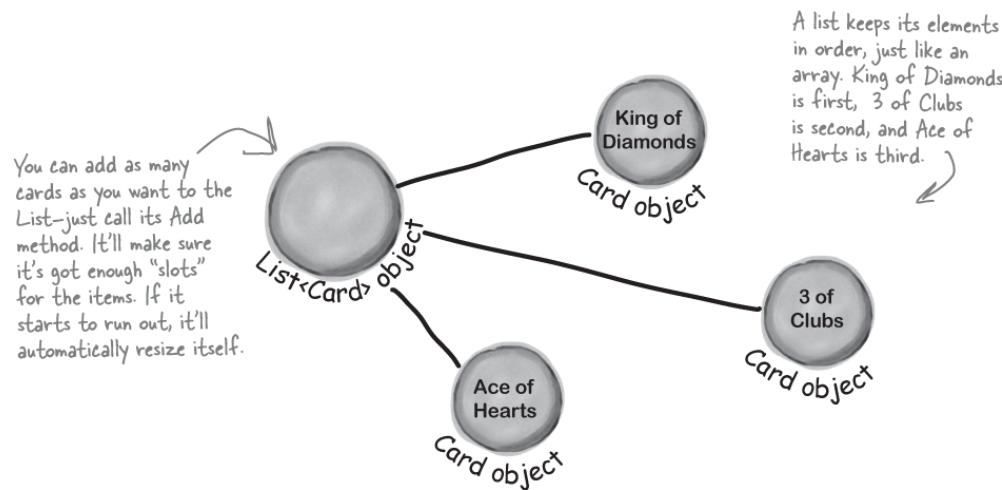
The <T> at the end of List<T> means it's generic.

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));
```



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>.

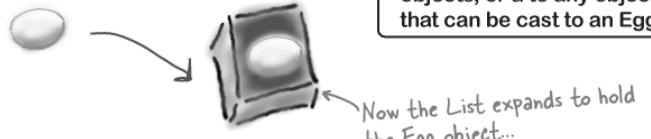
- 1** Use the `new` keyword to instantiate a List (like you'd expect!).

```
List<Egg> myCarton = new List<Egg>();
```

`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 a to any object that can be cast to an Egg.

- 2** Add something to a List.

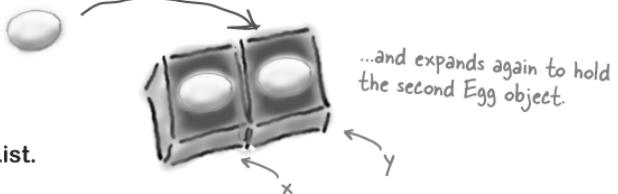
```
Egg x = new Egg();  
myCarton.Add(x);
```



Now the List expands to hold the Egg object...

- 3** Add something else to your List.

```
Egg y = new Egg();  
myCarton.Add(y);
```



...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 idx = myCarton.IndexOf(y);
```

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(y);
```



When we removed y, we left only x in the List, so it shrank! And eventually it will get garbage-collected.



SHARPEN YOUR PENCIL

Fill in the rest of the table below by looking at the List code on the left and putting in 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.

Here are a few lines from the middle of a program. Assume these statements are all executed in order, one after another, and that variables were previously declared.

List	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 putting in 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>	<code>bool foundIt = false; for (int i = 0; i < myList.Length; i++) { if (b == myList[i]) { isIn = true; } }</code>

Lists are objects that use 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.

And take another look at this—you can access the element at a specific index of a list using its **indexer**, just like you do with an array:

`Guy o = guys[1];`

With arrays, you're a lot more limited. You need to set the size of the array when you create it, and any logic that'll need to be performed on it will need to be written 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 three shoes and then removing them:

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: 4



Enter the color: black
The shoe closet contains:
Shoe #1: a black Wingtip

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

Press 'a' to add
a shoe, then
choose the type
of shoe and
type in the color.

Enter the color: blue and white
The shoe closet contains:
Shoe #1: a black Wingtip

Shoe #2: a blue and white Sneaker
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: 2

Press 'r' to
remove a shoe,
then enter
the number
of the shoe
to remove.

Enter the color: brown
The shoe closet contains:
Shoe #1: a black Wingtip

Shoe #2: a blue and white Sneaker
Shoe #3: a brown Sandal
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



The shoe closet contains:
Shoe #1: a black Wingtip
Shoe #2: a brown Sandal

Press 'a' to add or 'r' to remove a shoe: r
Enter the number of the shoe to remove: 2
Removing a brown Sandal

The shoe closet contains:
Shoe #1: a black Wingtip
Press 'a' to add or 'r' to remove a shoe: r

Enter the number of the shoe to remove: 1
Removing a black Wingtip

The shoe closet is empty.

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. Use 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,
}
```

*{ Remember from earlier
that you can cast an
enum to and from an
int. So Sneaker is equal
to 0, Loafer is 1, etc.*

- 2. Here's 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 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}"; }
    }
}
```

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
class ShoeCloset
{
    private readonly List<Shoe> shoes = new List<Shoe>();
    public void PrintShoes()
    {
        if (shoes.Count == 0)
        {
            Console.WriteLine("The shoe closet is empty.");
        }
        else
        {
            This foreach loop iterates through the shoes list and writes a line to the console for each shoe.
            {
                Console.WriteLine("The shoe closet contains:");
                int i = 1;
                foreach (Shoe shoe in shoes)
                {
                    Console.WriteLine($"Shoe #{i++}: {shoe.Description}");
                }
            }
        }
    }
    public void AddShoe()
    {
        Console.WriteLine("\nAdd a shoe");
        for (int i = 0; i < 6; i++)
        {
            Console.WriteLine($"Press {i} to add a {(Style)i}");
        }
        Console.Write("Enter a style: ");
        if (int.TryParse(Console.ReadKey().KeyChar.ToString(), out int style))
        {
            Here's where we create a new Shoe instance and add it to the list.
            →
            Console.Write("\nEnter the color: ");
            string color = Console.ReadLine();
            Shoe shoe = new Shoe((Style)style, color);
            shoes.Add(shoe);
        }
    }
    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))
        {
            Here's where we remove a Shoe instance from the list.
            →
            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 call 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. But `int.TryParse` needs a string, not a char, so we call `ToString` to convert the char to a string.

4. Here's 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("Press '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;
            }
        }
    }
}

```

Since there's no break statement after the 'a' case, 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. And since we wanted uppercase 'A' to work the same as lowercase 'a', 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 a break before it, it falls through to the next case. You can even have statements between the two case statements. But be really careful with this—it's easy to accidentally leave out a break statement.

Try debugging through 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!

Now run your app and reproduce the sample output.



LIST CLASS MEMBERS UP CLOSE

The ShoeCloset manages its List<Shoe> by calling its Add method to add a shoe to the list, its RemoveAt method to remove a shoe from a specific index in the list, and its Count property to check if the list is empty.

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 generics 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 (or Go to Declaration)

The `List` class is part of .NET Core, which has a whole bunch of really useful classes, interfaces, types, and more. And 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 Mac



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.

This is what it means to be a generic collection. 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 this basic job.



BULLET POINTS

- **List** is a .NET class that lets you store, manage, and easily work with a set of objects.
- A List **resizes dynamically** to whatever size is needed. It's got a certain capacity—once you add enough data to the list, it'll grow 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 using **RemoveAt**.
- You declare the type of the List using a **type argument**, which is a type name in angle brackets. Example: List<Frog> means the List will be able to hold only objects of type Frog.
- To find out the index of an item in a List, use **IndexOf**.
- The **Count** property returns the number of elements in the list.
- You can use the **Contains method** to find out if a particular object is in a 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** (Mac) 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.*

```
static void Main(string[] args)
{
    List<string> values = new List<string>();
    values.Add("a value");
    foreach (string s in values) ✘ Exception Unhandled
    {
        values.Add("another value");
    }
}
```

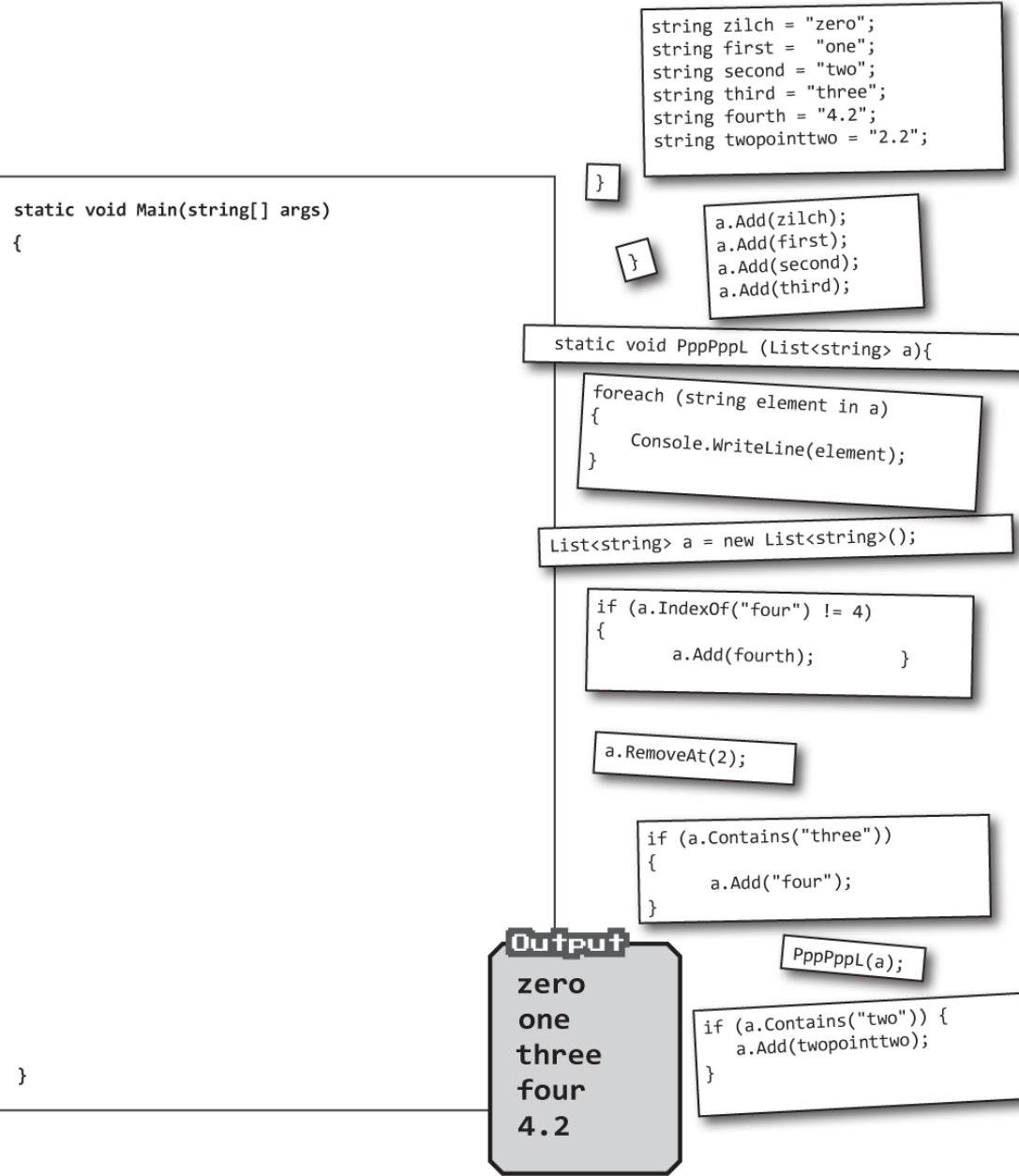
System.InvalidOperationException: 'Collection was modified; enumeration operation may not execute.'

[View Details](#) | [Copy Details](#) | [Start Live Share session...](#)
[Exception Settings](#)

Code Magnets

Can you reconstruct the code snippets to make a working Windows Form that will pop up the message box below when you click a button?



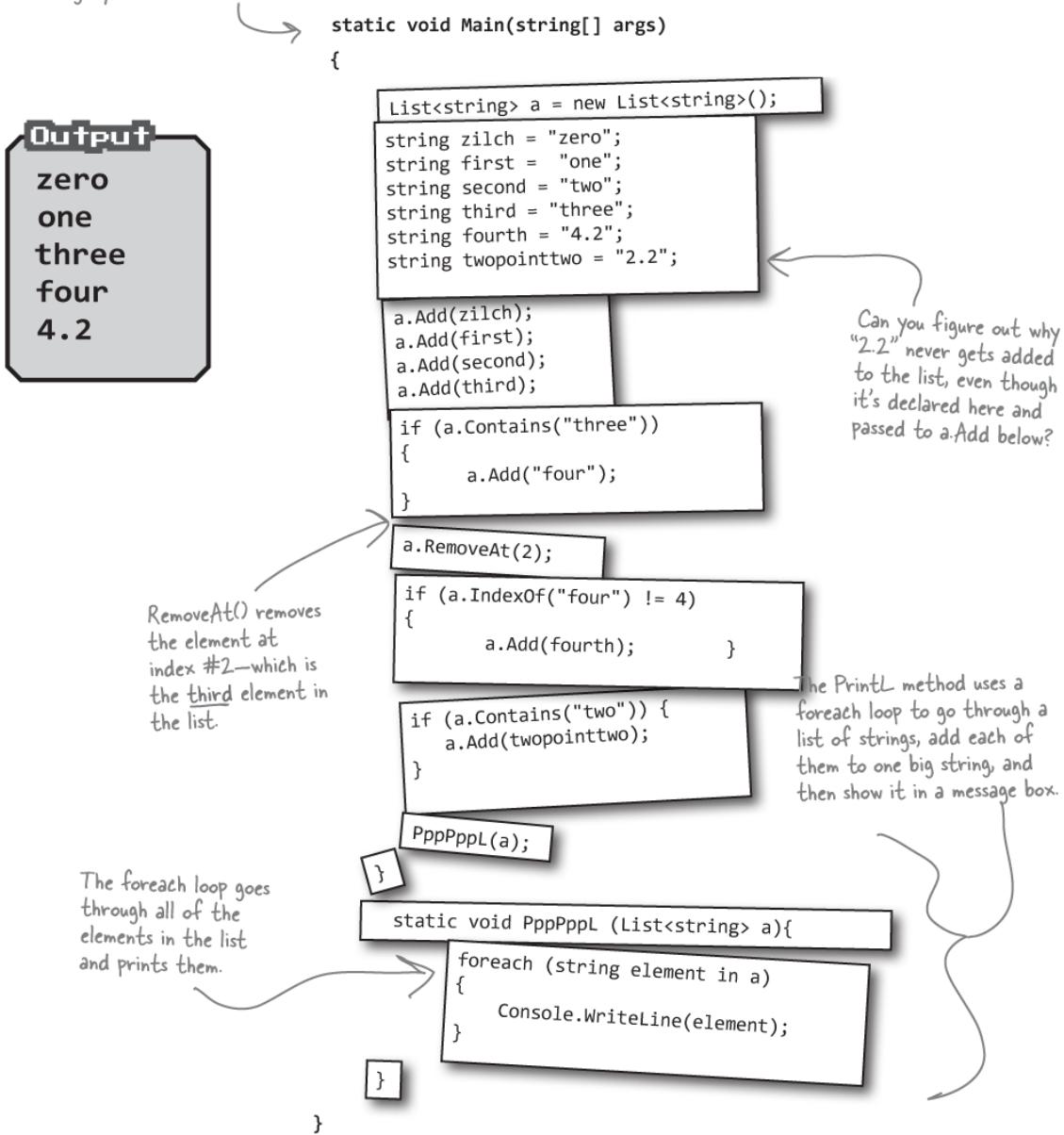


Code Magnets Solution



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!

If you want to run this code, make sure you have a "using System.Collections.Generic" line at the top.



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 you 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 on the first page of Chapter 4). You can cast them values, but not to 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. But 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: To be honest, if you need to store a collection of objects, you'll generally use a list and not an array. One place where you use arrays (which you'll see later in the book) is when you're reading with sequences of bytes—for example, when you read them 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).

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.

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, which 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 big white packages 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>` stuff is all about. It's the way that you tie a specific instance of a `List` to one type. But the `List` class as a whole is generic enough to work with ANY type. That's why generic collections are different from anything you've seen so far.

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. But you really shouldn't ever need to do this, because a `List<object>` will work just fine!

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.

```
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" });
```

This code creates a new `List<Shoe>` and fills it with new `Shoe` objects by calling the `Add` method over and over again.

Notice how each `Shoe` object is initialized with its own object initializer? You can nest them inside a collection initializer, just like this.

```
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" },  
};
```

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.

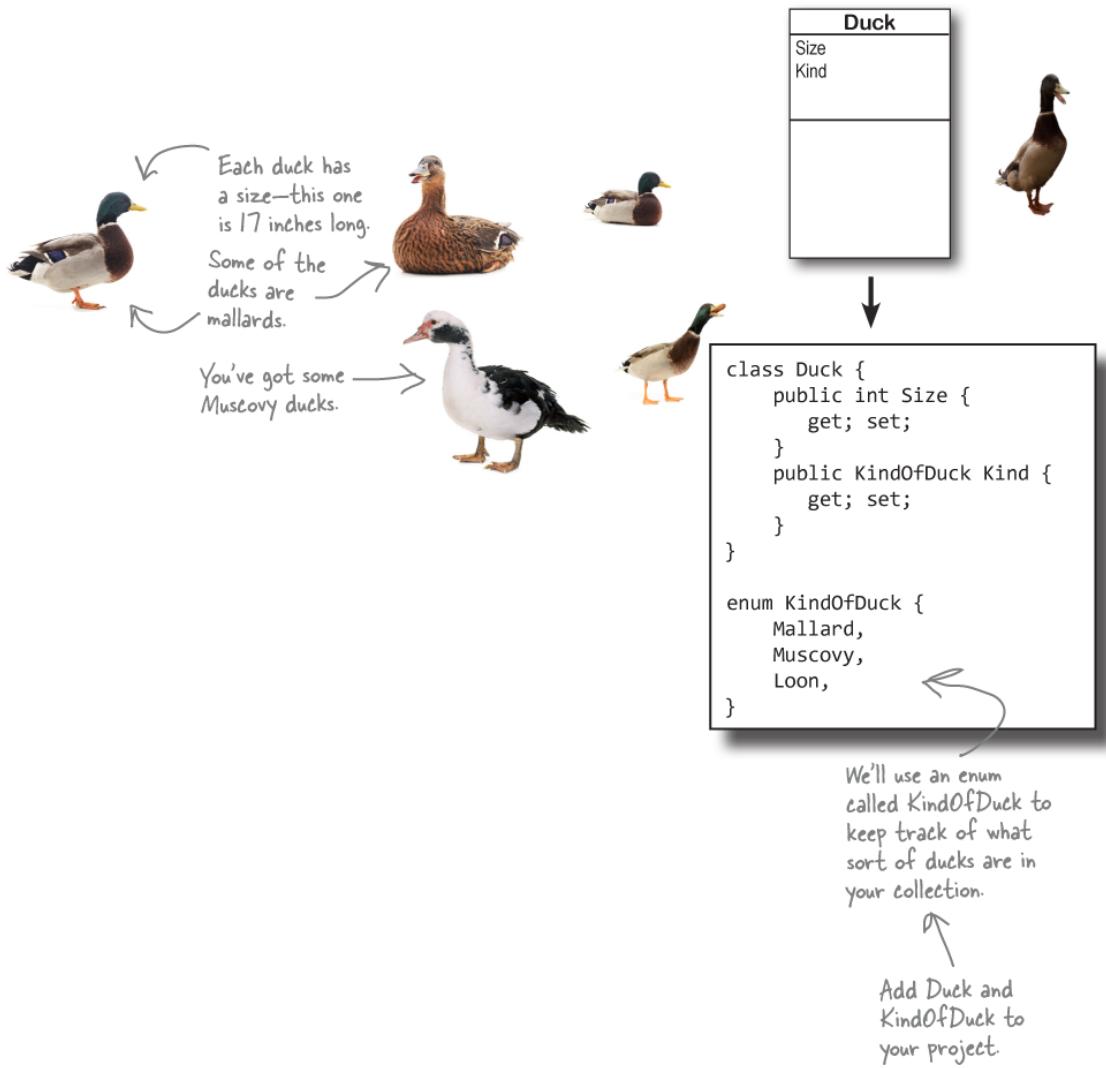
You can create a collection initializer by taking each item that was being added using `Add()` and adding it to the statement that creates the list.

A collection initializer makes your code more compact by letting you combine creating a list with adding an initial set of items.

Do this!

Let's create a List of Ducks

Here's a Duck class that keeps track of your extensive duck collection. (You *do* collect ducks, don't you?) **Create a new Console Application** and add a new Duck class and KindOfDuck enum.



Here's the initializer for your List of Ducks

We've got six ducks, so we'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` field. 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:

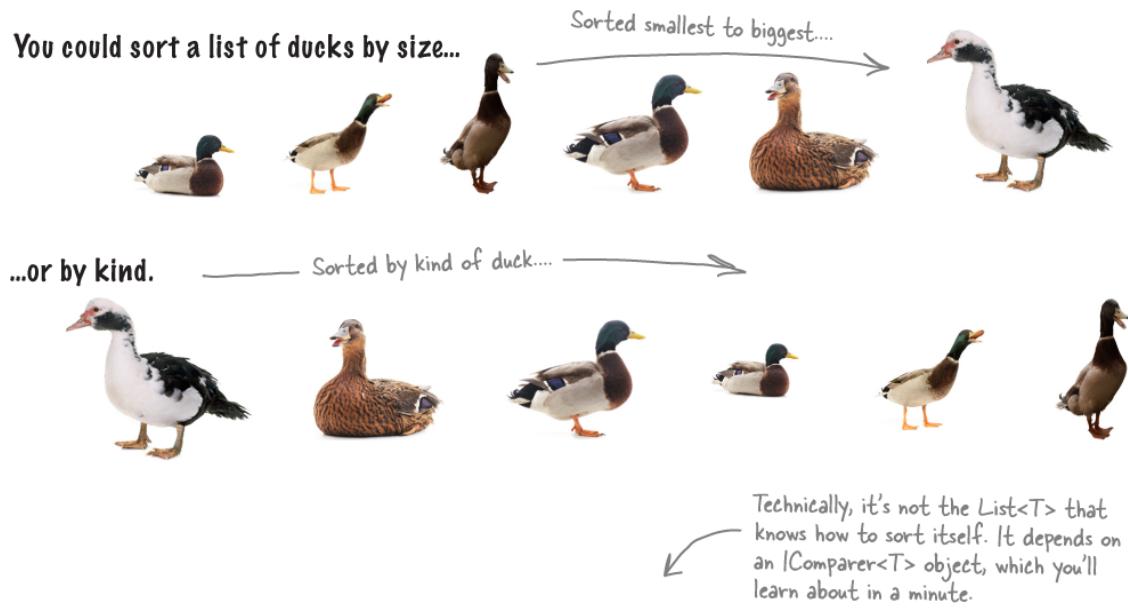
```
List<Duck> ducks = new List<Duck>() {
    new Duck() { Kind = KindOfDuck.Mallard, Size = 17
},
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 18
},
    new Duck() { Kind = KindOfDuck.Loon, Size = 14 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 11
},
    new Duck() { Kind = KindOfDuck.Mallard, Size = 14
},
    new Duck() { Kind = KindOfDuck.Loon, Size = 13 },
};

PrintDucks(ducks);
```

Run your code—it will print a bunch of ducks to the console.

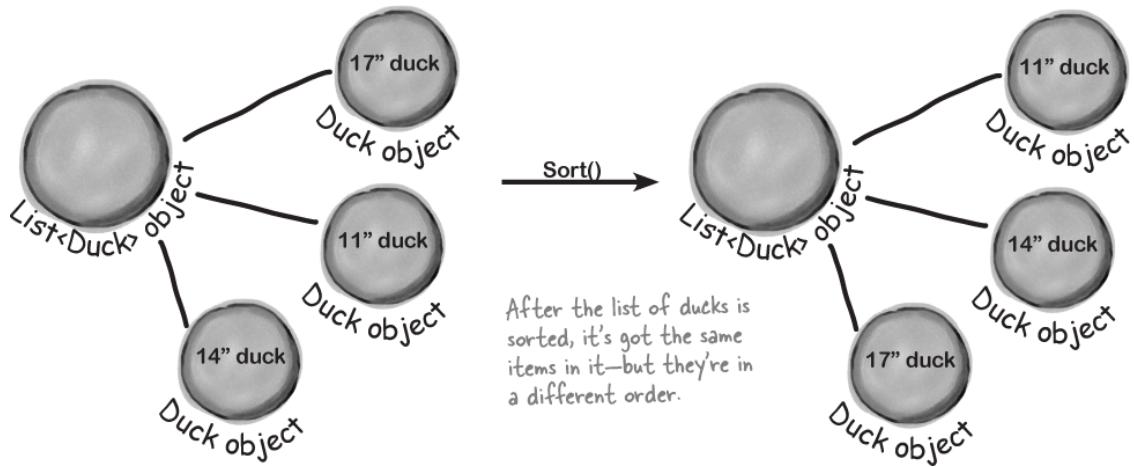
Lists are easy, but SORTING can be tricky

It's not hard to think about ways to sort numbers or letters. But what do you sort two objects on, 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 any of them.



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.



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. But how does the List know which way to sort your Duck objects? We tell the List.Sort method that the Duck class can be sorted. And we do that the way always indicate that a class can do a certain thing: *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.

You can make any class work with the List's built-in Sort method by having it implement `IComparable<T>` and adding a `CompareTo` method.

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. If the duck to compare should come after the current duck in the sorted list, `CompareTo()` returns a positive number.

Update your project's Duck class by implementing `IComparable<Duck>` so that it sorts itself based on duck size:

```
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;  
    }  
}
```

When you implement `IComparable<T>`, you specify the type being compared when you have the class implement the interface.

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 zero.

If you want to sort your list from smallest to biggest, have `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.

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

When you typed “List.Sort” into the IDE, did you notice that it has overloads? Have a look:

▲ 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`.

List.Sort 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.

You've been calling `List.Sort` without any arguments. But the IDE says you can also call an overloaded version 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.). And since pass it an instance of an

object that implements an interface, we know what that means: that it *does a specific job*. And in this case, that job is comparing two ducks (or whatever item is in the list).

The `IComparer<T>` interface has one member, a **method called Compare**, and it's like the `CompareTo` method in `IComparable<T>`: it takes two object parameters, `x` and `y`, and returns an `int`. If `x` is less than `y`, it should return a negative value. If they're equal, it should return zero. And if `x` is greater than `y`, it should return a positive value.

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 Compare returns a
        if (x.Size > y.Size)   negative number, that means
            return 1;           object x should go before
        return 0;              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."

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. That object exists for one reason—to help `List.Sort()` figure out how to sort the array. But like any other (nonstatic) class, you need to instantiate it before you use it.

```
IComparer<Duck> sizeComparer = new DuckComparerBySize();
ducks.Sort(sizeComparer);
PrintDucks(ducks);
```

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.

You'll pass `Sort()` a reference to the new `DuckComparerBySize` object as its parameter.



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;
    }
}
```

This comparer sorts by duck type. Remember, when you compare the enum `Kind`, you're comparing their index values.

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.

So Mallard comes before Muscovy, which comes before Loon.

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,
}

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);
}
```

This enum tells the object which way to sort the 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 to the end of your `Main` method, 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 Application. 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 chance to use the IDE Quick Fix 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 box appear underneath it. When you click on the box, the IDE pops up its Quick Fix window with an "Implement interface" option:



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)`

3. Make sure the output looks right.

Write the Main method so the output looks like this.

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

- 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

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.

```
class CardComparerByValue : IComparer<Card>
{
    public int Compare(Card x, Card y)
    {
        if (x.Suit < y.Suit) } We want all diamonds to come
        return -1; before all clubs, so we need to
        if (x.Suit > y.Suit) } compare the suits first. We can
        return 1; take advantage of the enum values.
        if (x.Value < y.Value) } These statements only get
        return -1; } executed if x and y have the
        if (x.Value > y.Value) same value—that means the
        return 1; first two return statements
        return 0; weren't executed.
    }

    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.Write("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 ...\n");

            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.

If none of the other four return statements were hit, the cards must be the same—so return zero.

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?

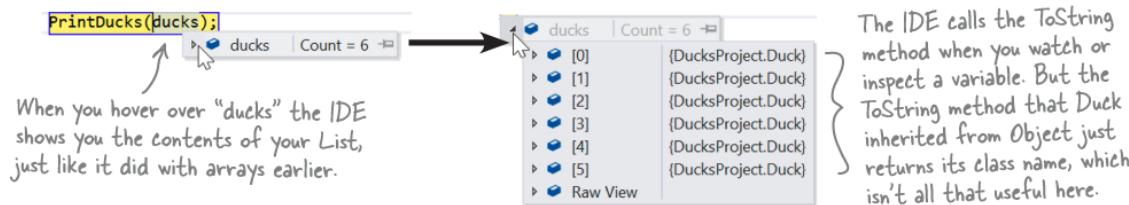
Overriding a `ToString` method lets an object describe itself

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 or the + operator to concatenate strings, 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`:

```
Console.WriteLine(new Duck().ToString()); → "DucksProject.Duck"
```

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 was `DucksProject.Duck`.

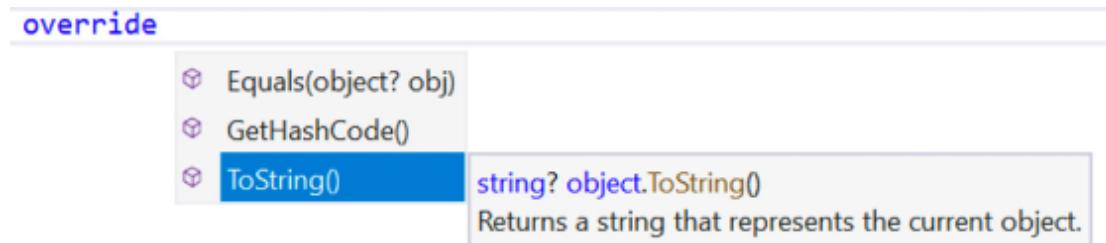
The IDE also calls the `ToString` method—for example, when you watch or inspect a variable:



Hmm, that's not as useful as we'd hoped. You can see that there are six `Duck` objects in the list ("DucksProject" is the namespace we used). If expand a `Duck`, you can see its `Kind` and `Size` values. But 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 on `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 press space, the IDE will show you the methods you can override:

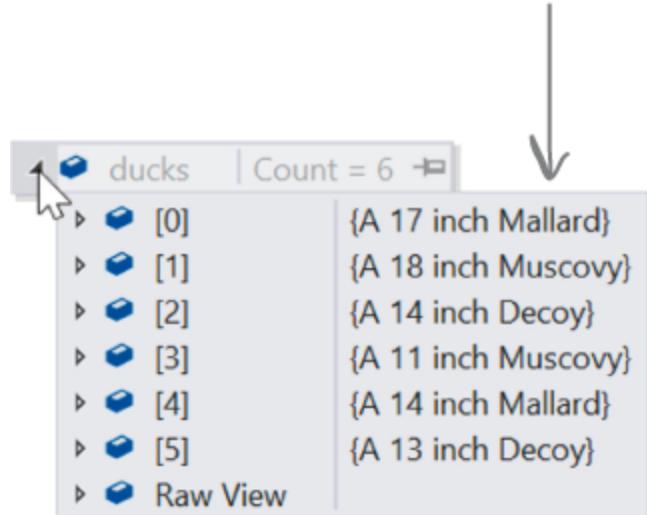


Click on `ToString()` to tell the IDE to add a new `ToString` method. Replace the contents so it looks like this:

```
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.

When the IDE's debugger shows you an object, it calls the object's `ToString` method.



Update your foreach loops to let your Ducks and Cards print themselves

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. But 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. And now if you want to add, say, a `Gender` 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;
}
```

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.

Now your programs that use `Card` objects will be easier to debug.



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 in here? See if you can figure it out just from reading the code, and without running the app.



SHARPEN YOUR PENCIL SOLUTION

Here's the output of the app. Did you get it right? It's okay 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 Porkchop

A Corgi named Carrie



BULLET POINTS

- **Collection initializers** let you specify 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, and 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>**.

And 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 a simple iteration over a non-generic collection**—or in simple terms, it lets you write code that loops through it. And the most common use iterate over an `IEnumerable<T>` is 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?

When a collection implements `IEnumerable<T>`, it's giving you a way to write a loop that goes through its contents in order.

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();
```

Technically, there's a little more code than this, but this is enough to give you the basic idea of what's going on.

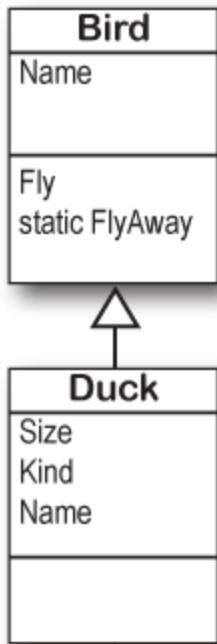
Both loops write the same ducks to the console. See this for yourself by running both of them—they'll both have the same output. (And don't worry about the last line for now; you'll learn about `IDisposable` in [Chapter 10](#).)

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 list, or false if the enumerator has reached the end of the list. 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 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.

Covariance is really useful when we need to pass a collection of objects to a method that only works with the class they inherit from.

Do this!

1. **Create a new Console Application.** 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);
        }
    }
}
```



The static `FlyAway` works with a collection of birds. But what if we want to pass a list of ducks?

Covariance is C#'s way of letting you implicitly convert a subclass reference to its superclass. That word "implicitly" just means you 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. 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. Make sure you also **add the KindOfDuck enum** from earlier in the chapter.

```
class Duck : Bird {           ← We added Bird to the declaration to make the
    public int Size { get; set; }
    public KindOfDuck Kind { get; set; }

    public override string ToString()
    {
        return $"A {Size} inch {Kind}";
    }
}
```

```
enum KindOfDuck {
    Mallard,
    Muscovy,
    Loon,
}
```

```
enum KindOfDuck {  
    Mallard,  
    Muscovy,  
    Loon,  
}
```

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 },  
};
```

} } Copy the same collection
initializer you've been
using to initialize your
List of ducks.

```
Bird.FlyAway(ducks, "Minnesota");
```

✖ 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. And the error message telling you that you can’t convert your Duck collection to a Bird collection. So let’s try assigning the ducks to a List< Bird> :

```
List<Bird> upcastDucks = ducks;
```

Well, that didn’t work. You got a different error, but it still says you 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 that we 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> ?

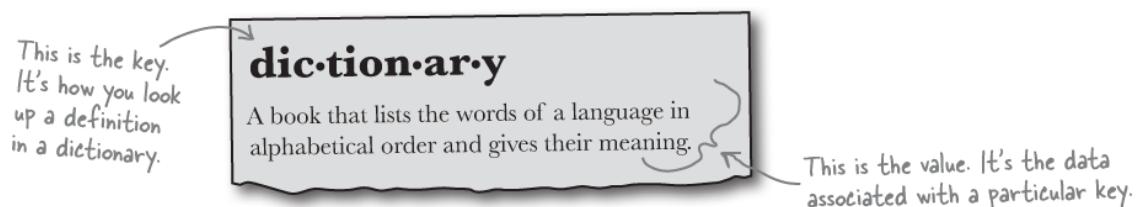
4. Use covariance to make your ducks fly away.

And that’s where **covariance** comes in: you can **use assignment to upcast your List<Duck> to an IEnumerable<Bird>**. And once you’ve got your `IEnumerable< Bird>`, you can call its `ToList` method to convert it to a `List< Bird>`.

```
IEnumerable<Bird> upcastDucks =  
ducks;  
Bird.FlyAway(upcastDucks.ToList(),  
"Minnesota");
```

Use a Dictionary to store keys and values

A list is like a big long page full of names. But 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 a **dictionary**. A dictionary lets you take a special value—the **key**—and associate that key with a bunch of data—the **value**. And one more thing: a specific key can **only appear once** in any dictionary.



Here's how you declare a 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 used to look 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 for a few friends.

```
using System.Collections.Generic;
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";
    }

    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");
    }
}
```

← You need this using directive to work with Dictionary, like you do with List.

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.

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 Dictionary methods:

- 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 will a string value.

```
string lookupValue =
myDictionary["some key"];
```

- **Remove an item.**

Just like 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;
```

Your key and value can be different types

It's common to see a dictionary that maps integers to objects when you're assigning unique ID numbers to objects.

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

Do this!

Here's a quick program that any New York baseball fan 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;
```

```
    }  
}
```

Yogi Berra was #8 for the New York Yankees, while Cal Ripken Jr. was #8 for the Baltimore Orioles. But in a dictionary only one key can map to a single value, so we'll only include numbers from one team here. Can you think of a way to store retired numbers for multiple teams?

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**. And 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!

```
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 objects are two of the **built-in generic collections** that are part of the .NET Framework. Lists and dictionaries are very flexible—you can access any of the data in them in any order. But sometimes you need to restrict how your program works with the data because the *thing* that you're representing inside your program works like that in the real world. For situations like this, you'll use 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:

- 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:

- 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!



The 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. And 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. But no matter which of these collections 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 >`.

A queue is like a list that lets you put objects on the end of the list and use the ones in the front. A stack only lets you access the last object you put into it.

You can, however, use foreach to enumerate through a stack or queue, because they implement `IEnumerable`!

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.

```
// 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");
```

After the first Dequeue call, the first item in the queue is removed and returned, and the second item shifts into the first place.

```
// Peek "looks" at the first item in the queue without removing it
Console.WriteLine($"Peek() returned:\n{myQueue.Peek()}");
```

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.

①

```
// Dequeue pulls the next item from the FRONT of the queue
Console.WriteLine(
    $"The first Dequeue() returned:\n{myQueue.Dequeue()}");
Console.WriteLine(
    $"The second Dequeue() returned:\n{myQueue.Dequeue()}");

// Clear removes all of the items from the queue
Console.WriteLine($"Count before Clear():\n{myQueue.Count}");
myQueue.Clear();
Console.WriteLine($"Count after Clear():\n{myQueue.Count}");
```

②
③
④
⑤



Output

```
Peek() returned:
first in line ①
The first Dequeue() returned:
first in line ②
The second Dequeue() returned:
second in line ③
Count before Clear():
2 ④
Count after Clear():
0 ⑤
```

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.

```
// Peek with a Stack works just like it does with a Queue  
Console.WriteLine($"Peek() returned: \n{myStack.Peek()}");
```

When you push an item onto a stack, it pushes the other items back one slot and sits on top.

```
// Pop pulls the next item from the BOTTOM of the stack  
Console.WriteLine(  
    $"The first Pop() returned: \n{myStack.Pop()}");  
Console.WriteLine(  
    $"The second Pop() returned: \n{myStack.Pop()}");
```

When you pop an item off the stack, you get the most recent item that was added.

```
Console.WriteLine($"Count before Clear(): \n{myStack.Count}");  
myStack.Clear();  
Console.WriteLine($"Count after Clear(): \n{myStack.Count}");
```

4
5

Output

```
Peek() returned:  
last in line ①  
The first Pop() returned:  
last in line ②  
The second Pop() returned:  
third in line ③  
Count before Clear():  
2 ④  
Count after Clear():  
0 ⑤
```

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. And 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. The 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 calling 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 them 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 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.





BULLET POINTS

- Lists, arrays, and other collections implement the **IEnumerable<T> interface**, which supports simple 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 we 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 **simple 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 to 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.

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 an 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 the `Dictionary.TryGetValue`, which lets you do this:

```
if (dict.TryGetValue("Key", out string value))  
{  
    // do something  
}
```

which does exactly the same thing as this:

```
if (dict.ContainsKey("Key"))  
{  
    string value = dict["Key"];  
    // do something  
}
```

GAME DESIGN... AND BEYOND



Keyboards and controllers

You've probably played games that use a familiar control scheme: video game controllers with two joysticks, four buttons, and a D-pad, or a mouse to look, W-A-S-D keys to move, spacebar to jump. That's the result of decades of evolution.

- The **WASD layout** dates back to the mid-1980s. Early PC games from the '80s and early '90s were more likely to use arrow keys, taking advantage of their "inverted T" layout. It wasn't until the popularity of first-person shooters in the late '90s—especially Quake (1996) tournaments and Half Life (1998)—that the WASD layout really took hold.
- The first video game console, the Magnavox Odyssey (1972), had a controller with two **paddles**, or knobs that would each control horizontal or vertical movement. While we don't see paddles much anymore, in many ways they're the predecessor to modern racing wheel controllers.
- The Atari 2600 (1977) was enormously successful, and its **joystick**, with one button and an 8-directional stick, was the first ubiquitous game controller. It became a standard for many game systems and computers in the 1980s—there were even adaptors for the IBM PC and Apple][.



Modern game controllers have similar layouts across different consoles. But game controllers, like keyboard layouts, actually evolved in a symbiotic relationship along with the games that used them.

- Nintendo introduced the **D-Pad** (or control pad), a flat rocker button, with their 1983 Famicom/NES console. Since then, a 4- or 8-direction D-Pad has been a mainstay of controllers, including the current Xbox One controller.
- Their SNES (1990) also introduced the **popular layout** still seen in modern game controllers, with the D-Pad on the left, select and start buttons in the

middle, a diamond of four buttons on the right, and shoulder buttons on the rear.

- Sony **set the standard for modern video game controllers** with its PlayStation DualShock (1997) that featured dual analog joysticks, a familiar button layout, rumble for physical feedback, and an ergonomic shape.
- Nintendo continued to push the boundaries, popularizing **motion controllers** that track the player's physical movements with their 2006 Wii console, an important step in making games more immersive and engaging.

Video game controls are about more than just hardware. A game's control scheme—or what the different keys, clicks, buttons, and stick movements, actually do—makes a huge difference in gameplay.

- While not the first game to use a joystick, **Pac Man**, released in 1980—during a period known as the golden age of video games—featured the now-iconic ball top stick. It stood out for its intuitive play: move the stick up, the player goes up; move it down, the player goes back down.
- A control scheme can **affect the difficulty** of a game. The arcade game *Defender* (1980), one of the first games with a side-scrolling, multi-screen playfield, is remembered as one of the most difficult of the era, in part due to its control scheme with less-intuitive buttons to thrust and reverse direction.
- Many modern games feature a familiar scheme for a two-stick controller: move with the left stick, look with the right stick. But this was the result of many years of **symbiotic evolution** between game designers, hardware, and players.
- In the early 2000s, when all of the major consoles featured dual-stick controllers, game designers were still figuring out how to work with them, and some games introduced **multiple options** with different control schemes.
- As players and game designers got used to these new controller designs, they found new ways to use them in games. With more buttons came **combos**, a gameplay element that involves an often complex set of actions that the player must be performed in sequence—like a set of button presses in a fighting game that yields an unblockable attack.





Getting IDE shortcuts into your muscle memory helps you code.

When you've been working on one of the projects in this book, have you had the feeling of time flying by? If you haven't felt that yet, don't worry... it'll come! Developers call that "flow"—it's a state of high concentration where you start working on the project, and after a while (and if there are no interruptions) the rest of the world sort of "slips away" and you find time passing quickly. This is a real state of mind that psychological researchers have studied—especially sports psychologists (if you've ever heard an athlete talk about being "in the zone," it's the same idea). Artists, musicians, and writers also get into a state of flow. And getting really familiar with the IDE's keyboard shortcuts can help you more quickly get into—and stay in—your own state of flow.

IDE Tip: Keyboard Shortcuts

Get familiar with the keyboard shortcuts in your IDE! And Microsoft has put together a really handy reference with some of the handiest shortcuts: <https://aka.ms/vsm-vs-keys> – print that out and stick it on the wall near your computer.

Also take a look at Microsoft's documentation page on navigating code in Visual Studio, which has more useful shortcuts: <https://docs.microsoft.com/en-us/visualstudio/ide/navigating-code>

Take the time to get especially familiar with the various options that pop up in the Quick Fix menu. You've already used it to implement interfaces and generate methods... but it does so much more! Next time you need to add a using declaration, add the code that needs it and then pop up the Quick Fix menu – you'll see an option to add the missing using declaration.

But don't take our word for it. Here's some great advice from Tatiana Mac, a great developer, and also a concert pianist—which gives her a unique viewpoint:

For me, effective coding is about transferring rote actions to muscle memory to maintain my mind palace. Being a power keyboard user is the trick (natural transition for a former concert pianist). Even if you don't play the piano, here's how you can keyboard more. You will:

- *Appreciate keyboard accessible programs/sites!*
- *Learn common patterns. Mistake keys will reveal other new shortcuts!*
- *Be slow at first, but that's okay!*

Shortcuts will speed you up and, more importantly, reduce cognitive load and help you keep focus. You can get fancy and learn more complex ones. I suggest getting adept at these first. Small learnings like this compound over time. You won't notice how much it helps until you switch code editors and realize how engrained you are.

You can see her full advice here, including specific keyboard commands that she recommends you learn to get a great start:

<https://twitter.com/TatianaTMac/status/1243209483947356161>

Let's build an app to work with decks of cards

In the next exercise you'll build an app that lets you move cards between two decks. It will have two `ListBox` controls that show the cards in each deck. 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. You'll use a `Grid` with two columns and four rows that contain two `ListBoxes`, four buttons—and two new controls you haven't seen yet, `Label` controls, which we'll use to give your `ListBoxes` **access keys**.

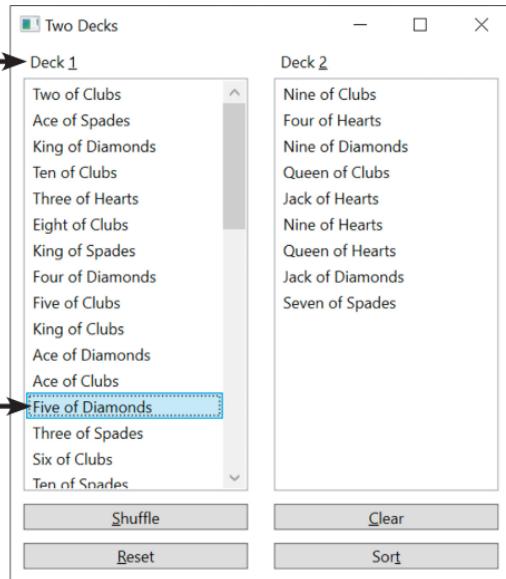
Pop up most apps—including Visual Studio—and tap the Alt key. You'll see underscores display under various buttons, menu items, and other controls. Those are access keys. Hold down alt and tap the key and you jump straight to the control that has it underlined. We'll use access keys to make your app fully keyboard accessible.

This is a Label control. It's used to label other controls, and provides access keys that you can use to jump straight to them. You set its access key by putting an underscore _ to the left of a character in the content, and set use its Target to set the control that it jumps to—and the IDE helps makes it easy to set the Label's target.

The app lets the user move cards between the two decks in one of two ways: double-clicking on the card that you want to move, or selecting it (using mouse or arrow keys) and pressing enter. Both actions remove the card and add it to the other deck.

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

<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
</Grid.RowDefinitions>
```



Row 0 has its height set to "Auto" so it grows or shrinks to fit its Label controls.

Row 1 has its height set to the default 1* so it fills up the rest of the window.

Rows 2 and 3 also have their row height set to "Auto" so they grow to fit their Buttons.

Here are the grid row and column definitions. The top row and bottom two rows have their height set to Auto, which causes them to expand to fit the contents of their cells. The second row has the default height (the same as 1*) so it fills up the rest of the window. Each ListBox has its vertical alignment set to Stretch so it fills up the row. That combination causes the ListBoxes to expand to fill up the space not taken up by the other controls.



LONG EXERCISE

Let's break this project into a few parts. In this first part, you'll lay out the window for your app. But we want this app to be fully **keyboard accessible**. You'll add **access keys** to your buttons, and use **Label controls** to add access keys to your ListBoxes.

1. Set the **title and width** of the <Window>: Title="Two Decks" Height="450" Width="400" – then **add the row and column definitions** that we just gave you. Now you're ready to **add the controls**. Here's what you'll add:

- Add a Label control to each of the two cells in the top row of the Grid. Set the values for their Content properties to "Deck 1" and "Deck 2" and use the Properties window to name them deck1Label and deck2Label.
- Add a ListBox control to the two cells in the second row (Grid.Row="1"). Use the Properties window to name the two listboxes leftDeckListBox and rightDeckListBox.
- Add a Button control to each cell in the bottom two rows and set their Contents to match the screenshot ("Shuffle" and "Reset" in the left column, "Clear" and "Sort" in the right column). Name the four buttons shuffleLeftDeck, resetLeftDeck, clearRightDeck, and sortRightDeck.
- Give each Label control a left margin of 10 (Margin="10,0,0,0") and the other controls left, right, and bottom margins of 10 (Margin="10,0,10,10"). Reset all other properties to their defaults.

2. **Add an access key to each button** by adding an underscore in the Content just before the letter the app should use as an access key. For example, we want the focus to jump to the Shuffle button when the user presses Alt-S, so its Content will be: Content="_Shuffle" — and remember, the access keys are normally hidden when you run the app, but you can test them by tapping the alt key to make it reveal the access keys. The other Button contents will be "_Reset", "_Clear", and "Sor_t" so they can be accessed with Alt-R, Alt-C, and Alt-T (but pressing them won't do anything yet, because they're clicking buttons that don't do anything).

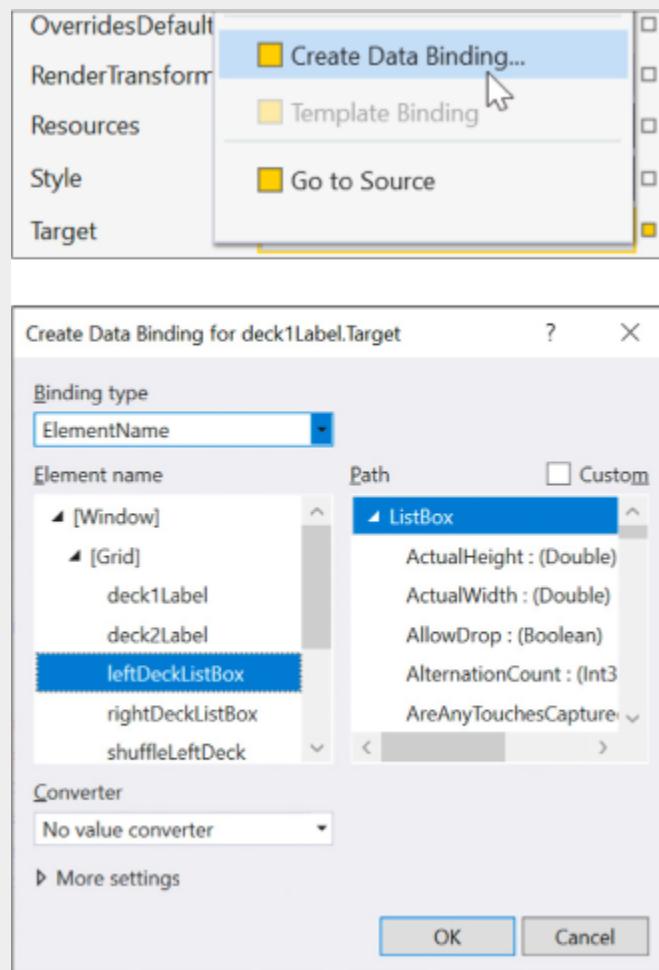
3. **Add access keys to the two Label controls** by setting their contents to "Deck _1" and "Deck _2". That's not all you need to do to get the labels to work. When the user presses Alt-1 or Alt-2, we want the focus to jump to the

ListBox controls, not the Labels. We can do this by **adding a target**—and Visual Studio makes it really easy to do that.

1. Click on the Label in the Designer or XAML editor to select it. Then in the Properties window, find Target (under Miscellaneous):



2. Click on the box next to Target and choose **Create Data Binding...**
3. The IDE will pop up its Create Data Binding window, and you should see the names of the controls that you set in Step 1 above. The Element Name box shows the controls in your window in a nested hierarchy, with Window at the top, Grid under that, and the controls you added to the Grid. Click on the control that you want to focus on when the user uses the Label's access key: for the Deck 1 label choose leftDeckListBox, and for the Deck 2 label choose rightDeckListBox.



Once you've set the Target properties, run your app and test access keys by running the app. You should see the focus jump to the left ListBox when you press Alt-1, and it should jump to the right ListBox when you press Alt-2.



LONG EXERCISE SOLUTION

Here's the XAML that you created in the first part of the exercise. You laid out a window two columns and four rows, taking advantage of Height="Auto" in the row definitions to let three of the rows expand to fit their contents. You two ListBoxes, two Labels with access keys and targets set to each ListBox, and four Buttons with their own access keys.

This is the solution to the Part 1. We'll get to the rest of the long exercise soon.

```
<Window x:Class="TestTwoDecks.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:TestTwoDecks"
    mc:Ignorable="d"
    Title="Two Decks" Height="450" Width="400" >
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        When you
        set the
        row's Height
        to "Auto"
        it expands
        to fit its
        contents.
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        Here are the grid row
        and column definitions
        that we gave you.
        <Label x:Name="deck1Label" Content="Deck 1"
            Margin="10,0,0,0" Target="{Binding ElementName=leftDeckListBox, Mode=OneWay}"/>
        <Label x:Name="deck2Label" Content="Deck 2" Grid.Column="1" Margin="10,0,0,0"
            Target="{Binding ElementName=rightDeckListBox, Mode=OneWay}"/>
        <ListBox x:Name="LeftDeckListBox" Grid.Row="1" Margin="10,0,10,10" />
        <ListBox x:Name="rightDeckListBox" Grid.Row="1" Grid.Column="1"
            Margin="10,0,10,10" />
        <Button x:Name="shuffleLeftDeck" Content="_Shuffle" Grid.Row="2"
            Margin="10,0,10,10" />
        <Button x:Name="clearRightDeck" Content="_Clear" Grid.Row="2" Grid.Column="1"
            Margin="10,0,10,10" />
        <Button x:Name="resetLeftDeck" Content="_Reset" Grid.Row="3"
            Margin="10,0,10,10" />
        <Button x:Name="sortRightDeck" Content="Sort" Grid.Row="3" Grid.Column="1"
            Margin="10,0,10,10" />
    </Grid>
</Window>
```

When you used Create Data Binding option in the Properties window to set the target for each Label control, it added these {Binding...} values. This is just like the data binding that you added to the Beehive Management System in Chapter 7.

When you use a Button control's access key, it clicks the button but doesn't change focus.

We had to use T as the access key for the Sort button because S was already taken by the Shuffle button.

Add a Deck class to hold the cards

Do this!

We want each ListBox to show us a different deck of cards.

Here's how you'll do that:

1. You'll create a Deck class—it will be a collection that holds Card objects
2. You'll add two instances of the Deck class to your window's resources (just like when did with an instance of the Queen class when you added data binding to your Beehive Management System app)
3. You'll use data binding to automatically populate each ListBox with the contents of one of the Decks

Create a Deck that extends ObservableCollection

Start by **adding the Suits and Values** enums to your project. Then **add the Card class**. Make sure you use the version that overrides the ToString object. Then **add this Deck class**:

```
using System.Collections.ObjectModel;  
class Deck : ObservableCollection<Card>  
{  
    // You'll fill in this class soon...  
}
```

Your Deck class will extend ObservableCollection.
That's a collection class that automatically notifies the app any time its items have changed.
You'll bind each ListBox to a Deck object. Make sure you include the using directive at the top.

You've worked with several different collections throughout this chapter: Lists, Dictionaries, Queues, and Stacks. Now

you'll use the ***ObservableCollection*** class, which is built for data binding. An ObservableCollection is a collection that has that same kind of notification built in, letting the app know whenever items get added, removed, or refreshed (just like the notification you added to your Queen class).

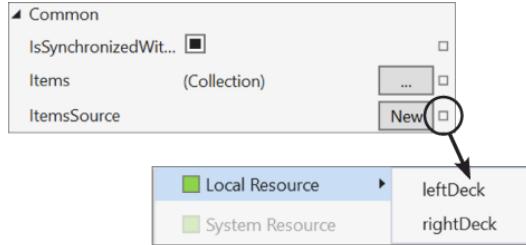
Add a Window.Resources with two instances of the Deck class

In Chapter 7 you added a Window.Resources tag to create an instance of the Queen class. Now you'll do something very similar. **Add this Window.Resources tag** to your xaml, just above the <Grid> tag:

```
<Window.Resources>
    <local:Deck x:Key="leftDeck"/>
    <local:Deck x:Key="rightDeck"/>
</Window.Resources>
```

Use Visual Studio to set up data binding

Click on the left ListBox in the designer, then expand Common in the Properties window and **click the box next to ItemsSource**. Choose Local Resource >> leftDeck from the menu. Then do the same thing for the right ListBox, setting its ItemsSource to rightDeck. Visual Studio will update the XAML to bind each ItemsSource to an instance of Deck:



Now the ListBox items are bound to the Deck instance, so they'll update any time the collection changes.

```
<ListBox x:Name="LeftDeckListBox" Grid.Row="1" Margin="10,0,10,10"
         ItemsSource="{DynamicResource leftDeck}" />
<ListBox x:Name="rightDeckListBox" Grid.Row="1" Grid.Column="1"
         Margin="10,0,10,10" ItemsSource="{DynamicResource rightDeck}" />
```



LONG EXERCISE

Finish the Two Decks app by adding the code-behind for the main window and implementing the Deck class. Each of the four buttons will need a Click event handler that calls a method on the correct instance of Deck. Each ListBox will get two event handlers: you can move cards by either double-clicking on the item in the ListBox or selecting an item in the ListBox and pressing enter while the ListBox is focused, so each ListBox will need a MouseDoubleClick event handler and a KeyDown event handler.

Start with this method to move a card from one deck to the other

Add this MoveCard method to the code-behind in MainWindow.xaml.cs. If you call MoveCard(true) it moves the card in that's currently selected in the left deck to the right deck. If you call MoveCard(false) it moves the selected card in the right deck to the left deck. Look carefully at the code—every piece of it is something you've learned so far.

```
private void MoveCard(bool leftToRight) {
    if ((Resources["rightDeck"] is Deck rightDeck) && (Resources["leftDeck"] is Deck leftDeck)) {
        if (leftToRight) {
            if (leftDeckListBox.SelectedItem is Card card) {
                leftDeck.Remove(card);
                rightDeck.Add(card);
            }
        } else {
            if (rightDeckListBox.SelectedItem is Card card) {
                rightDeck.Remove(card);
                leftDeck.Add(card);
            }
        }
    }
}
```

The `ListBox.SelectedItem` property returns a reference to the item that's currently selected. You used data binding to set the `ItemsSource` to a collection of Cards, so the selected item will be a Card reference.

The `Window.Resources` tag that you added to the XAML created a resource dictionary with two entries: references to two instances of the Deck class with keys `leftDeck` and `rightDeck`. Look closely at this code—it's how you safely get a reference to the left deck.

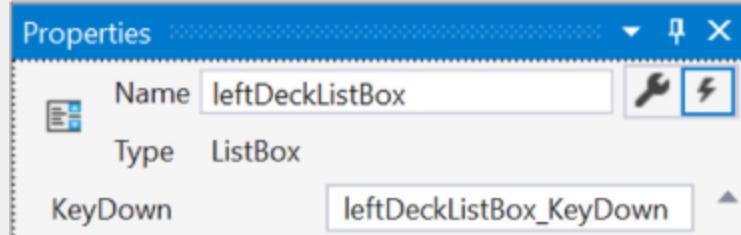
Modify the MainWindow constructor to safely get a reference to the right deck and call its Clear method.

Add a Click event handler to each Button and KeyDown and MouseDoubleClick event handlers to each ListBox

Double-click on each button to **add a Click event handler** for each button. Just like before, the IDE will automatically generate names like `shuffleLeftDeck_Click` because you set the Name properly. Use the `is` keyword to get a reference to either the right or left deck (like in the MoveCard method above) and call the appropriate method on the Deck class.

Next, use the Properties window to **add a KeyDown event handler and a MouseDoubleClick event handler** to each ListBox. The MouseDoubleClick event handlers will get called any time the user double-clicks on a ListBox. The first click selects the item, so the event handler just needs to call either `MoveCard(true)` or `MoveCard(false)`.

The KeyDown event handler is called any time the ListBox is focused and the user presses a key. We want the user to be able to use the up and down arrows to navigate the cards, so we only want the event hanlder to respond when the user presses the Enter key. Take a close look at the arguments for the KeyDown event handler:



```
private void leftDeckListBox_KeyDown(object sender,
KeyEventArgs e)
```

You should only call MoveCard if (e.Key == Key.Enter) – the KeyEventArgs class has a Key property, and its type is an enum called System.Windows.Input.Key. **Use “Go to Definition / Declaration” to explore the Key enum.**

Your Deck class extends ObservableCollection<Card> so you can bind each ListBox to an instance of Deck. You'll call its Add method to add a Card, RemoveAt to remove a Card from a specific index, and Clear to remove all cards. We've given you the method declarations and some of the code, and comments to help you finish the Deck class.

```
class Deck : ObservableCollection<Card>
{
    private static Random random = new Random();
    public Deck() { Reset(); }

    public void Reset() {
        /* Call Clear() to remove all cards from the deck, then use two for loops to add every
         * combination of suit and value, calling Add(new Card(...)) to add each card */
        throw new NotImplementedException("The Reset method restes the 52-card deck");
    }

    public Card Deal(int index) {
        // Use base[index] to pull out the specific card and RemoveAt(index) to remove it
        throw new NotImplementedException("The Deal method will deal a card from the deck");
    }

    public void Shuffle() {
        /* Use new List<Card>(this) to create a copy of the deck, then pick a random card
         * from copy, call copy.RemoveAt to remove it, and Add(card) to add it */
        throw new NotImplementedException("The Shuffle method will randomize the cards");
    }

    public void Sort() {
        List<Card> sortedCards = new List<Card>(this);
        sortedCards.Sort(new CardComparerBySuitThenValue());
        // Use a foreach loop to call Add for each card in sortedCards
        throw new NotImplementedException("The Sort method sorts the cards.");
    }
}
```

You'll need this using directive. ↴
using System.Collections.ObjectModel;

Your job is to replace the exceptions and comments with working code. ↴

List<T> has an overloaded constructor that takes an IEnumerable<T> and initializes the list with its contents. ↴

ObservableCollection doesn't have a Sort method! You'll need to add one yourself. ↴

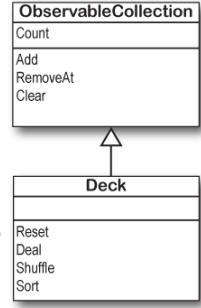
Flip back to Chapter 1 – this is really similar to the foreach loop you used to pull random emoji from a List<string>. ↴

We gave you the first two lines of the Sort method, including a call to List.Sort that takes an IComparer (just like the one you used earlier to sort Ducks). Click on CardComparerBySuitThenValue and use the Quick Fix menu to **generate the class** in a new file, then go to the declaration and use the Quick Fix menu again to

implement the `IComparer<Card>` interface. Here's the complete `CardComparerBySuitThenValue` class:

```
internal class CardComparerBySuitThenValue : IComparer<Card>
{
    public int Compare([AllowNull] Card x, [AllowNull] Card y)
    {
        if (x.Suit > y.Suit)      ↑
            return 1;
        if (x.Suit < y.Suit)      } The IDE's Quick Fix created the
            return -1;             class and method declarations,
        if (x.Value > y.Value)   so you just need to fill in the
            return 1;             method body. We'll talk about
        if (x.Value < y.Value)   [AllowNull] in Chapter 11.
            return -1;
        return 0;
    }
}
```

You'll add these methods to the `Deck` class to reset, deal, shuffle, or sort the cards in the collection.





LONG EXERCISE SOLUTION

FIXME This project is big, and it has a lot of different parts. But if you run into trouble, just take it piece by piece. None of it isмагHere's the code for the static HoneyVault class.FIXME

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

        if (Resources["rightDeck"] is Deck rightDeck)
        {
            rightDeck.Clear();
        }
    }

    private void MoveCard(bool leftToRight)
    {
        if ((Resources["rightDeck"] is Deck rightDeck) && (Resources["leftDeck"] is Deck leftDeck))
        {
            if (leftToRight)
            {
                if (leftDeckListBox.SelectedItem is Card card)
                {
                    leftDeck.Remove(card);
                    rightDeck.Add(card);
                }
            }
            else
            {
                if (rightDeckListBox.SelectedItem is Card card)
                {
                    rightDeck.Remove(card);
                    leftDeck.Add(card);
                }
            }
        }
    }

    private void leftDeckListBox_MouseDoubleClick(object sender, MouseButtonEventArgs e)
    {
        MoveCard(true);
    }

    private void rightDeckListBox_MouseDoubleClick(object sender, MouseButtonEventArgs e)
    {
        MoveCard(false);
    }
}
```

Here's where the MainWindow constructor was modified to safely get a reference to the right deck and call its Clear method.

This code safely gets a reference to the right deck. It's made up of two parts that you're familiar with: using a key to access an item in a Dictionary (in this case, the Window resource dictionary) and using the "is" keyword to safely cast it (in this case to a Deck reference).

Here's the MoveCard method that we gave you in the instructions. Take a few minutes and use the debugger to step through it, using the Locals window to keep an eye on the contents of both decks.

The ListBox' MouseDoubleClick event handlers just call the MoveCard method. The left deck ListBox calls MoveCard(true) to move the selected card from left to right, the right deck ListBox calls MoveCard(false) to move the card from right to left.

```

private void leftDeckListBox_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter)
    {
        MoveCard(true);
    }
}

private void rightDeckListBox_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter)
    {
        MoveCard(false);
    }
}

private void shuffleLeftDeck_Click(object sender, RoutedEventArgs e)
{
    if (Resources["leftDeck"] is Deck leftDeck) ←
    {
        leftDeck.Shuffle();
    }
}

private void resetLeftDeck_Click(object sender, RoutedEventArgs e)
{
    if (Resources["leftDeck"] is Deck leftDeck)
    {
        leftDeck.Reset();
    }
}

private void sortRightDeck_Click(object sender, RoutedEventArgs e)
{
    if (Resources["rightDeck"] is Deck rightDeck)
    {
        rightDeck.Sort();
    }
}

private void clearRightDeck_Click(object sender, RoutedEventArgs e)
{
    if (Resources["rightDeck"] is Deck rightDeck)
    {
        rightDeck.Clear();
    }
}

```

The KeyDown event handlers work just like the MouseDoubleClick event handlers, except they use the KeyEventArgs argument to move the card only if the user pressed the Enter key.

Each of the Button Click event handlers use the is keyword to safely get a reference to either the left or right deck.

/* Call Clear() to remove combination of suit
When we used /* and */ to create multi-line comments in the exercise instructions, we added an extra star at the beginning of each new line. It's not necessary, but it's something that a lot of developers do because it makes multi-line comments easier to read (especially if they're indented).



LONG EXERCISE SOLUTION

Here's the code for the Deck class. It extends ObservableCollection<Card> so you can bind it directly to a ListBox, and adds four additional methods to work with the cards in the collection.

```
using System.Collections.ObjectModel;
class Deck : ObservableCollection<Card>
{
    private static Random random = new Random();
    public Deck()
    {
        Reset();
    }
    public Card Deal(int index)
    {
        Card cardToDeal = base[index];
        RemoveAt(index);
        return cardToDeal;
    }
    public void Reset()
    {
        Clear();
        for (int suit = 0; suit <= 3; suit++)
            for (int value = 1; value <= 13; value++)
                Add(new Card((Values)value, (Suits)suit));
    }
    public void Shuffle()
    {
        List<Card> copy = new List<Card>(this); ← The Shuffle method randomizes the cards. The first thing it does is use create a copy of the deck using this overloaded List<T> constructor.
        Clear();
        while (copy.Count > 0)
        {
            int index = random.Next(copy.Count);
            Card card = copy[index];
            copy.RemoveAt(index);
            Add(card);
        }
    }
    public void Sort()
    {
        List<Card> sortedCards = new List<Card>(this);
        sortedCards.Sort(new CardComparerBySuitThenValue());
        Clear();
        foreach (Card card in sortedCards) ← We gave you the first two lines of the Sort method that create a copy of the . This foreach loop call Add for each card in sortedCards.
        {
            Add(card);
        }
    }
}
```

The ObservableCollection<T> class is a collection that's specially built for data binding, especially for displaying items in a ListBox. It implements the familiar IEnumerable<T> and ICollection<T> interfaces (just like List<T>), and also the interfaces that let WPF data binding know when its contents have changed. The Deck class uses inheritance to extend ObservableCollection—its methods use the inherited Add, RemoveAt, and Clear methods to update the cards in the collection.

The Deal method deals a card from a specific index in the Deck. It calls base[index] get the card, then RemoveAt(index) to remove it from the collection before returning it.

The Reset method clears the deck, then uses a nested for loop to add all 13 cards for each of the four suits, calling the Add method to add each card.

This while loop chooses a random card from the copy, removes it, and adds it back to the Deck. This is really similar to code that you wrote in the animal match game in Chapter 1.

We gave you the first two lines of the Sort method that create a copy of the . This foreach loop call Add for each card in sortedCards.

Chapter 9. LINQ and Lambdas

Get Control of your data



It's a data-driven world...it's good 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 not only lets you **query data** in a simple, intuitive way, but it lets you **group data** and **merge data from different data sources**. And once you've got the hang of wrangling your data into manageable chunks, you can use **lambda expressions** to refactor your code to make your C# code even more expressive (and impressive!).

Jimmy's a Captain Amazing super-fan...

Meet Jimmy, one of the most prolific 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.



↑
Jimmy tracked down
this rare 2005 Captain
Amazing action figure in
its original packaging.

That's right, that's the actual
Amazingmobile stunt car from the flop
Captain Amazing TV show that ran
from September through November 1973.
How'd Jimmy even get his hands on it?

...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?



Use LINQ to query your collections

In this chapter we'll learn about **LINQ** (or **Language-Integrated Query**). LINQ combines some really useful classes and methods with some powerful features built directly into C#, all created to help you work with sequences of data.

Let's use Visual Studio to do 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; ← Adding "using System.Linq;" gives your collections a bunch
    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.

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!

The screenshot shows the IntelliSense window for the variable 'numbers'. The window lists several LINQ methods: Select<>, SelectMany<>, SequenceEqual<>, Single<>, SingleOrDefault<>, Skip<>, SkipLast<>, and SkipWhile<>. Below the list are standard IntelliSense navigation buttons: a green plus sign, a black arrow, a purple question mark, and a purple downward arrow.

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.WriteLine($"{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 combine to help you work with sequences of data.

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. But they also get added to arrays, queues, stacks... in fact, they’re added to any object that extends `IEnumerable<T>`. And 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 it 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`:

```
IEnumerable<int> firstFive = numbers.Take(5);
IEnumerable<int> lastFive = numbers.TakeLast(5);
IEnumerable<int> firstAndLastFive =
    firstFive.Concat(lastFive);
```

But method chaining let 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 – it’s up to you to decide which is more readable.

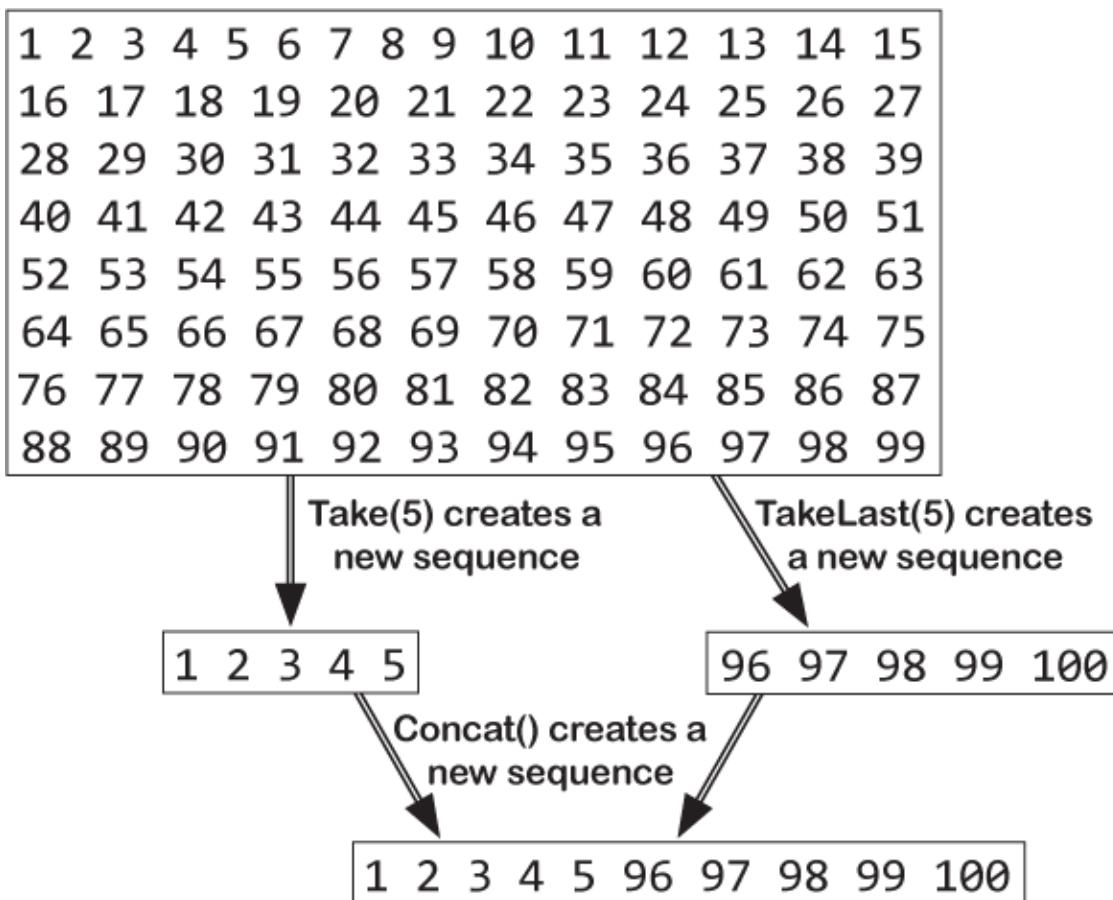
LINQ works with any `IEnumerable<T>`

When you added the `using System.Linq;` directive to your code, your `List` of numbers suddenly got “superpowered”—a bunch of LINQ methods suddenly appeared on it. And you can do the same thing for **any class that implements `IEnumerable<T>`.**

When an object implements `IEnumerable<T>` , any instance of that class is a **sequence**:

- That 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 five elements.

- When you called its `TakeLast` method, it returned another 5-element sequence.
- And when you used `Concat` to combine the two 5-element sequences, it created a new sequence with 10 elements and returned a reference to that new sequence.



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.

LINQ methods enumerate your sequences

You already know that foreach loops work with `IEnumerable` objects. Think about what does a foreach does: it starts at the first element in the sequence, then does an operation on each element in the sequence in order. 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 `IEnumerable` can be enumerated.

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.*

Here's a really useful method that comes with LINQ. The static `Enumerable.Range` method generates a sequence of integers. Calling `Enumerable.Range(8, 5)` return a sequence of 5 numbers starting with 8: 8, 9, 10, 11, 12.

`Enumerable.Range(8, 5);` → 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`

↗ Get some practice with the `Enumerable.Range` method in this next pencil-and-paper exercise.

WHAT'S MY OUTPUT?

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()`

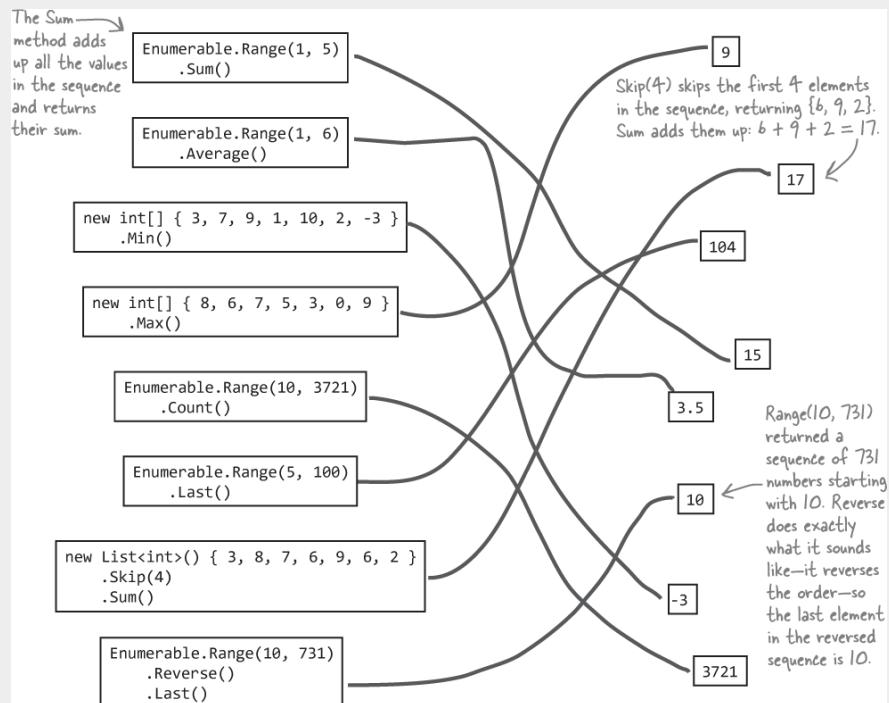
-3

`Enumerable.Range(10, 731)
.Reverse()
.Last()`

3721

WHAT'S MY OUTPUT? SOLUTION

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. And 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.

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). And 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. But 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.

Q: Can you show me an example of a class that uses method chaining?

A: Here’s a really simple class that shows how you can use method chaining in your own classes:

```
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 };
    }
}
```

And 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 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. But when us developers look at them, we see something else: **lots and lots of data** just waiting to be organized. And 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!

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 method to build 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 we 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 },           Jimmy's rare
            { 36, 650M },
            { 57, 13525M },         misprinted edition
            { 68, 250M },           of issue #57
            { 74, 75M },            ("Hippie Madness"
            { 83, 25.75M },
            { 97, 35.25M },         from 1973) is worth
                                    $13,525. Wow!
        };

```

We're using the
IReadOnlyDictionary
interface to make
encapsulate the
dictionary to
prevent code from
using the Price field
to change prices.

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?

LINQ's query syntax

So far you've seen LINQ's methods. But they're not quite 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.

Let's build a simple query now. It selects all 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, what criteria to use to determine which of its members to select, how to sort the results, and how the results should be returned.

LINQ queries work on sequences (objects that implement `IEnumerable<T>`). They start with the `from` keyword:

```
from (variable) in (sequence)
```

This tells the query what sequence to execute against.

```
int[] values = new int[] {0, 12, 44, 36, 92, 54, 13, 8};
```

```
IEnumerable<int> result =
```

This LINQ query has four clauses:
the from clause, a where clause, an orderby clause, and the select clause.

```
{ from v in values  
  where v < 37  
  orderby -v  
  select v;
```

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 it from highest to lowest.

The query ends with a select clause with an expression that tells it what to put in results.

```
// use a foreach loop to print the results
```

```
foreach(int i in result)
```

```
    Console.WriteLine("{i} ");
```

Output: 36 13 12 8 0



EXERCISE

Look closely at the LINQ query we just showed you, and use it as a template to finish your Program's Main method so it **prints a list of comics with a price > 500 in reverse order**. Make sure you include the two using declarations so you can use `IEnumerable<T>` and LINQ methods. The query will return an `IEnumerable<Comic>`—use a foreach loop to iterate through it and print this: `($"{comic} is worth {prices[comic.Issue]:c}")`—Here's what the output looks like:

```
Hippie Madness (misprinted) (Issue #57) is worth $13,525.00  
Johnny America vs. the Pinko (Issue #6) is worth $3,600.00  
Woman's Work (Issue #36) is worth $650.00
```

} Remember ":c" formats a number as local currency, so if you're in, say, the UK, you'll see £ instead of \$.



EXERCISE SOLUTION

We asked you to use the LINQ that we gave you as a template query Jimmy's comic collection. Here's the entire Program class, including the using directives that you needed to add to the top.

```
using System.Collections.Generic; ← You need System.Collections.Generic to
using System.Linq; ← use the IEnumerable<T> interface, and
← you need System.Linq to add the LINQ
← methods to any object that implements it.

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; ← The select clause determines what the
                           ← query actually returns. Since it's selecting
                           ← a Comic variable, the result of the query
                           ← is an IEnumerable<Comic>.
        foreach (Comic comic in mostExpensive)
        {
            Console.WriteLine($"{comic} is worth {Comic.Prices[comic.Issue]:c}");
        }
    }
}
```

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.

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
```



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 change that clause to use the `descending` keyword—that should make it easier to read.

- 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.

Changing the select clause causes the query to return a sequence of strings.

```

IEnumerable<string> mostExpensiveComicDescriptions =
    from comic in Comic.Catalog
    where Comic.Prices[comic.Issue] > 500
    orderby Comic.Prices[comic.Issue] descending
    select $"{comic} is worth {Comic.Prices[comic.Issue]:c}";

```

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.

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 including comics whose price is greater than 500.

The orderby clause starts with the results of the where clause and sorts it in descending order by price.

The select clause loops through the results of the orderby clause, using the 'comic' range variable with string interpolation to return a sequence of string.

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. Sometimes 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;
```

These declarations 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:

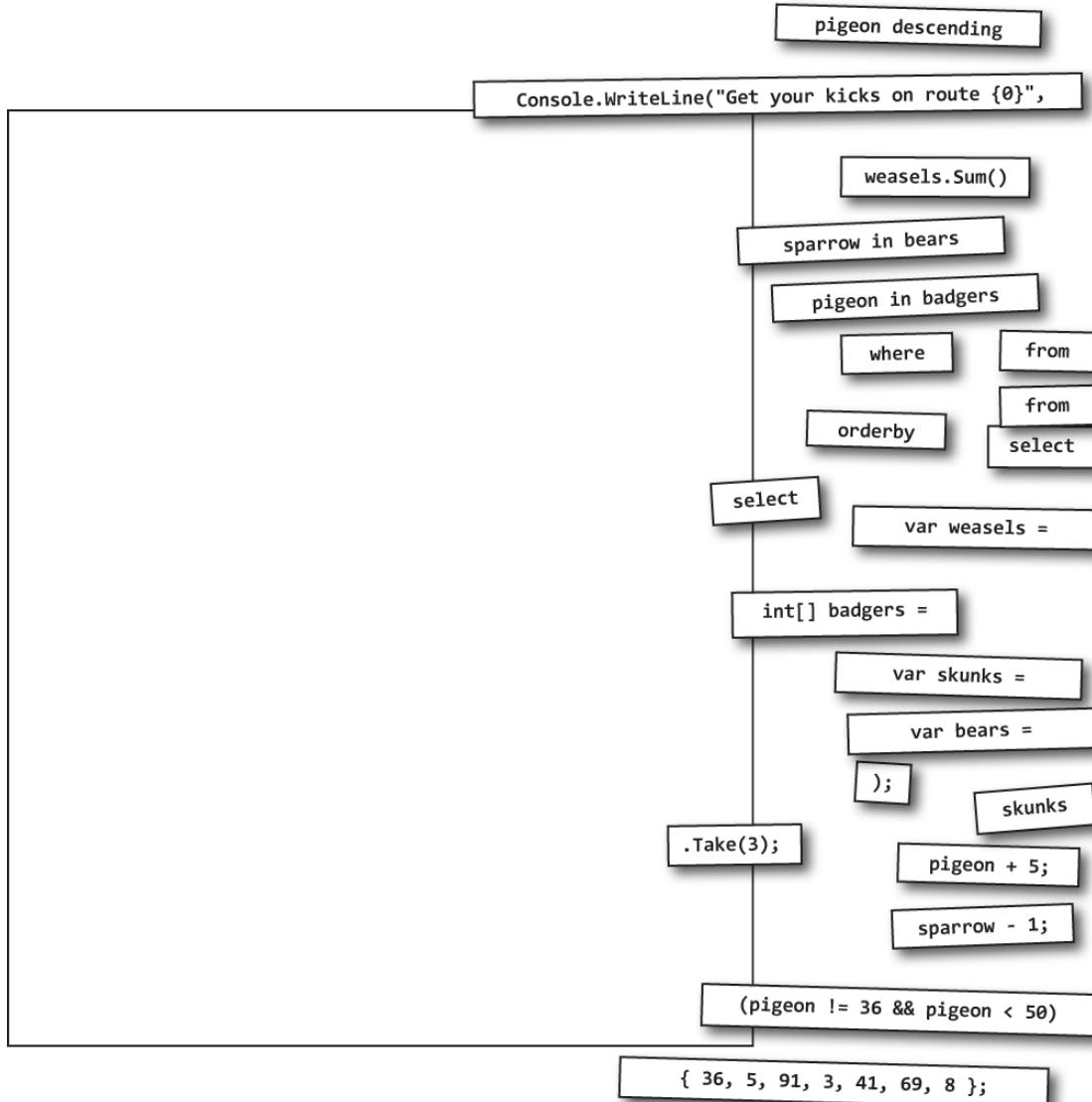
```
foreach (Comic comic in mostExpensive)  
{  
    Console.WriteLine($"{comic}");  
}
```

The IDE figured out the `mostExpensive` variable's type—and it's a type we haven't even seen before. Remember how we talked in [Chapter 7](#) 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 hover 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



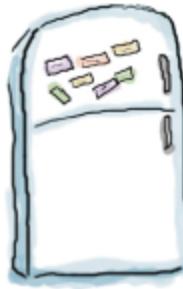
We had a nice LINQ query that used the var keyword arranged with magnets on the refrigerator—but someone slammed the door and the magnets fell off! Rearrange the magnets so they produce the output at the bottom of the page.



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 };`

"from Pigeon in badgers" makes for a good puzzle, but an unreadable LINQ query. "from Badger in badgers" is more readable.

```
var skunks =
  from pigeon in badgers
  where (pigeon != 36 && pigeon < 50)
  orderby pigeon descending
  select pigeon + 5;
```

After this statement, skunks contains four numbers: 46, 13, 10, and 8.

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



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`.

There are a couple of rules: 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, and you can't declare it equal to `null`. And if you create a variable named `var`, you won't be able

to use it as a keyword anymore. And 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. But 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. But one thing to keep in mind here is that LINQ queries are just extension methods. They call methods that do all the real work. You could call those same methods without LINQ.

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. But what if you're selecting from a list of Comic objects? You could do what Jimmy did and select the whole class. But 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 see 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. .NET detects the type from 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 it 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, or set a variable, or do a mathematical operation, or call a method...those are all single operations. For example:

```
var under10 =
  from number in sequenceOfNumbers
  where number < 10
  select number;
```

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.



BULLET POINTS

- When an object 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.
- And 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 its 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.
- 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 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 the 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

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. And 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 that we've already seen.

- **Modify every item returned from the query.**

This code will add a string onto the end of each string in an array. It doesn't change the array itself—it creates a new sequence of modified strings.

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() };
```

```
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 that 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}");
```

The static `String.Join` method concatenates all of the items in a sequence into a string, specifying the separator to use between them.

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.

```
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

Do this!

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}");
    }
}
```

The `Console.WriteLine` in the getter isn't called until the `foreach` loop actually executes. That's what deferred execution looks like.

Did you get a weird compiler error? Make sure you added the two using directives to your code!

Set up the query
Run the foreach
Getting #1
Writing #1
Getting #2
Writing #2
Getting #3
Writing #3
Getting #4
Writing #4

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. One easy way is to call its `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}");
```

Now run the app. 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 see immediate execution as well.

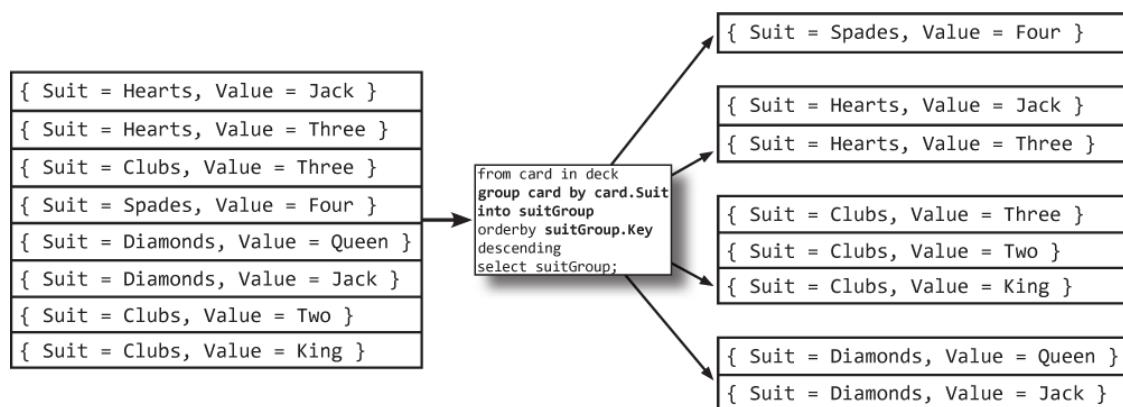
When you call `ToList` or another LINQ method that needs to access every element in the sequence, you'll get immediate evaluation.

```
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'd want to group your data together. And that's where the **LINQ group query** comes in handy.

Group this!



1. Create a new console app and 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 CardComparerBySuitThenValue 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.

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, and the IDE changes its name everywhere it occurs in the code.

2. Make your card comparable.

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 CardComparerBySuitThenValue().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. So all you need to do to make it support method chaining is to 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 an
                           instance to the Deck, you can call it and then
                           chain additional method calls to the result.
}
```

4. Use a LINQ query with group 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 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()}");
        }
    }
}
```

Now that the Shuffle method supports method chaining, you can chain the LINQ Take method right after it.

Use LINQ's Count, Min, and Max methods to get information about each group the query returns.

Hover over the "grouped" variable to see its type.

The group clause of a LINQ query separates a sequence into groups:

group card by card.Suit into suitGroup

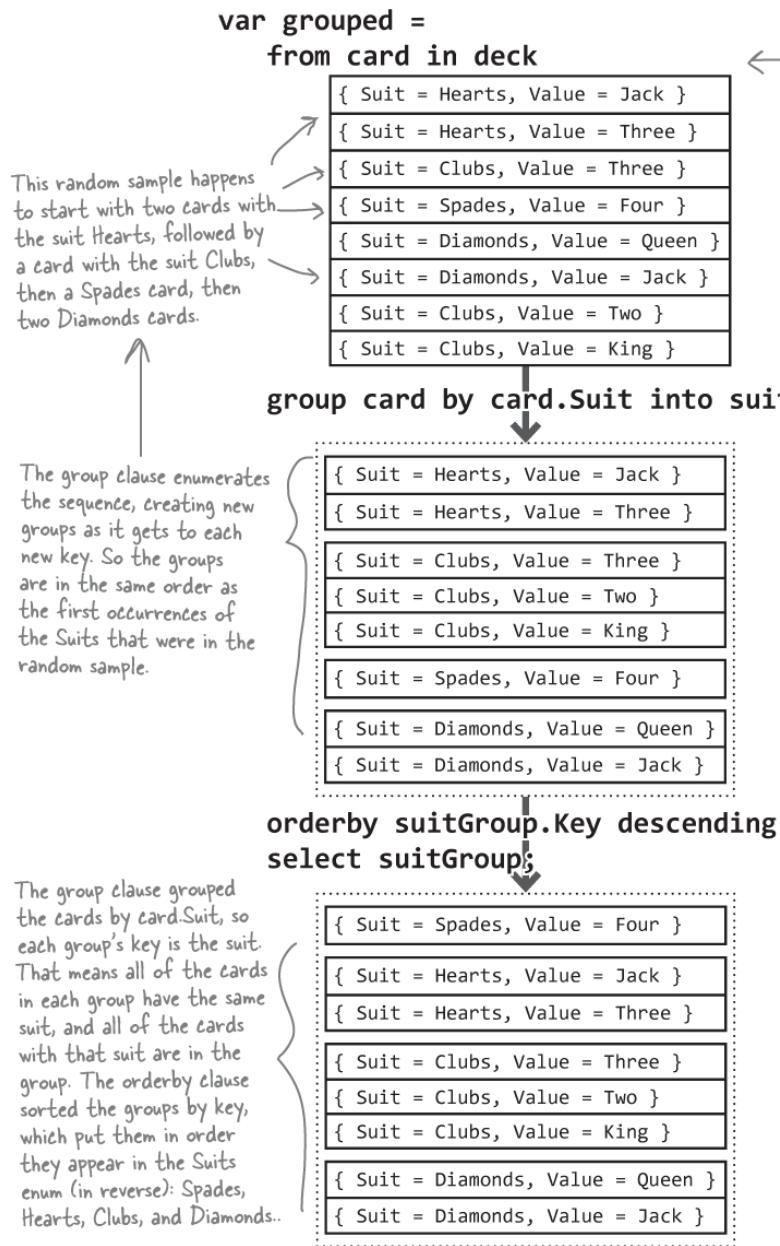
The group keyword tells it which sequence contains the elements to group, the by keyword specifies the criteria used to determine the groups, and the into keyword declares a new variable that the other clauses can use to refer to the groups. The output of a group query is a sequence of sequences.

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.

Anatomy of a group query

Let's take a closer look at how that group query works.





The from clause works just like 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 clause splits the cards into groups. It includes 'by card.Suit' - that specifies that the key for each group is the card's suit. And it declares a new variable called suitGroup that the remaining clauses can use to work with the groups.

The group 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 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 sources** 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.

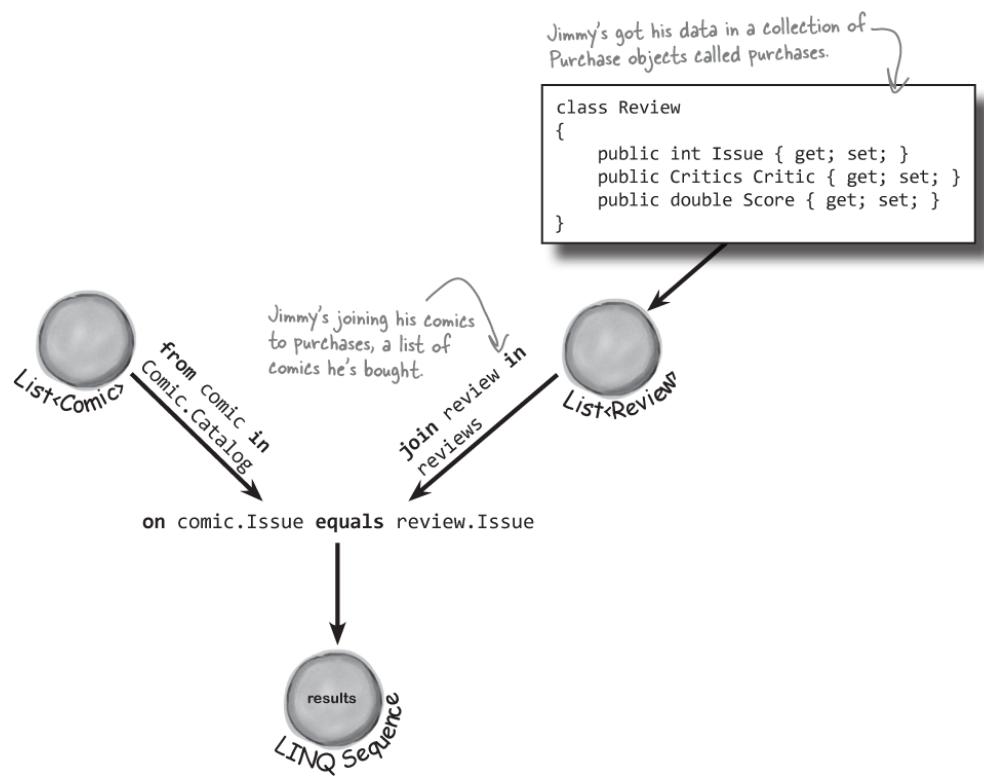
1. Start off your query with the usual `from` clause. But instead of following it up with the criteria to determine what goes into the results, you add:

```
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.

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.

3. You'll continue the LINQ query with `where` and `orderby` clauses as usual. You could finish it with a normal `select` clause, but you usually want to return results that pull some data from one collection and other data from the other.



```
{
  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 the Comic, but Critic and Score properties from the Review. But it 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. ***So what's the type of the sequence generated by the query?***

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`). But 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. 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 = Orange, Flavor = Tasty, Height = 37 }
```

Now hover over whatAmI in the IDE and have a look at the IntelliSense window:

whatAmI

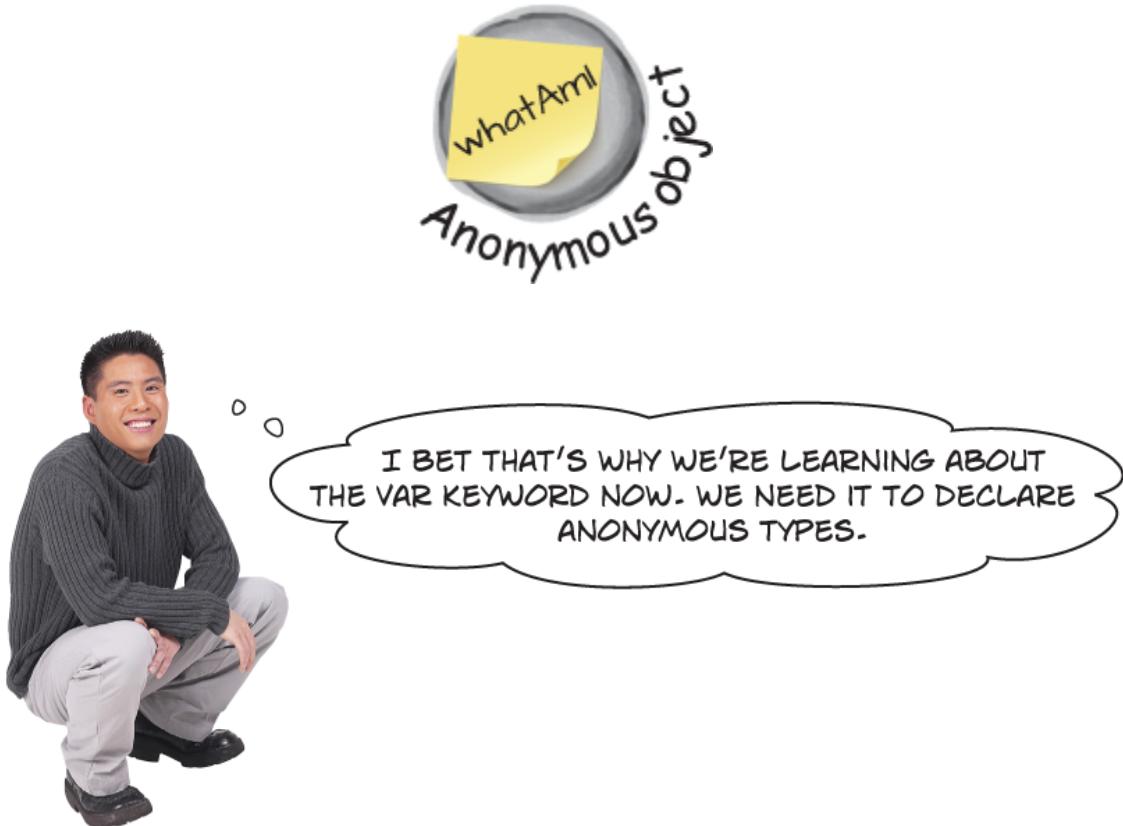
⌚ (local variable) 'a whatAmI
Anonymous Types:
'a is new { string Color, string Flavor, int Height }

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 whoAmI 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, its three properties.

```
Console.WriteLine($"My color is {whatAmI.Color} and  
I'm {whatAmI.Flavor}");
```

Besides the fact that they don't have names, anonymous type are just like any other types.





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[] ← We're using var and
{
    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);

{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 2, Winnings = 2 }
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 4, Winnings = 1.3 }
```

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. And 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.



SHARPEN YOUR PENCIL SOLUTION

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 returns 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! Sure, 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.

And 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."

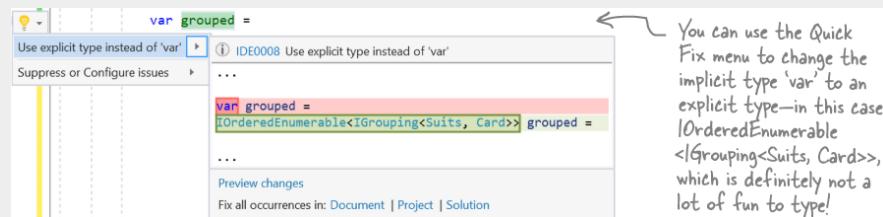
IDE TIP: REFACTORING TOOLS

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 Fix 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 Fix menu to change the variable back to `var`.

Extract methods

Your code can 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. And the IDE gives you a really useful refactoring tool to make that easy.

A screenshot of the Visual Studio code editor showing a block of code highlighted with a blue selection bar. The code is part of a `Main` method:static void Main(string[] args)
{
 var deck = new Deck().Shuffle();

 var grouped =
 from card in deck
 group card by card.Suit into suitGroup
 orderby suitGroup.Key descending
 select suitGroup;The line `var grouped =` is specifically highlighted with a yellow lightbulb icon, indicating it's the target for extraction.

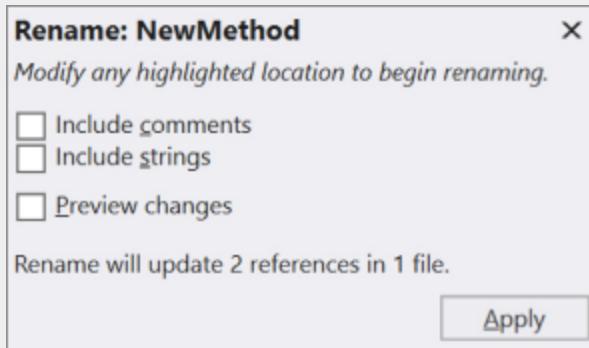
Start by selecting a block of code.

Then **choose Refactor >> Extract Method** from the Edit menu (on Windows) or **Extract Method** from the Quick Fix menu (on Mac).

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 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 we need to get their shirt size? If we 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 his shirt size, and I don't care about the number at all?

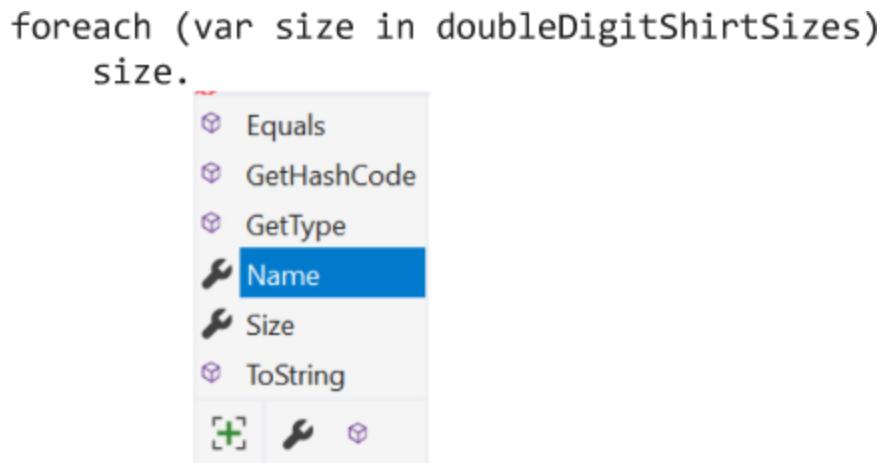
A: That's what **anonymous types** are for—you can construct an anonymous type that only has the data you want in it. And it lets you pick and choose from the various collections that you're joining together, too.

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.



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. And 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 use to figure out the type of the variable. But 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 write a method that returns an anonymous type, or which takes one as a parameter, or use one with a field or a property.



EXERCISE

Use what you've earned 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 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]}");  
    }  
    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 GroupComicsByPrice and GetReviews methods call methods in the static ComicAnalyzer class (which you'll write) that run LINQ queries.

The foreach loops in GroupComicsByPrice are nested: one loop is inside the other. The outer loop prints information about each group, and the inner one enumerates the group.

Your job is to **create a static class called ComicAnalyzer** with three static methods (two of which are public):

- Private static method called CalculatePriceRange takes a Comic reference and returns a new enum 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 the 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 string like this:

```
MuddyCritic rated #74 'Black Monday' 84.20
```



EXERCISE SOLUTION

Use what you've earned 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.

First you added these Critics and PriceRange enums.

```
enum Critics {
    MuddyCritic,
    RottenTornadoes,
}

enum PriceRange {
    Cheap,
    Expensive,
}
```

Then you added the Review class.

```
class Review {
    public int Issue { get; set; }
    public Critics Critic { get; set; }
    public double Score { get; set; }
}
```

Once you had those, you could add the static ComicAnalyzer class with a private PriceRange method and public GroupComicsByPrice and GetReviews methods.

```

using System.Collections.Generic; } Don't forget the
using System.Linq;           ↗ using directives.

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)
    {
        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}}";
    }
}

```

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 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 that 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** 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, 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 just doesn't have a name. The members specified in the new statement become automatic properties on 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 Fix menu to change a var declaration to an **explicit type**, or back to an 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.



GAME DESIGN... AND BEYOND TESTING

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 be 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:** Just 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 game play 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 **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 user 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 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, they can be automated or manual and the goal of this kind of testing is 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 it. Those bugs get written down with steps 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 as they can of their testing and run as many of those tests as possible with each commit. That way, they know if they inadvertently introduced a bug when they were making a fix or adding a new feature.

Unit tests help you make sure your code works

You've sleuthed out a lot of bugs in the first 8 chapters of this book, so you know how easy it is to write code that doesn't do exactly what you intended it to do. Luckily, there's a way for us to help 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. And most large programs have a **suite** of tests that cover most or all of the code.

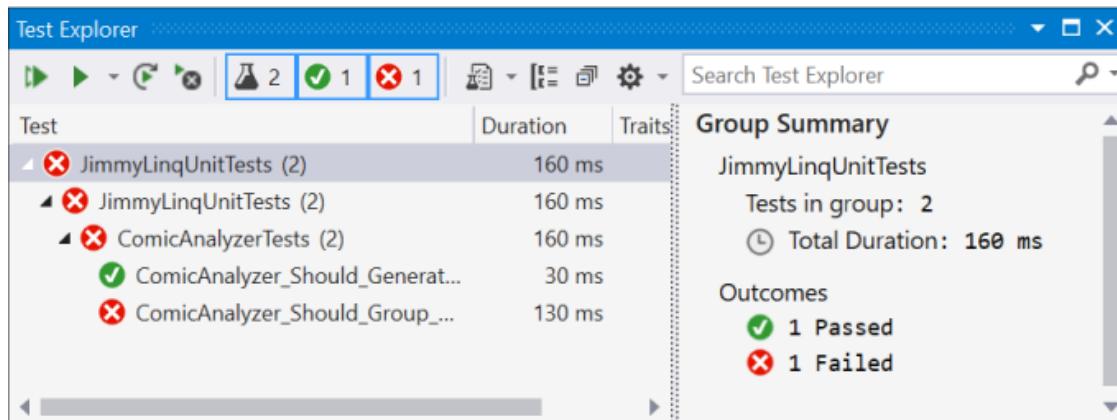
Visual Studio has built-in unit testing tools to help you write your tests and track which ones pass or fail. The unit tests in

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.

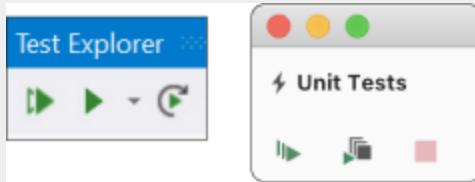
Visual Studio also supports unit tests written in NUnit and xUnit, two popular open source unit test frameworks for C# and .NET code.

Visual Studio for Windows has the Test Explorer window

Open the Test Explorer window by choosing **View > Test Explorer** from the menu. 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.

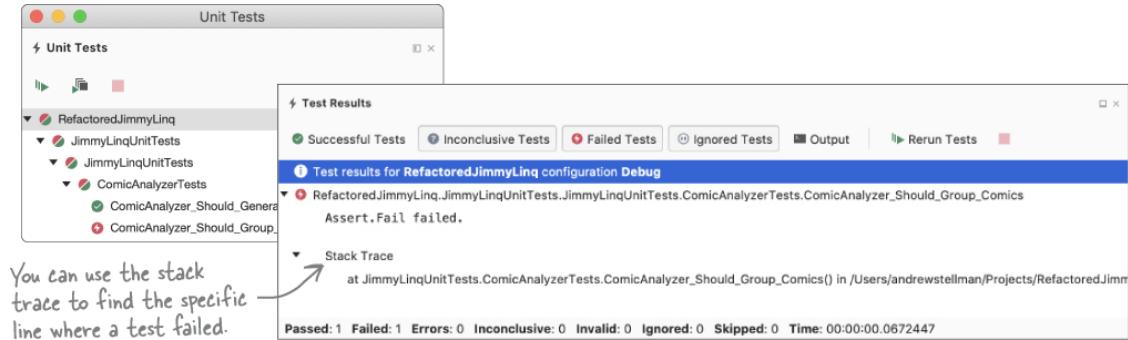


When you add unit tests to your solution, you can run your tests by clicking the Run All Tests button. You can debug your unit tests on Windows by choosing Tests >> Debug all tests, and on Mac by clicking Debug all tests in the Unit Test pad.



Visual Studio for Mac has the Unit Test pad

Open the Unit Test pad by choosing **View >> Pads >> Unit Tests** from the menu. It has buttons to run or debug your tests. When you run the unit tests, the IDE displays the results in a Test Results pad (usually at the bottom of the IDE window).



Add a unit test project to your solution

1. Add a new MS Test (.NET Core) project.

Right-click on the solution name in the Solution Explorer, then choose **Add >> New Project...** from

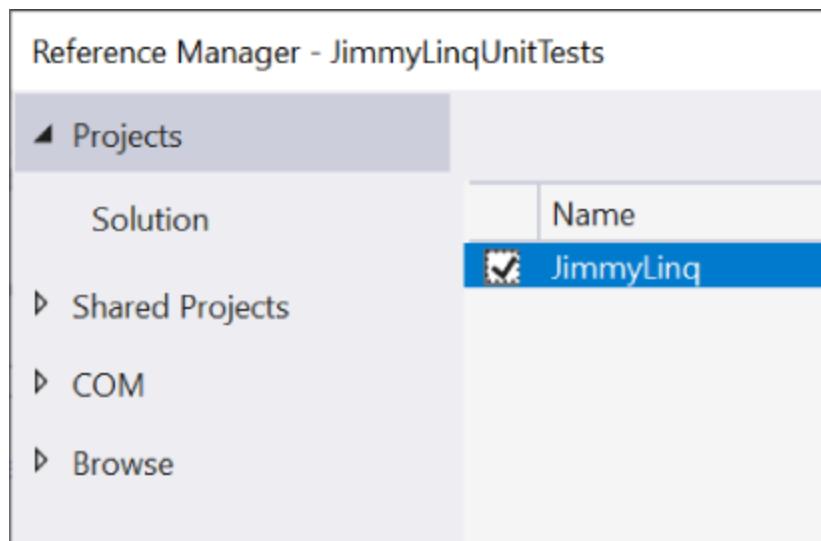
the menu. Name your project *JimmyLinqUnitTests*.



2. Add a dependency on your existing project.

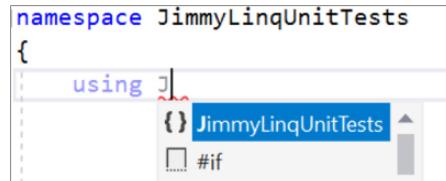
You'll be building unit tests for the `ComicAnalyzer` class. When you have two different projects in the same solution, they're *independent*—by default, the classes in one project can't use classes in another project—so we'll need to set up a dependency to let your unit tests use `ComicAnalyzer`.

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.cs. Edit that file and try adding the `using JimmyLinq;` directive inside the namespace:



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, and it's a way C# has to tag classes and methods. A test class includes test methods, which must be marked with the `[TestClass]` 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 simple 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.

Now run your test by choosing **Test >> Run All Tests** (Windows) or **Run >> Run Unit Tests** (Mac) from the menu. The IDE will pop up a Test Explorer window (Windows) or Test Results panel (Mac) 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 this icon in Windows:  or this message at the bottom of the IDE window in Visual Studio for Mac: **Failed: 1** – 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 theorise before one has data." So let's get some data.

Let's start with the assertions

And 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`. And 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 line of code threw the exception—so 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 when 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!**

And 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

And now run your test again. This time it passes!



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. But the `GetReivews` 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()
```

```

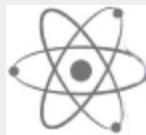
{
    var testReviews = new[]
    {
        new Review() { Issue = 1, Critic =
Critics.MuddyCritic, Score = 14.5},
        new Review() { Issue = 1, Critic =
Critics.RottenTornadoes, Score = 59.93},
        new Review() { Issue = 2, Critic =
Critics.MuddyCritic, Score = 40.3},
        new Review() { 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.



BRAIN POWER

What happens if you pass ComicAnalyzer. GetReviews a sequence with duplicate reviews? 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. But what if we get some weird review scores—like negative ones, or really big ones, or zero? And 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 that truncates the score to two decimal places. We happen to know that it rounds that value up.

Adding unit tests that handle edge cases and weird data 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 that you create your program from. Different languages have different units; in C# the basic unit of code is a class.

So from that perspective, the name “unit testing” 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. But 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. But you can add multiple Console App projects, WPF projects, 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. And 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—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 okay if some of them fail?

A: Yes! Think of it this way—is it okay 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 in 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.

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? Would you 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** 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.



Yes! You can use lambda expressions to refactor many methods and properties.

You've written a lot of methods and throughout this book that just contain 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—it prints output each time with a different random number: The value is 37.8709



Now **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}");
```

These lambdas each take a parameter, just like the methods they're replacing.

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();
```

(field) static Random Program.random

This isn't a true refactoring, because we changed the behavior of the code, not just its structure.



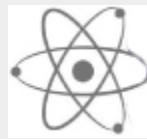
Now **modify the random field** to use a lambda expression:

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

The program still runs the same way. But **hover over the random field again**:

```
static Random random => new Random();  
    ↴  
    ⚡ 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. But 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**.



BRAIN POWER

Which of the versions of this code do you think is easiest to read?



Refactor a clown with lambdas

Do this!

Back in Chapter 7, you created 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 **Implement interface** option in the Quick Fix menu tells the IDE to add any missing interface members.

So 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 Fix 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!");
```

The IDE created a method body when it added the interface members, but you can replace it with a lambda expression.

The FunnyThingIHave property and Honk methods work exactly like they did in [Chapter 7](#). Flip back and find the Main

method you used before—your new implementation will work exactly the same way. But now that we've refactored them to use lambda expressions, they're much more compact.

We think the new and improved TallGuy class is easier to read. Do you?





SHARPEN YOUR PENCIL SOLUTION

Here are the NectarCollector from the Beehive Management System project in [Chapter 6](#) and the ScaryScary class from [Chapter 7](#). Your job is to **refactor as 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:



SHARPEN YOUR PENCIL

Here are the NectarCollector from the Beehive Management System project in [Chapter 6](#) and the ScaryScary class from [Chapter 7](#). Your job is to **refactor as 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 ScaryThingIHave property as a lambda:

```
public string ScaryThingIHave { .get => $"{scaryThingCount} spiders"; }
```

Refactor the ScareLittleChildren method as a lambda:

```
public void ScareLittleChildren() => Console.WriteLine($"Boo! Gotcha! Look at my {ScaryThingIHave}");
```

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 only 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 it to this:

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

the C# compiler no longer saw a field. Instead, it saw a property. And this makes sense—we 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—and it by showing you that it has a get accessor { get; }.

You can use the => operator to turn a field into 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. So 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 MachineGun class in Chapter 5:

```

public void Reload()
{
    if (bullets > MAGAZINE_SIZE)
        BulletsLoaded = MAGAZINE_SIZE;
    else
        BulletsLoaded = bullets;
}

```

The condition (bullets > MAGAZINE_SIZE) returns executes the then-statement (BulletsLoaded = MAGAZINE_SIZE) if it's true, or the else-statement (BulletsLoaded = bullets) if it's not.

We converted this to use ?: by having both the consequent and alternative return a value, and used that to set the BulletsLoaded property.

We can rewrite that as a lambda expression that does exactly the same thing:

```
public void Reload() => BulletsLoaded = bullets > MAGAZINE_SIZE ? MAGAZINE_SIZE : bullets;
```

Notice the slight change—in the if/else version, the BulletsLoaded property was set inside the then- and else-statements. We changed this to use a conditional operator that checked bullets against MAGAZINE_SIZE and returned the correct value, and use that return value to set the BulletsLoaded 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 about that 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 simple 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> IEnumerable<int>.Select<int, int>(Func<int, int> selector)
```

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.

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>`

We'll learn a lot more about `Func` (and its friend, `Action`) in Chapter 13. But for now, it means that there's a LINQ method

called `Select` that takes a `Func<int, int>` parameter. So what does that mean?

Use lambda expressions with methods that take a `Func` parameter

When a method that takes a `Func<int, int>` parameter, you can **call it with a lambda expression** that takes an `int` parameter returns an `int`. 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.



RELAX

Any time you see a Func in a LINQ method, it just means you can use a lambda.

We'll learn a lot more about Func in Chapter 13. For now, when you see 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;
```

The OrderBy LINQ method sorts a sequence

Hover over the **orderby** keyword and take a look at its parameter:

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.

⌚ (extension) `IOrderedEnumerable<int> IEnumerable<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 `Where` LINQ method pulls out a subset of a sequence

Now hover over the **where** keyword in the LINQ query:

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.

⌚ (extension) `IEnumerable<int> IEnumerable<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.



MINI EXERCISE

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 mini exercise solution—the LINQ query could be refactored like this:

```
var result = values.Where(v => v < 37).OrderBy(v => -v); ←—————
```



MiNi Exercise solution

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. So you c

```
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;
```

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

And then another lambda to order the groups by key: group => group.Key

(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.

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.

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. Here's how they work:

```
var returnValue = valueToCheck switch ← A switch expression starts
{ with a value to check followed
    pattern1 => returnValue1, by the switch keyword.
    pattern2 => returnValue2,
    ...
    _ => defaultReturnValue, } The body of the switch expression
} is a series of arms that use the =>
operator to check valueToCheck and
return a value if it matches a pattern.
```

```
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.

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:

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 6 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);
        }
    }
}
```

This lambda expression takes two parameters, a Suit and an int, and returns an interpolated string.

These LINQ methods are just like the ones you saw at the beginning of the chapter.

You can use OrderByDescending because you made your Card class implement IComparable<Card> earlier in the chapter.

The Select uses a switch expression to check the card's value and generate a string.

Write down the output of the program, then add the code to a console app to check if your solution is correct:

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

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 what how 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.

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. But 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. And 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 ever fails, you know you broke something, so go fix it.

When you see a test fail, it's tempting to think the test itself is broken. And sometimes it is! But it's more likely that you found a problem in the code being tested—especially while you're refactoring.

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,
             Lambda expression that takes two parameters and returns the 'select' output);
```

This lambda is passed every 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. And here's a hint—the last parameter's lambda starts like this:

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 arms. You can test it by running the app—if you can press the correct key and see the right output, you're done. When you see a test fail, it's tempting to think the test itself is broken. And sometimes it is! But it's more likely that you found a problem in the code being tested—especially while you're refactoring.



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'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 set the same variable (done) to a value.

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;
}
```

Compare the OrderBy and GroupBy lambdas with the orderby and group clauses in the LINQ query. They're almost identical.

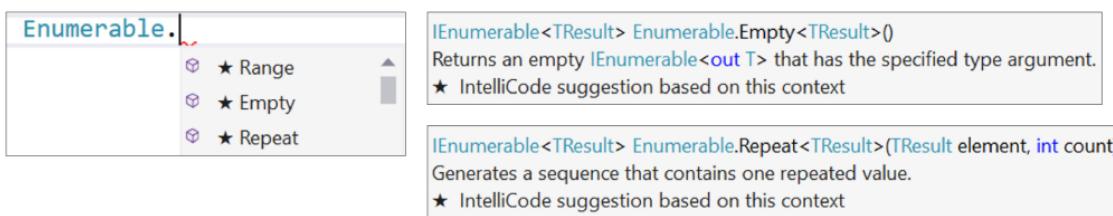
The join query starts "join reviews" so the first argument passed to the Join method is reviews.

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 pair matched comic and review from the two joined sequences and returns the string to include in the output.

Explore the Enumerable class

Before we finish this chapter, let's take a closer look at enumerable sequences. 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.



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). When you do, the **Enumerable.Empty method** comes in handy.

```
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

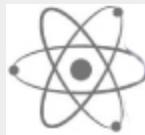
Let's say you need a sequence of 100 3's, or 12 "yes" strings, or 83 identical anonymous objects. You'd be surprised at how

often that happens! And that's what the **Enumerable.Repeat** method does—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. But we haven't really answered the question of what an enumerable sequence *actually is*. So let's finish the chapter by building some sequences ourselves. But before we do, take a minute or two to ponder the question of how you would design the `IEnumerable<T>` interface if we asked you to.



BRAIN POWER

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 just for the sake of exploring how sequences work, let's 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 Fix menu item, it used these fully qualified class names for `IEnumerator` and `IEnumerable`.

Okay, 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 IEnumarator<Sport>:

```
using System.Collections.Generic;
class ManualSportEnumerator : IEnumarator<Sport> {
    int current = -1;
    public Sport Current { get { return (Sport)current; } }
    public void Dispose() { return; } // We'll learn the Dispose method in Chapter 10
    object System.Collections.IEnumarator.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 we'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 IEnumarable. 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. And a good way to understand it is to see a simple example. And let's use a multi-project 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 creating 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);  
}
```

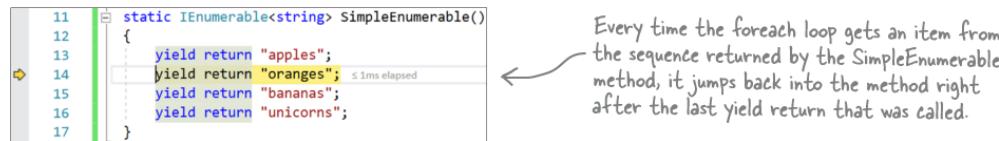
This method returns an `IEnumerable<string>` so every `yield return` returns a string value.

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.



- 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 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 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> GetGetEnumerator() {
        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 GetGetEnumerator();
    }
}
```

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

We've seen that you can use `yield return` in a method to create an `IEnumerable< T >`, and you can also use 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. And 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).



WATCH IT!

Sequences are not collections.

Try creating a class that implements `ICollection<int>` and use the Quick Fix 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
```

Here are a couple of methods from the static `System.Math` class that you'll used in your app.

- Use `Math.Pow(power, 2)` to compute a specific power of 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. And 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. But when I used `yield return`, How was I was 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

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.

```
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:

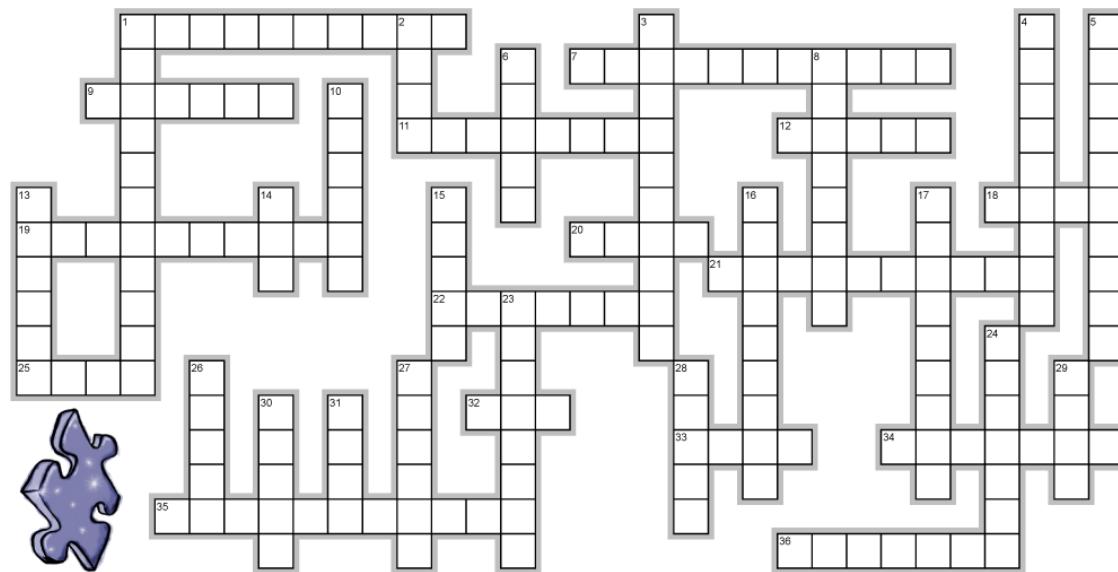
```
using System;
using System.Linq;
using System.Collections;
using System.Collections.Generic;
```



BULLET POINTS

- **Unit tests** are automated tests that help you make sure your code does what it's supposed to do.
- 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 multi-project 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 Fix menu tells the IDE to add any missing interface members.
- 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.
- orderby and where clauses in LINQ queries can be rewritten using the **OrderBy and Where** LINQ methods.
- 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.
- Unit tests help you **safely refactor** your code.
- 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



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

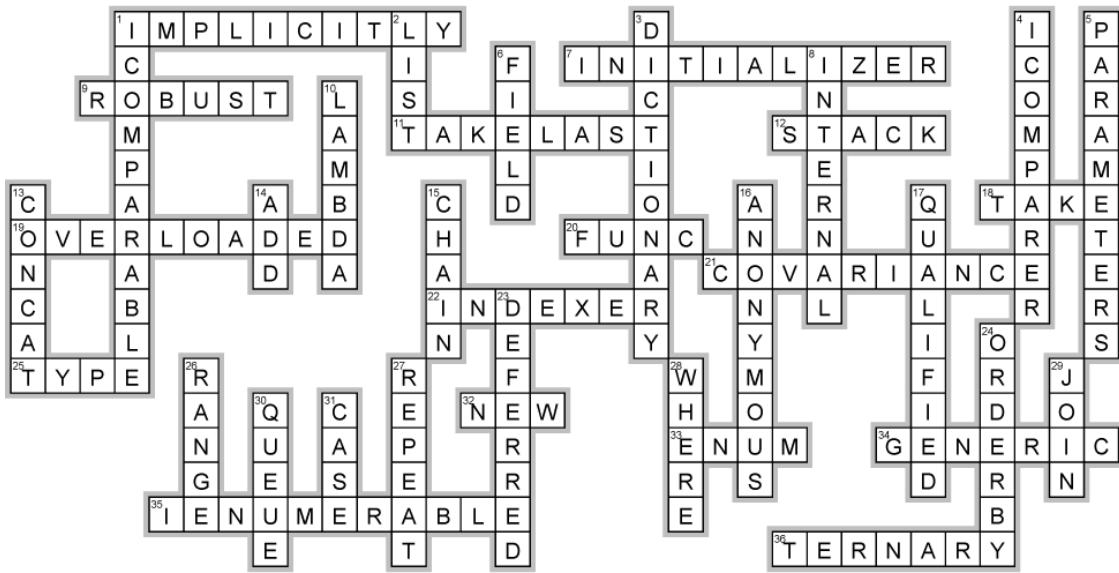
Down

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 multi-project 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

Collectioncross solution





**CAPTAIN AMAZING
WILL RETURN
IN--**



THE DEATH OF THE OBJECT

Chapter 10. Reading and writing files

Save the last byte for me!



Sometimes it pays to be a little persistent.

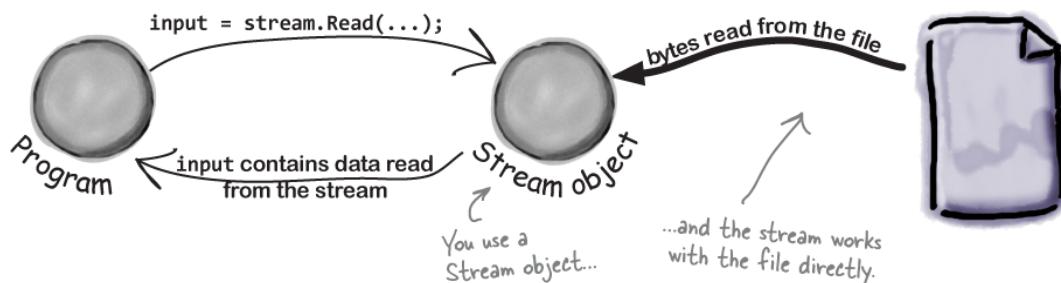
So far, all of your programs have been pretty short-lived. They fire up, run for a while, and shut down. But that's not always enough, especially when you're dealing with important information. You need to be able to **save your work**. In this chapter, we'll look at how to **write data to a file**,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

.NET uses streams to read and write data

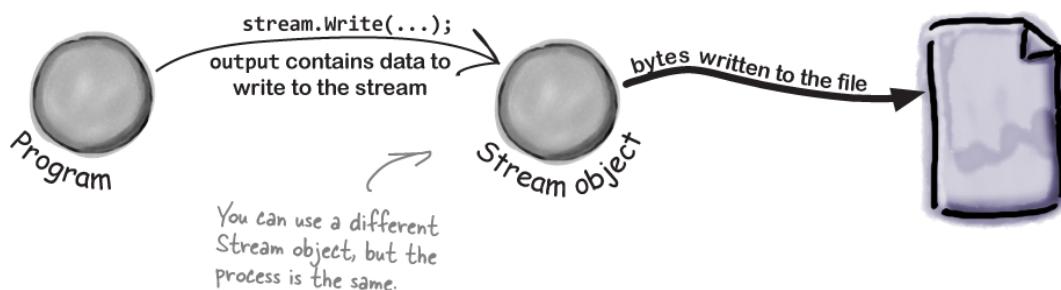
A **stream** is the .NET Framework's way of getting data in and out of your program. Any time your program reads or writes a file, connects to another computer over a network, or generally does anything where it **sends or receives bytes** from one place to another, you're using streams. Sometimes you're using streams directly. But even when you're using classes that don't directly expose streams, under the hood they're almost always using streams.

Whenever you want to read data from a file or write data to a file, you'll use a Stream object.

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.



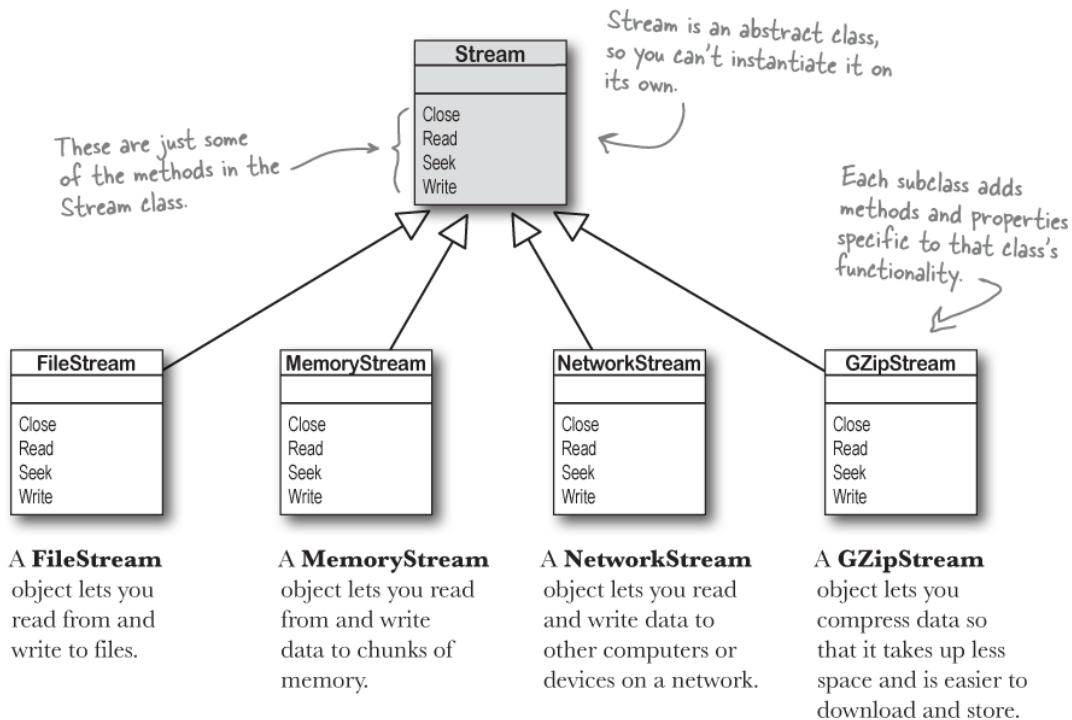
And if your app needs to write data out to the file, it can use another **Stream** object.



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. But 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 to 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:

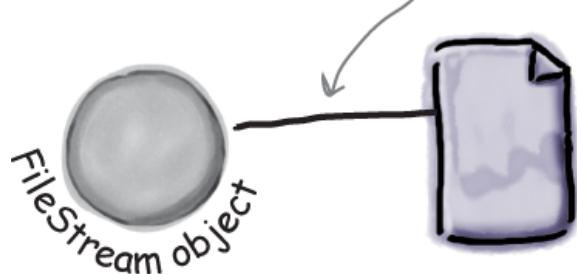
1. Create a new `FileStream` object and tell it to write to the file.

Make sure you add `using System.IO;` to any program that uses `FileStreams`.



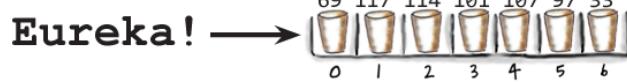
2. The `FileStream` attaches itself to a file.

A `FileStream` can only be attached to one file at a time.

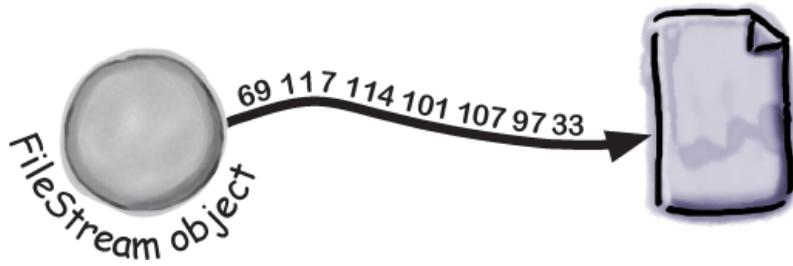


3. Streams write bytes to files, so you'll need to convert the string that you want to write to an array of bytes.

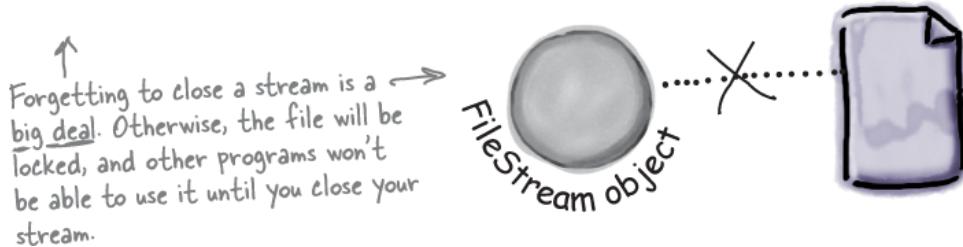
This is called encoding, and we'll talk more about it later on...



4. Call the stream's `Write` method and pass it the byte array.



5. Close the stream so other programs can access the file.



Write text to a file in three simple steps

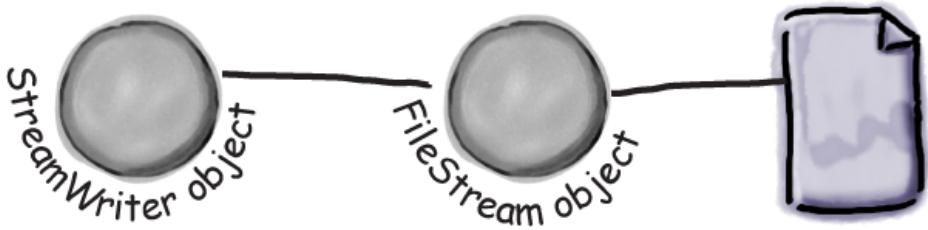
C# comes with a convenient class called **StreamWriter** that does all of those things in one easy step. All you have to do is create a new StreamWriter object and give it a filename. It **automatically** creates a FileStream and opens the file. Then you can use the StreamWriter's Write and WriteLine 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.

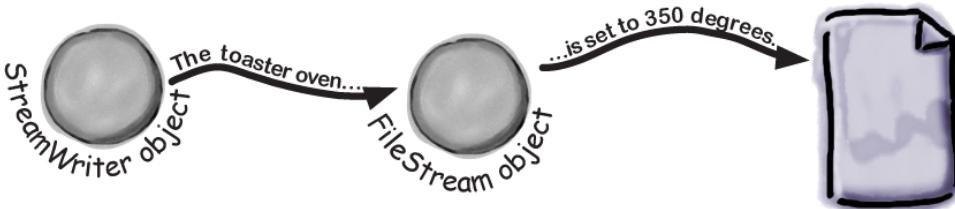
```
var writer = new StreamWriter("toaster
oven.txt", true);
```



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 open 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 arch-nemesis. 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 Application and **add this Main code**, starting with a using declaration because StreamWriter is in the **System.IO namespace**.

StreamWriter's Write and WriteLine methods work just like Console: Write writes text, and WriteLine writes text with a line break. And both classes support {curly brackets} like this:

```
sw.WriteLine("Clone #{0} attacks {1}",  
            number, location);
```

When you include {0} in the text, it's replaced by the first parameter after the string; {1} is replaced by the second, {2} by the third, etc.

```
using System.IO;           ← StreamWriter is in the  
class Program             System.IO namespace  
{  
    static void Main(string[] args)  
    {  
        This line      ← creates the  
        creates the   ← StreamWriter object and tells  
        StreamWriter  it where the  
        object and    file will be.  
        ← See if you can figure out what's  
        ← going on with the location variable  
        ← and the ?: ternary operator.  
        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();          ← 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!).
```

Here's the output of the app. Since you didn't include a full path in the filename, it wrote the 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.

Here's the output that it writes to secret_plan.txt:

Output

```
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
```

StreamWriter Magnets

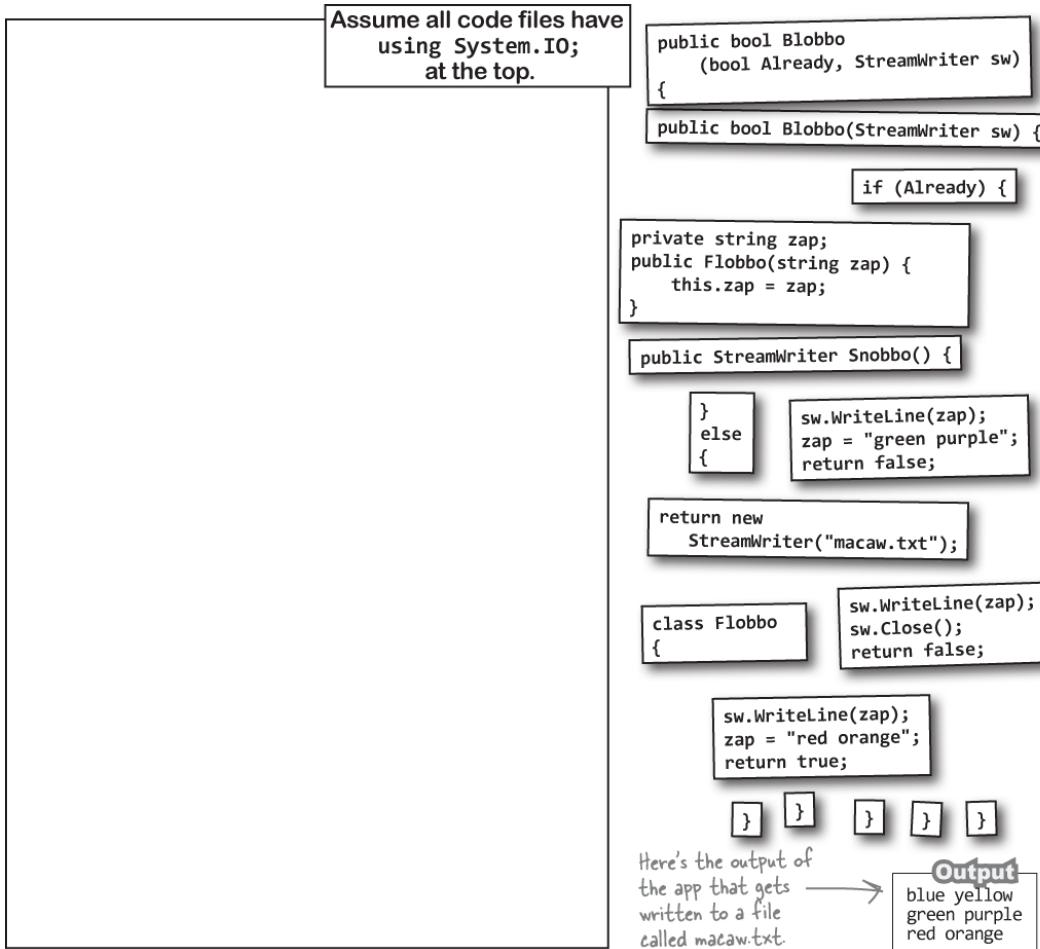


Oops! These magnets were nicely arranged on the fridge with the code for the Flobbo class, but someone slammed the door and they all fell off. Can you rearrange them so the Main method produces the output below?

```
static void Main(string[] args) {  
    Flobbo f = new Flobbo("blue yellow");  
    StreamWriter sw = f.Snobbo();  
    f.Blobbo(f.Blobbo(sw), sw), sw);  
}
```

We added an extra challenge.

Something weird is going on with the Blobbo method. See how it has two different declarations in these 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.

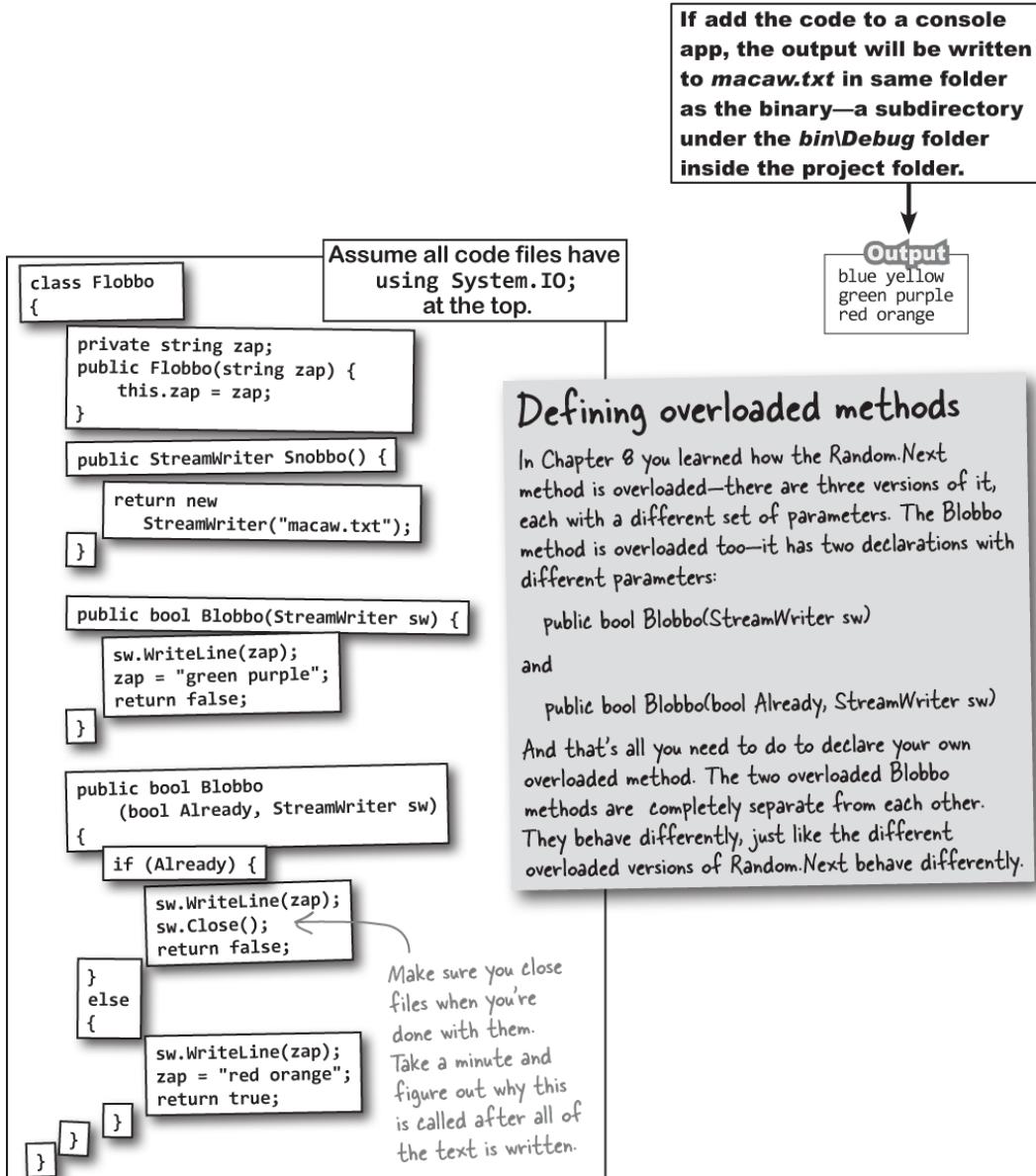


StreamWriter Magnets Solution



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(f.Blobbo(sw), sw), sw);
}
```



Just a reminder: we picked intentionally weird variable names and methods in these puzzles because if we used really good names, the puzzle would be too easy! Don't use names like this in your code, OK?

Use a StreamReader to read a file

Let's read Swindler's secret plans with a **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.

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.

```
using System.IO;
class Program
{
    static void Main(string[] args)
    {
        var folder = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
        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.

        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.
    }
}
To: CaptainAmazing@objectville.net
From: Commissioner@objectville.net
Subject: Can you save the day... again?

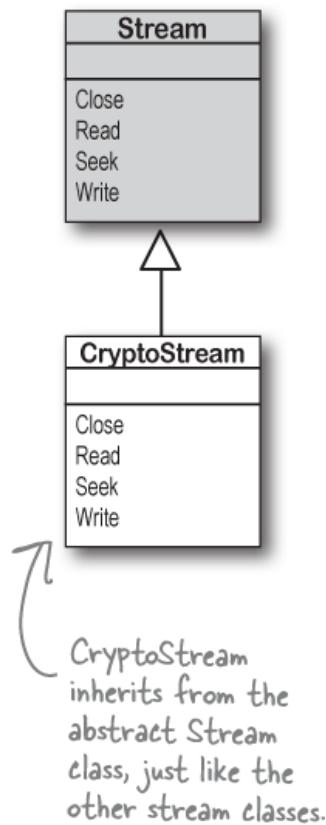
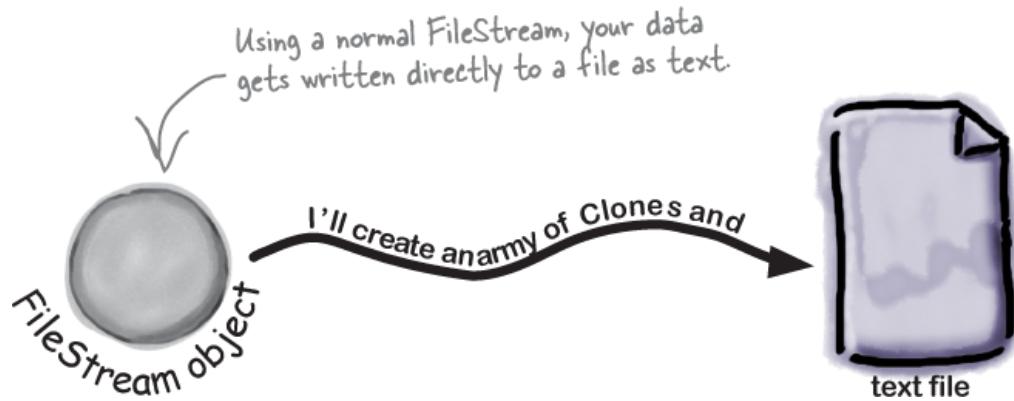
We've discovered the Swindler's terrible plan:
The plan -> How I'll defeat Captain Amazing
The plan -> Another genius secret plan by The Swindler
The plan -> I'll unleash my army of clones upon the citizens of Objectville.
The plan -> Clone #1 attacks the mall
The plan -> Clone #2 attacks downtown
The plan -> Clone #3 attacks the mall
The plan -> Clone #4 attacks downtown
The plan -> Clone #5 attacks the mall
Can you help us?
```

Output

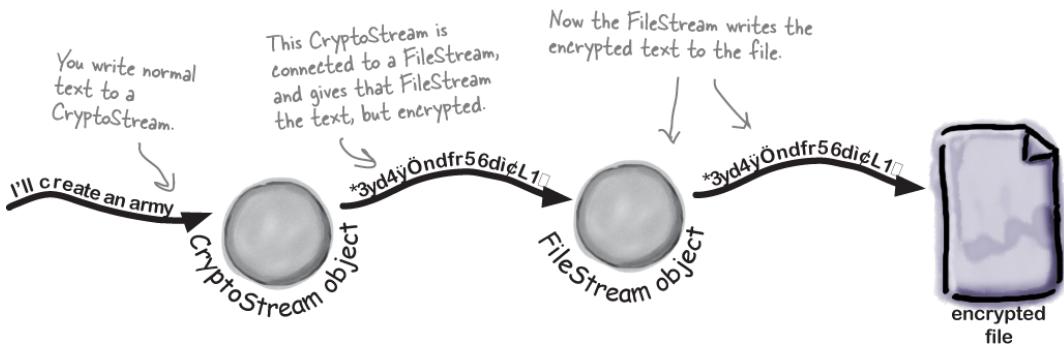
Data can go through more than one stream

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.



The Swindler can **chain streams together** and send the text through a CryptoStream object before writing its output to a FileStream.

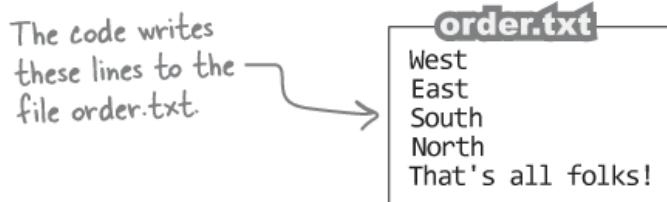


You can **CHAIN** streams. One stream can write to another stream, which writes to another stream... often ending with a network or file stream.



Pool Puzzle

Your **job** is to take code snippets from the pool and place them into the blank lines in the 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 shown to the right.



Mini Sharpen your pencil

What text does the app write to `delivery.txt`?

```

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._____();
    }
}

```

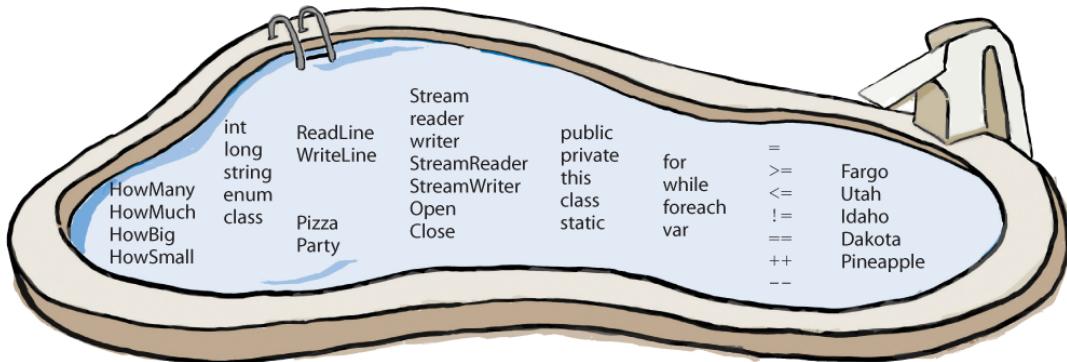
```

class Pizza {
    private _____writer;
    public Pizza(_____) {
        this.writer = writer;
    }
    public void Idaho(_____.Fargo f) {
        writer._____();
        writer._____();
    }
}

class Party {
    private _____reader;
    public Party(_____) {
        this.reader = reader;
    }
    public void HowMuch(_____) {
        q._____();
        reader._____();
    }
}

```

Note: each snippet from the pool can be used more than once!



Pool 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 (specifically, its `ToString` method) is used to print a lot of the output.

```
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();
    }
}
```



Mini Sharpen your pencil Solution

What text does the app write to `delivery.txt`?

North

`order.txt`

West

East

South

North

That's all folks!

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, in other cases using a format string. And 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} tells it to format the fourth argument in the local currency.

Oh, and one more thing—format strings work with Console.WriteLine 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 Window 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 readonly 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 in 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 a 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). And when you type text into Notepad and save it, Notepad converts each of the characters 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. But 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. But don't worry, 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. And it'll do that for your program when it opens a file. If you don't call the Close method, then it's possible for your program to keep a file locked open 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 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. Directory objects let you work with whole directories full of files.

Things you can do with File:

1. Find out if the file exists.

You can check to see if a file exists using the 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 OpenRead method to get data from a file, or the Create or OpenWrite method to write to the file.

3. Append text to the file.

The 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 GetLastAccessTime and GetLastWriteTime methods return the date and time when the file was last accessed and modified.

Things you can do with Directory:

1. Create a new directory.

Create a directory using the `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 `GetFiles` method; just tell the method which directory you want to know about, and it will do the rest.

3. Delete a directory.

Deleting a directory is really simple too. Just use the `Delete` method.

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.



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`.



SHARPEN YOUR PENCIL

.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
if (!Directory.Exists(@"C:\SYP")) { Directory.CreateDirectory(@"C:\SYP"); } if (Directory.Exists(@"C:\SYP\Bonk")) { Directory.Delete(@"C:\SYP\Bonk"); } Directory.CreateDirectory(@"C:\SYP\Bonk");	
Directory.SetCreationTime(@"C:\SYP\Bonk", new DateTime(1996, 09, 23));	
string[] files = Directory.GetFiles(@"C:\SYP\", "*.log", SearchOption.AllDirectories);	
File.WriteAllText(@"C:\SYP\Bonk\weirdo.txt", @"This is the first line and this is the second line and this is the last line");	
File.Encrypt(@"C:\SYP\Bonk\weirdo.txt"); <i>See if you can guess what this one does—you haven't seen it yet</i>	
File.Copy(@"C:\SYP\Bonk\weirdo.txt", @"C:\SYP\copy.txt");	
DateTime myTime = Directory.GetCreationTime(@"C:\SYP\Bonk");	
File.SetLastWriteTime(@"C:\SYP\copy.txt", myTime);	
File.Delete(@"C:\SYP\Bonk\weirdo.txt");	
Why did we put @ in front of each of the strings above?	
The filenames in the strings all start with C:\, so they'll work on Windows. What happens if you run that code on a Mac or Linux?	



SHARPEN YOUR PENCIL SOLUTION

.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"); } if (Directory.Exists(@"C:\SYP\Bonk")) { Directory.Delete(@"C:\SYP\Bonk"); }</pre>	Check if the C:\SYP folder exists. If it doesn't, create 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 above?	The @ keeps the backslashes in the string from being interpreted as <u>escape sequences</u> .
The filenames in the strings all start with C:\, so they'll work on Windows. What happens if you run that code on a Mac 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” (or “Go to Declaration”) feature in the IDE to show you 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();
    }
}
```

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.

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.

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

We've been telling you all chapter 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.

The diagram illustrates a C# using statement with handwritten annotations:

```
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");
}
```

- A bracket above the opening brace of the using block is labeled: "A using statement is always followed by an object declaration...".
- A bracket above the closing brace of the using block is labeled: "...and then a block of code within curly braces."
- An arrow points from the text "After the last statement in the using block executes, it" to the closing brace of the using block.
- An arrow points from the text "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." to the closing brace of the using block.
- A callout box contains the following text:

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
}
```

When you declare an object in a using block and that object's Dispose method is called automatically.

Every stream has a Dispose method that closes the stream. When if 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.

THERE ARE NO DUMB QUESTIONS

Q: Why did you put an @ in front of the strings that contained the filenames in that “Sharpen Your Pencil” exercise?

A: When you add a string literal to your program, the compiler converts escape sequences like \n and \r to special characters. That makes it difficult to type filenames, which have a lot of backslash characters in them. 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: Remind me again—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 and \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. And 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: \\.

Q: You mentioned a stream called `MemoryStream` that writes to memory. Why would I want to use that?

A: We've been using streams to read and write files. But what if you want to read data from a file and then, well, do something with it? And that's where `MemoryStream` comes in. When you create a new `MemoryStream`, it starts keeping track of all data streamed into it—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` is 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.

Q: What do I do with data once I have it in a byte array?

A: One of the most common things that you'll do with byte arrays is to convert them to strings. For example, if you have a byte array called `bytes`, here's one way to convert a byte array to a string:

```
var converted = Encoding.UTF8.GetString(bytes);
```

We mentioned “encoding” earlier.
What do you think that means?



SLEUTH IT OUT

Here's a small console app that prints uses composite formatting to write a number to a `MemoryStream`, and then convert it to a byte array and then to a string. There's just one tiny little problem... *it doesn't work!*

Create a new console app and add this code to it. When you run it, it doesn't print anything to the console. Can you sleuth out the problem and fix it?

```
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()));
        }
    }
}
```

See if you can sleuth out what's wrong on your own before we give you the solution.

We just talked about `MemoryStream.ToArray` and `Encoding.UTF8.GetString` in the "No Dumb Questions." ↗

All streams implement `IDisposable`, so any time you use a stream, you should ALWAYS declare it inside a `using` statement. That makes sure it's always closed!

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.” So let’s start at the scene of the crime: our code that’s not working. And 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, the fact that 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:

Every stream has a Dispose method that closes the stream. When if 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.

We learned this critical clue just moments before we took on our mysterious code caper.

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*. So 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.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.



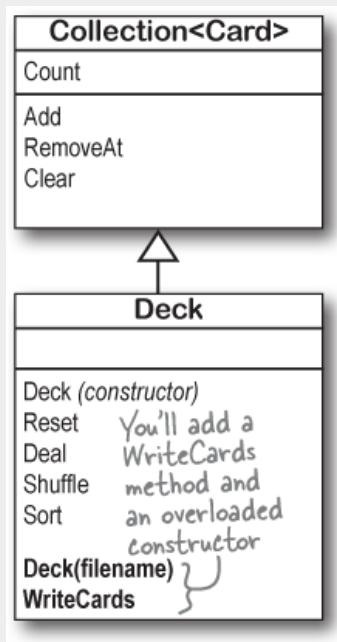
EXERCISE

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 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
}
```

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.

In Chapter 9 we 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.

```
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);
}
```



EXERCISE SOLUTION

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 to memory.
- You can write your data to a stream through a stream's **Write method**, and read data using its **Read method**.
- 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.
- A **StreamWriter** is an easy way to read data from 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. constructor, it creates a stream for you, 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 readonly 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 in 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 it).
- The **FileInfo class** is similar to the File class, except instead of using static methods it's instantiated.
- 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 at the end of the block.
- Use **multiple using statements** in a row to declare objects that are disposed at the end of the same block.

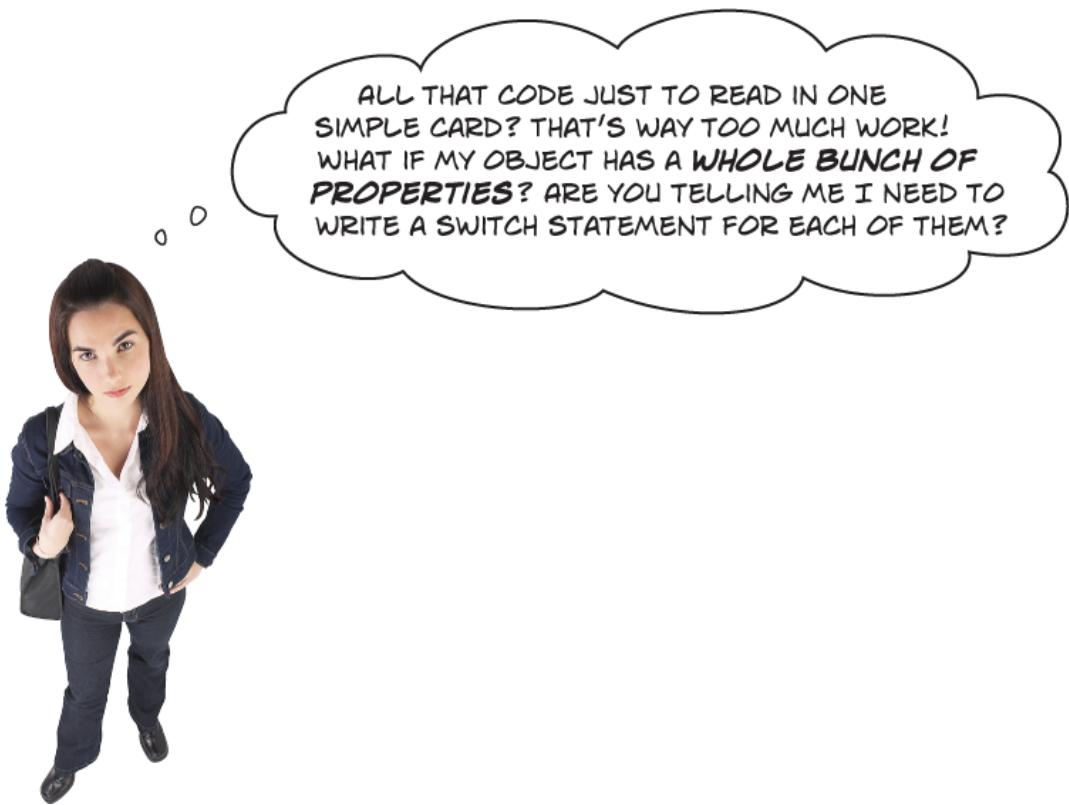
Windows and MacOS have different line endings

If you're running Windows, open Notepad. If you're running MacOS, openTextEdit. Create a file with these 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 4c 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. But the line break is encoded differently: on Windows it's encoded with two bytes, 13 and 10. But 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 Unix.



There's an easier way to store your objects in files. It's called serialization.

Serialization means writing an entire object's 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.

And on the other end, you can **deserialize** it, which 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`. And 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

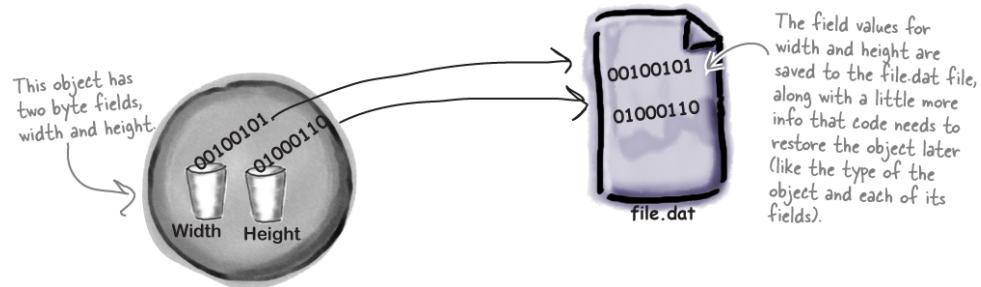


When you create an instance of an object, it has a **state**. Everything that an object "knows" is what makes one instance of a class different from another instance of the same class.

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.



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.

Object on the heap again

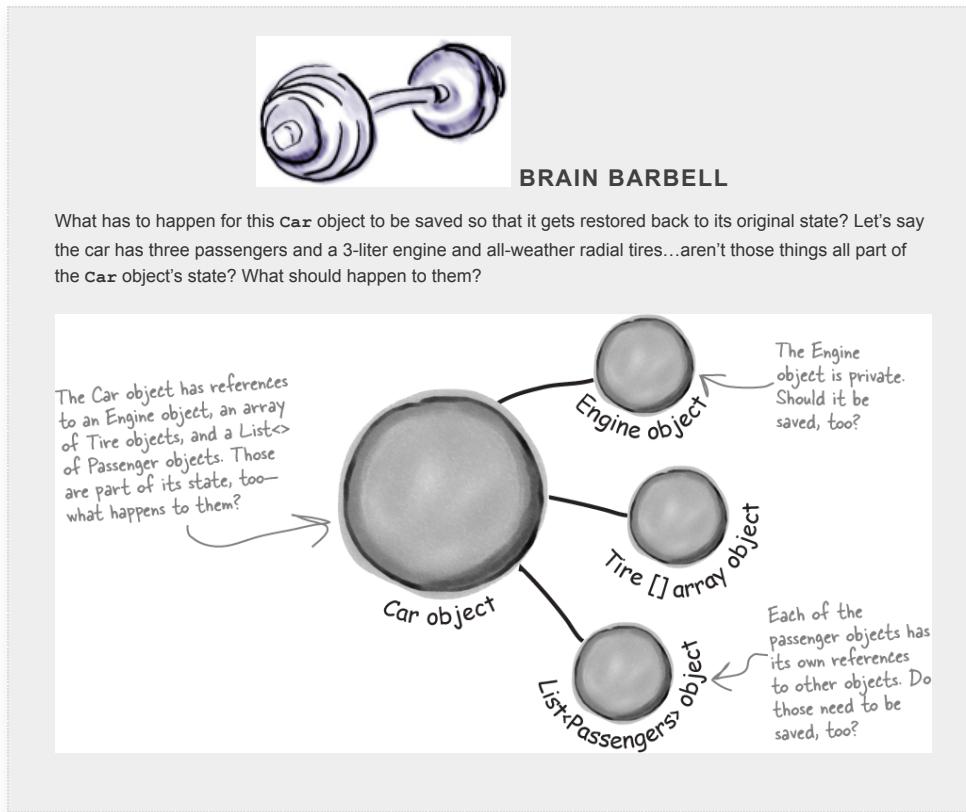


But what exactly IS an object's state? What needs to be saved?

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. But 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?

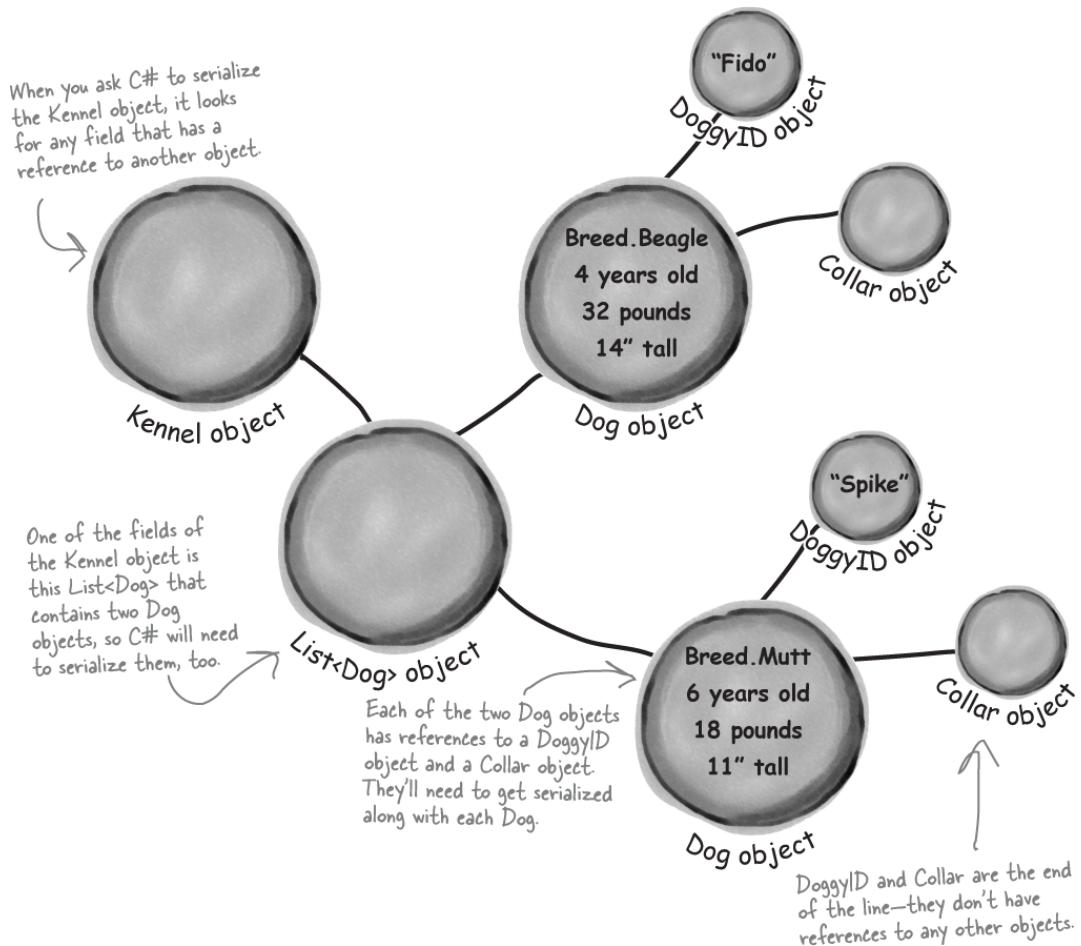
Think about it for a minute. What part of an object is potentially unique? Imagine what needs to be restored in order to get an object that's identical to the one that was saved. Somehow everything on the heap has to be written to the file.



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. But 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.



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 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
}

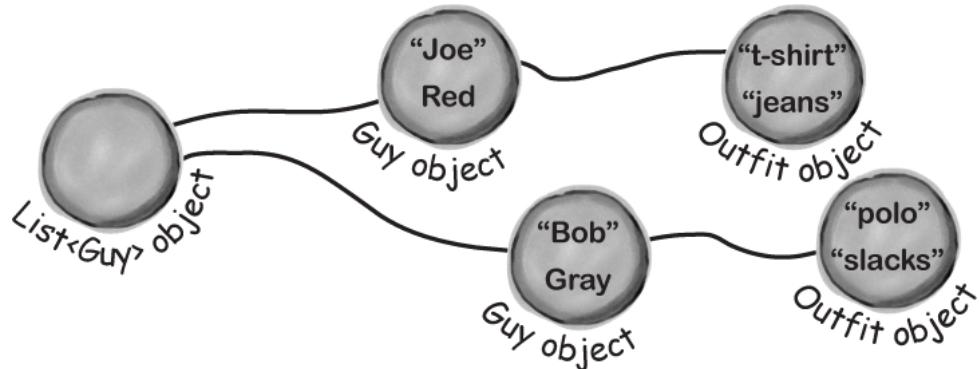
class Hairstyle {
    public HairColor Color { get; set; }
    public float Length { get; set; }
    public override string ToString() => $""
}
```

```
{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 that 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. And 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. And 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 deserialized 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
```



EXERCISE

We just showed you how to use `JsonSerializer` to serialize an object graph to a string. Can you figure out how to modify the `WriteCards` method and the overloaded `Deck` constructor that you added to your `Deck` class in the last exercise so they use `JsonSerializer`?

Here's a hint: your `WriteCards` method can be just two lines long, and your overloaded `Deck` method can be four lines. And don't forget to add `using System.Text.Json;` to the top of the file.

You should be able to refactor the `WriteCards` method and overloaded `Deck` constructor to use JSON serialization, and your app should still work without modifying any other code.



EXERCISE SOLUTION

We just showed you how to use JsonSerializer to serialize an object graph to a string. Can you figure out how to modify the WriteCards method and overloaded Deck constructor that you added to your Deck class in the last exercise so they use JsonSerializer?

```
public void WriteCards(string filename)
{
    var jsonString = JsonSerializer.Serialize(this); } This method serializes the Deck collection
    File.WriteAllText(filename, jsonString); } and all of its cards (with their suits and
} values) to a string, and writes it to a file.

public Deck(string filename) To deserialize the deck, we just need to read the JSON from the
{ file, deserialize it into a List, and add each card to the deck.
    var jsonString = File.ReadAllText(filename);
    var deserializedDeck = JsonSerializer.Deserialize<List<Card>>(jsonString); ←
    foreach (var card in deserializedDeck)
        Add(card);
}
```



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:

```
[
  {
    "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"
    }
  }
]
```

Let's break down exactly what we're seeing:

- The JSON starts and ends with square braces `[]`. 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 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.

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. But did you think about what you *didn’t* see? JSON data **does not include the names of types** like Guy, Outfit, HairColor, or HairStyle. 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; }
}
```

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");
}
```

We’re deserializing the data from a List of Guy objects into a Stack of Dude objects.

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). And 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—it prints Bob first because it’s popping them off of the stack in first-in-last-out, and Joe was added before Bob:

```
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
```



SHARPEN YOUR PENCIL

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.

You used the emoji panel in [Chapter 1](#) to enter emoji.

```
Console.WriteLine(JsonSerializer.Serialize(3));
.....
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("🐘"));
.....
```

C# strings are encoded with Unicode

We've been using strings since you typed "Hello, world!" into the IDE at the start of [Chapter 1](#). And 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?***

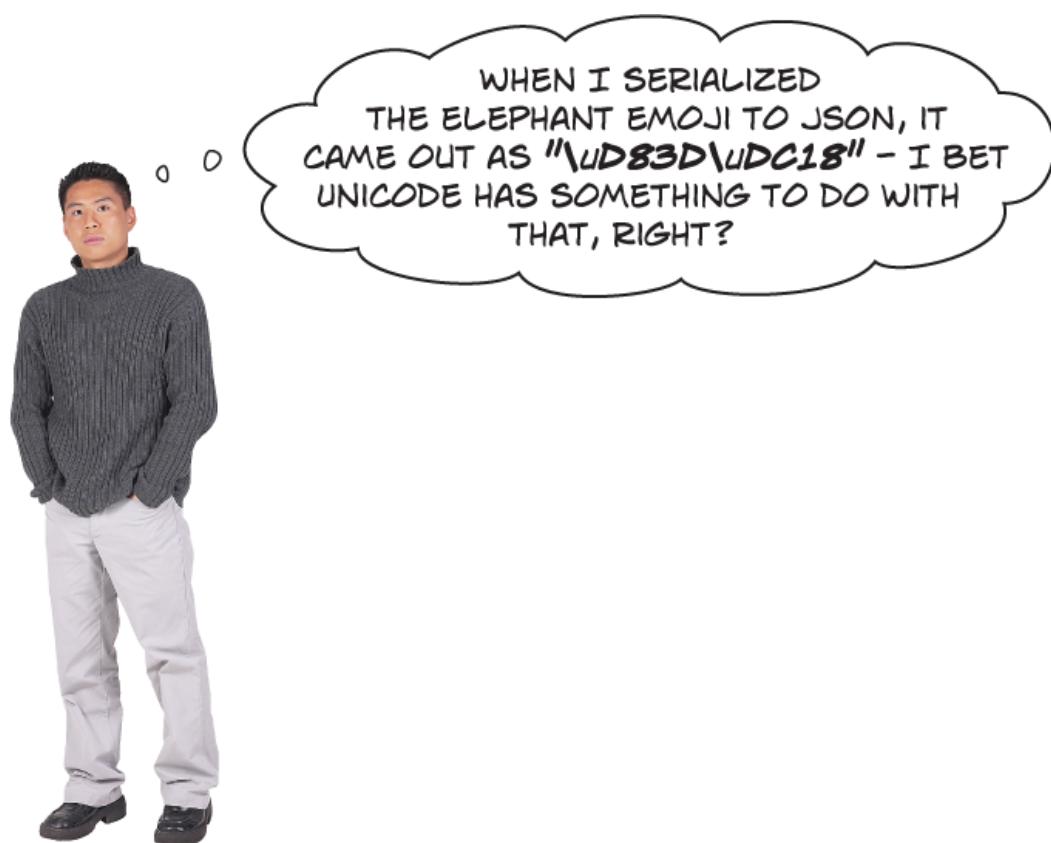
A C# string is a **read-only collection of chars**. So if you actually look at how a string is stored in memory, you'll see the string "Elephant" stored as chars 'E', 'l', 'e', 'p', 'h', 'a', 'n', and 't'. But now ask yourself... ***what exactly is a char?***

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 do pretty much anything you want with them—and you're guaranteed that you'll always get the correct character.

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 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.



Every Unicode character—including emoji—has a unique number.

The number for a Unicode character is called a **code point**. You 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 this “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 numbers 0 to 9 and letters A to F. 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 three bytes: 0x01 (or 1), 0xF4 (or 244), and 0x18 (or 24).

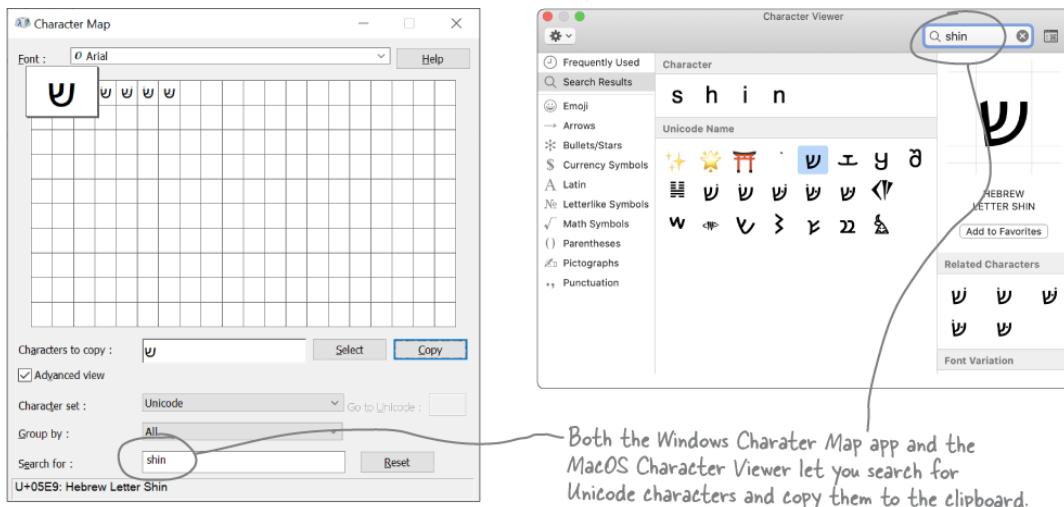
But that's not what JSON serializer printed. It printed a longer Hex number: D83DDC18. That's because **the C# char type uses UTF-16**, which uses code points that consist of either one or two two-byte numbers. The UTF-16 code point of the elephant emoji is 0xD83D 0xDC18. UTF-8 is much more popular than 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.

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 Mac, press **Ctrl-⌘-space** to pop up the Character Viewer. Then search for the Hebrew letter shin (ש) and copy it to the clipboard.



Both the Windows Character Map app and the Mac OS Character Viewer let you search for Unicode characters and copy them to the clipboard.

Place your cursor at the end of the string between the space and the quotation mark, and paste 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 þψ");
```

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 þוּשׁ");
```

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 לוישׁ – watch carefully what happens when the selection reaches the Hebrew letters. It skips to the shin ψ 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.WriteAllBytes 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 Encoding.UTF8.GetString, which converts a byte array with UTF-8 encoded characters to a string. Now **open the file in the Notepad** (Windows) **or TextEdit** (Mac). It says “Eureka!”

2. Then add code to write the bytes as hex numbers.

Whenever you’re encoding data you 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 Hebrew letters “שלום” instead of “Eureka!”

You just added the Hebrew text "שלום" 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 "שלום" 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 add using System.Text; to the top:

```
File.WriteAllText("eureka.txt", "שלום",
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 two-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 Text Edit 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 this: \uD83D\uDC18 – which we now know is the four-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 code 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 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 two-byte UTF-8 code points have the same code points in UTF-16. But 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 Hebrew letter shin is ox05E9 in both UTF-8 and UTF-16, the elephant emoji is ox1F418 in UTF-8 and oxD8ED oxDC18 in UTF-16.

Use \u escape sequences to include Unicode in strings

When you serialized the elephant emoji, JsonSerializer generated this: \uD83D\uDC18 – which we now know is the four-byte UTF-16 code point for 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 two-byte code points like ⚡ are represented with a \u followed by the hex code point (\u05E9) and characters with four-byte code points like 🐘 are represented with \u and the highest two bytes, followed by \u and the lowest two bytes.

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 4 bytes long. That's yet another Unicode encoding, and it's really useful because it's really easy to convert UTF-8 to UTF-32: just pad the hex number with zeroes, 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");
```

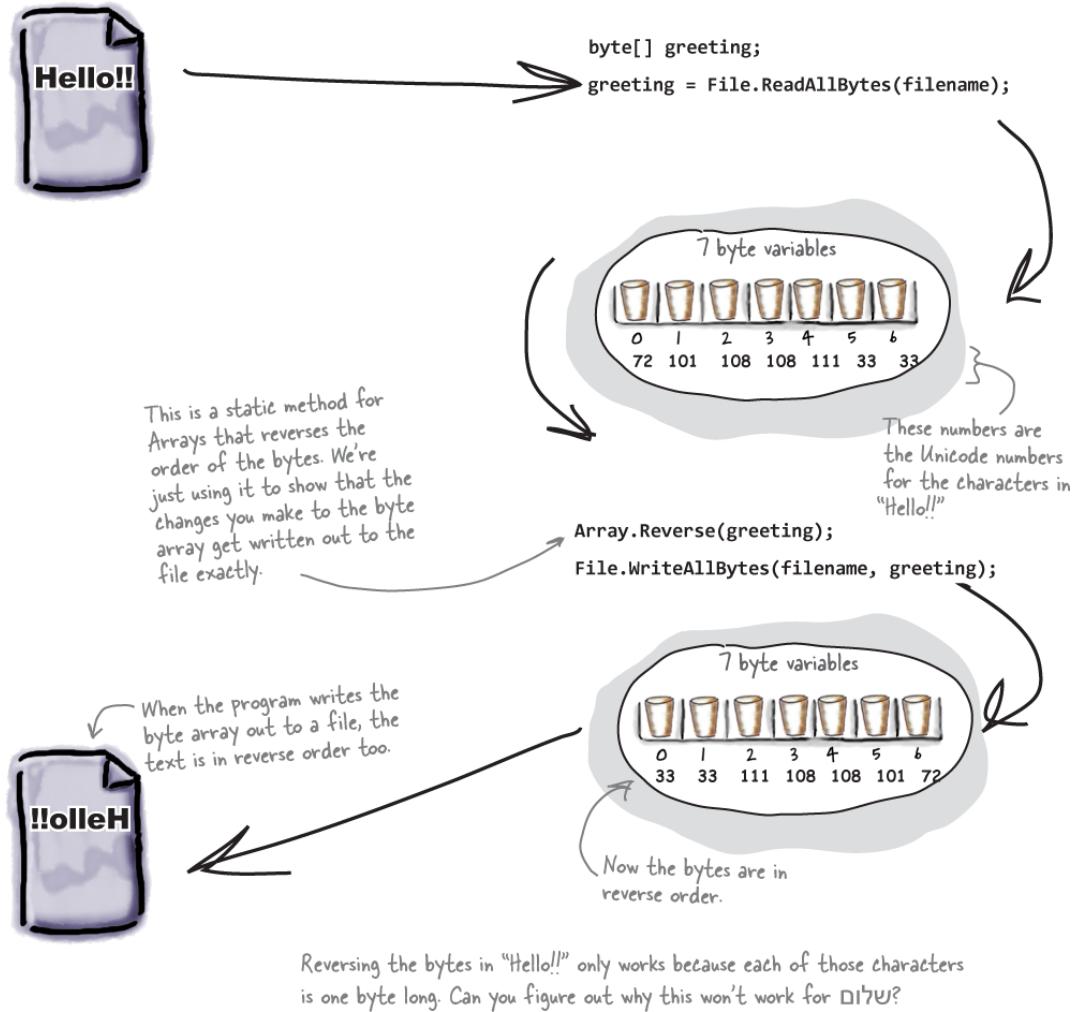
Run your app again, then open both of those files in Notepad or Text Edit. You should see the correct character written to the file.

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.

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.



Use a BinaryWriter to write binary data

`StreamWriter` also encodes your data. It just specializes in text and text encoding—it defaults to UTF-8.

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. So let's create a new Console Application that uses BinaryWriter to write binary data to a file.

Do this!

1. Start by creating a console Application and setting up some data to write to a file.

```
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. There's also the `File.OpenWrite` method, which opens the existing one and starts overwriting it from the beginning.

2. To use a `BinaryWriter`, first you need to open a new stream with `File.Create`:

```
using (var output =  
File.Create("binarydata.dat"))  
    using (var writer = new  
BinaryWriter(output))  
    {
```

3. Now just call its `Write` method. Each time you do, 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);  
}
```

The diagram illustrates the flow of data from the `Write` statements to the final output file. A brace on the left groups the five `writer.Write` calls. An arrow points from this brace to a text box containing the following explanatory text:

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.

A second arrow points from the text box to the right, indicating that the `FileStream` writes the bytes to the end of the file.

4. 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.Write("{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? Put a bracket under each group of bytes that corresponds with each statement, and write the name of the variable under it.



SHARPEN YOUR PENCIL

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!" are have code points that consist of a single byte. Downloaded UnicodeData.txt from unicode.org (we gave you the URL earlier) and use it to look up the code points for each character.

bytes



SHARPEN YOUR PENCIL SOLUTION

float and int values take up 4 bytes when you write them to a file. If you'd used long or double, then they'd take up 8 bytes each.

86 29 e8 02 06 48 65 be be bf 21 2f 81 00 74 f6 d8 f5 43 45 - 20 bytes

intValue stringValue byteArray floatValue charValue

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.

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!** And it has no way of knowing. Your float value of `491.695F` was encoded as `d8 f5 43 45`. But 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.

1. Start out by setting up the `FileStream` and `BinaryReader` objects:

```
using (var input =
File.OpenRead("binarydata.dat"))
    using (var reader = new
BinaryReader(input))
{
```

2. 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.

3. You tell `BinaryReader` what type of data to read by calling its different methods.

```
Console.Write("int: {0} string: {1}
bytes: ", intRead, stringRead);
foreach (byte b in byteArrayRead)
    Console.Write("{0} ", b);
Console.Write(" 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
```

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. We've been talking about hexadecimal (or hex) throughout the chapter.

It turns out that hex is a convenient way to display bytes in a file. A byte takes 2 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 easy to spot patterns. And 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—and since the Latin characters all have one-byte UTF-8 code points, each will be represented by a two-digit hex number from 00 to 7F.

Then print each segment, starting with its offset (or position in the file) written as a hex number followed by a colon and a space, then the first eight code points in hex, then a divider (a space, two hyphens, and another space), then the next eight code points, then four spaces and the dumped characters:

```
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:

```
0000: 57 68 65 6e 20 79 6f 75 -- 20 68 61 76 65 20 65 6c When you  
have el  
0010: 69 6d 69 6e 61 74 65 64 -- 20 74 68 65 20 69 6d 70 iminated  
the imp  
0020: 6f 73 73 69 62 6c 65 2c -- 20 77 68 61 74 65 76 65 ossible,  
whateve  
0030: 72 20 72 65 6d 61 69 6e -- 73 2c 20 68 6f 77 65 76 r  
remains, howev  
0040: 65 72 20 69 6d 70 72 6f -- 62 61 62 6c 65 2c 20 6d er  
improbable, m  
0050: 75 73 74 20 62 65 20 74 -- 68 65 20 74 72 75 74 68 ust be  
the truth  
0060: 2e 20 2d 20 53 68 65 72 -- 6c 6f 63 6b 20 48 6f 6c . -  
Sherlock Hol  
0070: 6d 65 73 0a -- mes.
```

And that's the 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. The first four characters are the offset in the file—the first line starts at character 0, the next at character 16 (or hex 10), then character 32 (hex 20), etc.

A hex dump is a hexadecimal view of data in a file or memory, and can be really useful tool to help you to debug binary data.

Use StreamReader to build a simple hex dumper

Let's build a simple hex dump app that uses StreamReader to read data from a file and writes it 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 or TextEdit 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 Main method—it reads the textdata.txt file and writes a hex dump to the console:

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.

```
var position = 0;
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;

        // This loop goes through the characters and prints each of them to a line in the output.
        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));
    }
}
```

A StreamReader's EndOfStream property returns false if there are characters still left to read in the file.

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. And 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  
Elementary, my d  
0010: 65 61 72 20 57 61 74 73 -- 6f 6e 21           ear  
Watson!
```

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? ??CE
```

These bytes were 81 and f8 in the “Sharpen your pencil” solution, but `StreamReader` changed them to fd.

The text characters (“Hello!”) seem okay. 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); ← The Stream.Read method takes three
                                                       : arguments: the byte array to read
                                                       : into, the starting index in the array,
                                                       : and the number of bytes to read.
```

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 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. And that means you can use Encoding.UTF8.GetString to convert it. And 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? { Now your app
0010: d8 f5 43 45          --           ??CE                 reads all of the
                                                               bytes from the
                                                               file, not just the
                                                               text characters.
```

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!/. .  
0010: d8 f5 43 45          --          .. CE
```

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**: C:\>HexDump myfile.txt

If you don't pass it a filename, it reads its input from **standard input** (or **stdin**): C:\>dir || HexDump

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)
```

Let's modify the hex dumper to use command-line arguments. First, **add this GetInputStream method** that uses a switch expression to return a stream. If the user passed a command line argument, the app will use it to call File.OpenRead to get a FileStream. If they didn't, the app will open the standard input instead. static Stream GetInputStream(string[] args) => args.Length switch

```
static Stream GetInputStream(string[] args) =>  
    args.Length switch  
    {  
        0 => Console.OpenStandardInput(),  
        _ => File.OpenRead(args[0]),  
    };
```

The `Console.OpenStandardInput` method returns a Stream object that's connected to the standard input for the app. If you run the app in the IDE and type input into the console or terminal, that data will get passed to the stream.

Now modify the Main method to use the stream—and it will work the same whether that stream comes from a file or standard input. We'll

```
static void Main(string[] args)  
{  
    var position = 0;  
    using (Stream input = GetInputStream(args))  
    {  
        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) ←  
            The app gets the  
            command-line arguments  
            in its args parameter  
            and passes them to the  
            GetInputStream method.  
            You'll also need to delete the  
            first line of the while block  
            that declares bytesRead and  
            calls input.Read on the stream.
```

Test the dump from stdin by debugging the app in the IDE. Type some text into the console or terminal—as soon as you press enter, the app will generate the hex dump for it. You can also test the command line argument in the IDE. Right-click on the project in the solution, then:

- On Windows, choose Properties, then click Debug and enter the filename in the Application arguments box (either the full path or the name of a file in the binary folder)
- On Mac, choose Options, expand Run >> Configurations, click Default, and enter the filename in the Arguments box

You can also run the app from the command line. **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 from that folder.

On 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 of the command will include a line like this: `HexDump -> /path-to-binary/osx-x64/publish/`

Then you can open a Terminal window, cd to the full path that it printed, and run `./HexDump`.

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 Hebrew letters in מילן take two bytes? And why did it write 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. But 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 מילן 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, 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. And 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 the 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 to binary files. Its methods for reading and writing operate with bytes and byte arrays. But a lot of programs work exclusively with text files, and that’s where the 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 in 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.

Q: When should I use File, and when should I use FileInfo?

A: The main difference between the File and FileInfo classes is that the methods in File are static, so you don’t need to create an instance of them. On the other hand, FileInfo requires that you instantiate it with a filename. In some cases, that would be more cumbersome, like if you only need to perform a single file operation (like just deleting or moving one file). On the other hand, if you need to do many file operations to the same file, then it’s more efficient to use FileInfo, because you only need to pass it the filename once. You should decide which one to use based on the particular situation you encounter. In other words, if you’re doing one file operation, use File. If you’re doing a lot of file operations in a row, use FileInfo.

If you're writing a string that only has Unicode characters with low numbers, it writes one byte per character. But if it's got high-numbered characters, they'll be written using two or more bytes each.

One more thing! We showed you basic serialization with JsonSerializer. But 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.

Flip back to [Chapter 5](#) and have a look at the `SwordDamage` class. Its `Damage` property has a private set accessor:

```
public int Damage { get; private set; } ← 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.
```

It also has a constructor that takes an `int` parameter:

```
public SwordDamage(int startingRoll)
```

You'll be able to use `JsonSerializer` to serialize a `SwordDamage` object without any problems. But 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, 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 entire object's state to a file or string. Deserialization means reading the object's state back from that file or string.
- 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.
- **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.
- A C# string is a **read-only collection of chars**. C# characters are represented as Unicode. The C# `char` type uses **UTF-16**, a variable-length encoding that encodes characters using either one or two two-byte sequences.
- 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.
- StreamWriter and StreamReader work well with text, but will not handle many characters outside of the Latin character sets. Use **BinaryWriter** and **BinaryReader** to read and write binary data.
- 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.
- 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.
- The **Encoding.UTF8.GetString** method converts a byte array with a UTF-8 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.
- C# makes the command-line arguments for a console app available as the `args` string array that gets passed to the Main method.
- The **Console.OpenStandardInput** method returns a Stream object that's connected to the app's `stdin`.