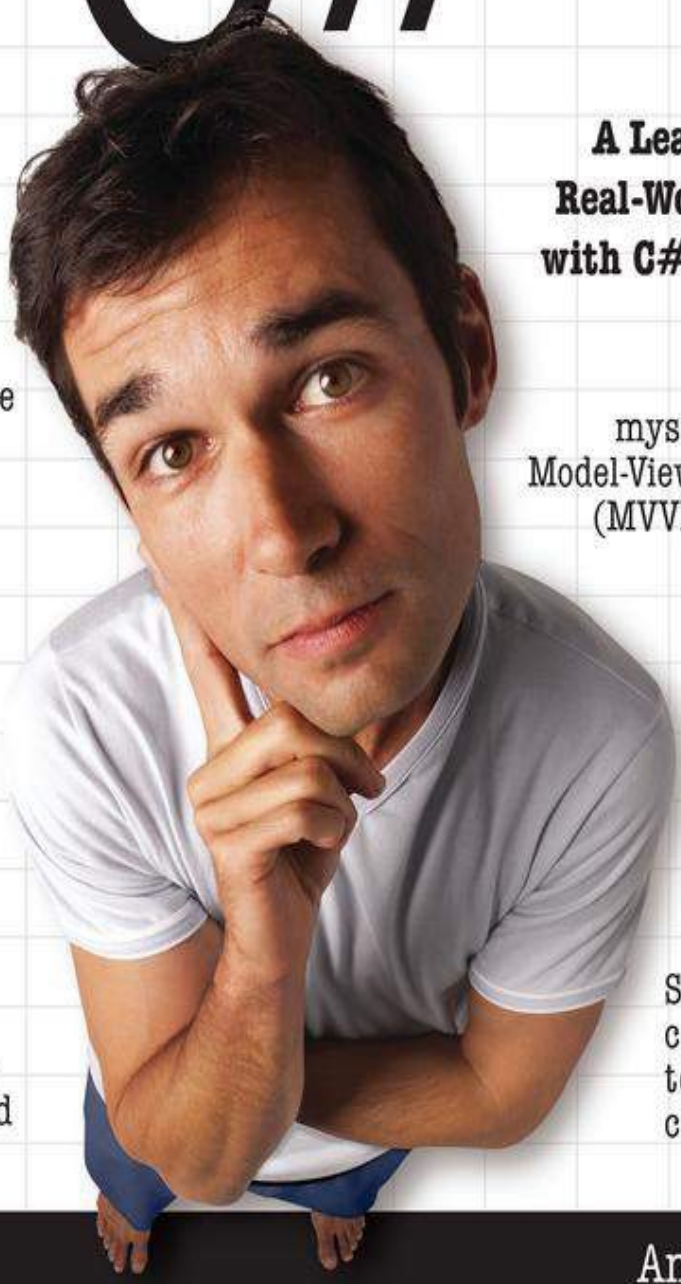
A Brain-Friendly Guide

3rd Edition
Updated to include Visual Studio 2013 and Windows 8.1

Head First C#

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

Boss your objects around with abstraction and inheritance



Unravel the mysteries of the Model-View-ViewModel (MVVM) pattern

Build a fully functional retro classic arcade game

Learn how asynchronous programming helped Sue keep her users thrilled

See how Jimmy used collections and LINQ to wrangle an unruly comic book collection

Andrew Stellman
& Jennifer Greene

O'REILLY®

Head First C#

Table of Contents

Special Upgrade Offer
A Note Regarding Supplemental Files
Advance Praise for Head First C#
Praise for other Head First books

How to Use this Book: Intro
- Who is this book for?
  - Who should probably back away from this book?
- We know what you're thinking.
- And we know what your brain is thinking
- Metacognition: thinking about thinking
- Here's what WE did
- Here's what You can do to bend your brain into submission
- Read me
- What version of Windows are you using?
  - Using Windows 8 or later? Then you'll start with Windows Store apps
  - Don't have Windows 8 or VS2013 yet? No problem — you'll start with WPF apps
  - You'll move on to create desktop applications
- The technical review team
- Acknowledgments
- Safari® Books Online

1. Start Building with C#: Build something cool, fast!
Why you should learn C#
Here’s what the IDE automates for you...
What you get with Visual Studio and C#...
C# and the Visual Studio IDE make lots of things easy
What you do in Visual Studio...
What Visual Studio does for you...
Aliens attack!
Only you can help save the Earth
Here’s what you’re going to build
Start with a blank application
Set up the grid for your page
Add controls to your grid
Use properties to change how the controls look
Controls make the game work
You’ve set the stage for the game
What you’ll do next
Add a method that does something
Use the IDE to create your own method
Fill in the code for your method
Finish the method and run your program
Here’s what you’ve done so far
Add timers to manage the gameplay
Make the Start button work
Run the program to see your progress
Add code to make your controls interact with the player
Dragging humans onto enemies ends the game
Your game is now playable
Make your enemies look like aliens
Add a splash screen and a tile
Publish your app
Use the Remote Debugger to sideload your app
Start remote debugging

2. It’s all Just Code: Under the hood
When you’re doing this...
...the IDE does this
Where programs come from
Every program starts out as source code files
Build the program to create an executable
The .NET Framework gives you the right tools for the job
Your program runs inside the Common Language Runtime
The IDE helps you code
Anatomy of a program
Let’s take a closer look at your code
Two classes can be in the same namespace
Your programs use variables to work with data
Declare your variables
Variables vary
You have to assign values to variables before you use them
A few useful types
C# uses familiar math symbols
Use the debugger to see your variables change
Loops perform an action over and over
Use a code snippet to write simple for loops
if/else statements make decisions
Build an app from the ground up
Make each button do something
Set up conditions and see if they’re true
Use logical operators to check conditions
Windows Desktop apps are easy to build
Rebuild your app for Windows Desktop
Your desktop app knows where to start
You can change your program’s entry point
So what happened?
When you change things in the IDE, you’re also changing your code
Wait, wait! What did that say?

3. Objects: Get Oriented!: Making code make sense
How Mike thinks about his problems
How Mike's car navigation system thinks about his problems
Mike's Navigator class has methods to set and modify routes
Some methods have a return value
Use what you've learned to build a program that uses a class
So what did you just build?
Mike gets an idea
He could create three different Navigator classes...
Mike can use objects to solve his problem
You use a class to build an object
An object gets its methods from its class
When you create a new object from a class, it's called an instance of that class
A better solution...brought to you by objects!
Theory and practice
A little advice for the code exercises
An instance uses fields to keep track of things
Methods are what an object does. Fields are what the object knows.
Let's create some instances!
Thanks for the memory
What's on your program's mind
You can use class and method names to make your code intuitive
Give your classes a natural structure
Let's build a class diagram
Class diagrams help you organize your classes so they make sense
Build a class to work with some guys
Create a project for your guys
Build a form to interact with the guys
There's an easier way to initialize objects
A few ideas for designing intuitive classes

4. Types and References: It’s 10:00. Do you know where your data is?
The variable’s type determines what kind of data it can store
A variable is like a data to-go cup
10 pounds of data in a 5-pound bag
Even when a number is the right size, you can’t just assign it to any variable
So what happened?
When you cast a value that’s too big, C# will adjust it automatically
C# does some casting automatically
When you call a method, the arguments must be compatible with the types of the parameters
Debug the mileage calculator
Combining = with an operator
Objects use variables, too
Using an int
Using an object
Refer to your objects with reference variables
References are like labels for your object
If there aren’t any more references, your object gets garbage-collected
Multiple references and their side effects
Two references means TWO ways to change an object’s data
A special case: arrays
Use each element in an array like it is a normal variable
Arrays can contain a bunch of reference variables, too
Welcome to Sloppy Joe’s Budget House o’ Discount Sandwiches!
Objects use references to talk to each other
Where no object has gone before
Build a typing game
Controls are objects, just like any other object
I. C# Lab: A Day at the Races

5. Encapsulation: Keep your privates... Private
Kathleen is an event planner
What does the estimator do?
You're going to build a program for Kathleen
Kathleen's test drive
Each option should be calculated individually
It's easy to accidentally misuse your objects
Encapsulation means keeping some of the data in a class private
Use encapsulation to control access to your class's methods and fields
But is the RealName field REALLY protected?
Private fields and methods can only be accessed from inside the class
Encapsulation makes your classes...
Mike's navigator program could use better encapsulation
Think of an object as a black box
A few ideas for encapsulating classes
Encapsulation keeps your data pristine
A quick example of encapsulation
Properties make encapsulation easier
Build an application to test the Farmer class
Use automatic properties to finish the class
Fully encapsulate the Farmer class
What if we want to change the feed multiplier?
Use a constructor to initialize private fields

6. Inheritance: Your object's family tree
Kathleen does birthday parties, too
We need a BirthdayParty class
Here's what we're going to do:
Build the Part y Planner version 2.0
One more thing...can you add a $100 fee for parties over 12?
When your classes use inheritance, you only need to write your code once
Dinner parties and birthday parties are both parties
Build up your class model by starting general and getting more specific
How would you design a zoo simulator?
Use inheritance to avoid duplicate code in subclasses
Different animals make different noises
Think about what you need to override
Think about how to group the animals
Create the class hierarchy
Every subclass extends its base class
C# always calls the most specific method
Use a colon to inherit from a base class
We know that inheritance adds the base class fields, properties, and methods to the subclass...
...but some birds don't fly!
A subclass can override methods to change or replace methods it inherited
Any place where you can use a base class, you can use one of its subclasses instead
A subclass can hide methods in the superclass
Hiding methods versus overriding methods
Use different references to call hidden methods
Use the new keyword when you're hiding methods
Use the override and virtual keywords to inherit behavior
A subclass can access its base class using the base keyword
When a base class has a constructor, your subclass needs one, too
The base class constructor is executed before the subclass constructor
Now you're ready to finish the job for Kathleen!
Build a beehive management system
How you'll build the beehive management system
Use inheritance to extend the bee management system

7. Interfaces and Abstract Classes: Making classes keep their promises
Let's get back to bee-sics
Lots of things are still the same
We can use inheritance to create classes for different types of bees
An interface tells a class that it must implement certain methods and properties
Use the interface keyword to define an interface
Now you can create an instance of NectarStinger that does both jobs
Classes that implement interfaces have to include ALL of the interface's methods
Get a little practice using interfaces
You can't instantiate an interface, but you can reference an interface
Interface references work just like object references
You can find out if a class implements a certain interface with "is"
Interfaces can inherit from other interfaces
Any class that implements an interface that inherits from IWorker must implement its methods and properties
The RoboBee 4000 can do a worker bee's job without using valuable honey
is tells you what an object implements; as tells the compiler how to treat your object
A CoffeeMaker is also an Appliance
Upcasting works with both objects and interfaces
Downcasting lets you turn your appliance back into a coffee maker
When downcasting fails, as returns null
Upcasting and downcasting work with interfaces, too
There's more than just public and private
Access modifiers change visibility
Some classes should never be instantiated
An abstract class is like a cross between a class and an interface
Like we said, some classes should never be instantiated
Solution: use an abstract class
An abstract method doesn't have a body
The four principles of object-oriented programming
Polymorphism means that one object can take many different forms
Keep your eyes open for polymorphism in the next exercise!

8. Enums and Collections: Storing lots of data
Strings don't always work for storing categories of data
Enums let you work with a set of valid values
Enums let you represent numbers with names
We could use an array to create a deck of cards...
...but what if you wanted to do more?
Arrays are hard to work with
Lists make it easy to store collections of...anything
Lists are more flexible than arrays
Lists shrink and grow dynamically
Generics can store any type
Collection initializers are similar to object initializers
Let's create a List of Ducks
Here's the initializer for your List of Ducks
Lists are easy, but SORTING can be tricky
Lists know how to sort themselves
IComparable<Duck> helps your list sort its ducks
An object's CompareTo() method compares it to another object
Use IComparer to tell your List how to sort
Create an instance of your comparer object
Multiple IComparer implementations, multiple ways to sort your objects
IComparer can do complex comparisons
Overriding a ToString() method lets an object describe itself
Update your foreach loops to let your Ducks and Cards print themselves
Add a ToString() method to your Card object, too
You can upcast an entire list using IEnumerable
Combine your birds into a single list
You can build your own overloaded methods
Use a dictionary to store keys and values
The dictionary functionality rundown
Your key and value can be different types
Build a program that uses a dictionary
And yet MORE collection types...
Generic collections are an important part of the .NET Framework
A queue is FIFO — First In, First Out
A stack is LIFO — Last In, First Out

9. Reading and Writing Files: Save the last byte for me!
.NET uses streams to read and write data
Different streams read and write different things
Things you can do with a stream:
A FileStream reads and writes bytes to a file
Write text to a file in three simple steps
The Swindler launches another diabolical plan
Reading and writing using two objects
Data can go through more than one stream
Use built-in objects to pop up standard dialog boxes
ShowDialog() pops up a dialog box
Dialog boxes are just another WinForms control
Dialog boxes are objects, too
Use the built-in File and Directory classes to work with files and directories
Things you can do with File:
Things you can do with Directory:
Use file dialogs to open and save files (all with just a few lines of code)
IDisposable makes sure your objects are disposed of properly
Avoid filesystem errors with using statements
Use multiple using statements for multiple objects
Trouble at work
You can help Brian out by building a program to manage his excuses
Writing files usually involves making a lot of decisions
Use a switch statement to choose the right option
Use a switch statement to let your deck of cards read from a file or write itself out to one
Add an overloaded Deck() constructor that reads a deck of cards in from a file
What happens to an object when it's serialized?
But what exactly IS an object's state? What needs to be saved?
When an object is serialized, all of the objects it refers to get serialized, too...
Serialization lets you read or write a whole object graph all at once
You'll need a BinaryFormatter object
Now just create a stream and read or write your objects
If you want your class to be serializable, mark it with the [Serializable] attribute
Let's serialize and deserialize a deck of cards
.NET uses Unicode to store characters and text
C# can use byte arrays to move data around
Use a BinaryWriter to write binary data
Use BinaryReader to read the data back in
You can read and write serialized files manually, too
Find where the files differ, and use that information to alter them
Working with binary files can be tricky
Use file streams to build a hex dumper

How to make a hex dump
Working with hex
StreamReader and StreamWriter will do just fine (for now)
Use Stream.Read() to read bytes from a stream
II. C# Lab: The Quest

10. Designing Windows Store Apps with XAML: Taking your apps to the next level
- Brian's running Windows 8
- Windows Forms use an object graph set up by the IDE
- Use the IDE to explore the object graph
- Windows Store apps use XAML to create UI objects
- Redesign the Go Fish! form as a Windows Store app page
- Page layout starts with controls
- Rows and columns can resize to match the page size
- Use the grid system to lay out app pages
- Data binding connects your XAML pages to your classes
  - Context, path, and binding
  - Two-way binding can get or set the source property
  - Bind to collections with ObservableCollection
  - Use code for binding (without using any XAML at all!)
- XAML controls can contain text...and more
- Use data binding to build Sloppy Joe a better menu
- Use static resources to declare your objects in XAML
- Use a data template to display objects
- INotifyPropertyChanged lets bound objects send updates
- Modify MenuMaker to notify you when the GeneratedDate property changes

11. Async, Await, and Data Contract Serialization: Pardon the interruption
Brian runs into file trouble
Windows Store apps use await to be more responsive
Use the FileIO class to read and write files
Use the file pickers to locate file paths
Build a slightly less simple text editor
A data contract is an abstract definition of your object's data
Data contract serialization uses XML files
Use async methods to find and open files
KnownFolders helps you access high-profile folders
The whole object graph is serialized to XML
Stream some Guy objects to XML files
Take your Guy Serializer for a test drive
Use a Task to call one async method from another
Build Brian a new Excuse Manager app
Separate the page, excuse, and Excuse Manager
Create the main page for the Excuse Manager
Add the app bar to the main page
Build the ExcuseManager class
Add the code-behind for the page

12. Exception Handling: Putting out fires gets old
Brian needs his excuses to be mobile
But the program isn't working!
When your program throws an exception, .NET generates an Exception object
Brian's code did something unexpected
All exception objects inherit from Exception
The debugger helps you track down and prevent exceptions in your code
Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager
Uh oh — the code's still got problems...
Handle exceptions with try and catch
What happens when a method you want to call is risky?
Use the debugger to follow the try/catch flow
If you have code that ALWAYS should run, use a finally block
Use the Exception object to get information about the problem
Use more than one catch block to handle multiple types of exceptions
One class throws an exception that a method in another class can catch
Bees need an OutOfHoney exception
An easy way to avoid a lot of problems: using gives you try and finally for free
Exception avoidance: implement IDisposable to do your own cleanup
The worst catch block EVER: catch-all plus comments
You should handle your exceptions, not bury them
Temporary solutions are OK (temporarily)
A few simple ideas for exception handling
Brian finally gets his vacation...
...and things are looking up back home!

13. Captain Amazing: The Death of the Object
Your last chance to DO something... your object's finalizer
When EXACTLY does a finalizer run?
You can SUGGEST to .NET that it's time to collect the garbage
Dispose() works with using; finalizers work with garbage collection
Finalizers can't depend on stability
Make an object serialize itself in its Dispose()
A struct looks like an object...
...but isn't an object
Values get copied; references get assigned
Structs are value types; objects are reference types
Here's what happened...
The stack vs. the heap: more on memory
Use out parameters to make a method return more than one value
Pass by reference using the ref modifier
Use optional parameters to set default values
Use nullable types when you need nonexistent values
Nullable types help you make your programs more robust
"Captain" Amazing...not so much
Extension methods add new behavior to EXISTING classes
Extending a fundamental type: string

14. Querying Data and Building Apps With Linq: Get control of your data
Jimmy's a Captain Amazing super-fan...
...but his collection's all over the place
LINQ can pull data from multiple sources
.NET collections are already set up for LINQ
LINQ makes queries easy
LINQ is simple, but your queries don't have to be
Jimmy could use some help
Windows Store apps use page-based navigation
Use the IDE to explore app page navigation
Start building Jimmy an app
Use the new keyword to create anonymous types
LINQ is versatile
Add the new queries to Jimmy's app
LINQ can combine your results into groups
Combine Jimmy's values into groups
Use join to combine two collections into one sequence
Jimmy saved a bunch of dough
Use semantic zoom to navigate your data
Add semantic zoom to Jimmy's app
You made Jimmy's day
The IDE's Split App template helps you build apps for navigating data

15. Events and Delegates: What your code does when you're not looking
Ever wish your objects could think for themselves?
But how does an object KNOW to respond?
When an EVENT occurs...objects listen
Want to DO SOMETHING with an event? You need an event handler
One object raises its event, others listen for it...
Then, the other objects handle the event
Connecting the dots
Use a standard name when you add a method to raise an event
The IDE generates event handlers for you automatically
Generic EventHandlers let you define your own event types
C# does implicit conversion when you leave out the new keyword and type
Windows Forms use many different events
One event, multiple handlers
Windows Store apps use events for process lifetime management
Use the IDE to explore process lifetime management events
Add process lifetime management to Jimmy's comics
XAML controls use routed events
IsHitTestVisible determines if an element is "visible" to the pointer or mouse
Create an app to explore routed events
Connecting event senders with event listeners
"My people will get in touch with your people."
A delegate STANDS IN for an actual method
A delegate adds a new type to your project
Delegates in action
An object can subscribe to an event...
...but that's not always a good thing!
Use a callback to control who's listening
A callback is just a way to use delegates
MessageDialog uses the callback pattern
Use delegates to use the Windows settings charm

16. Architecting Apps with the mvvm Pattern: Great apps on the inside and outside
The Head First Basketball Conference needs an app
But can they agree on how to build it?
Do you design for binding or for working with data?
MVVM lets you design for binding and data
Use the MVVM pattern to start building the basketball roster app
User controls let you create your own controls
The ref needs a stopwatch
MVVM means thinking about the state of the app
Start building the stopwatch app's Model
Events alert the rest of the app to state changes
Build the view for a simple stopwatch
Add the stopwatch ViewModel
Finish the stopwatch app
Converters automatically convert values for binding
Converters can work with many different types
Styles set properties on multiple controls
Use a resource dictionary to share resources between pages
Visual states make controls respond to changes
Use DoubleAnimation to animate double values
Use object animations to animate object values
Build an analog stopwatch using the same ViewModel
UI controls can be instantiated with C# code, too
C# can build "real" animations, too
Create a project and add the pictures
Create a user control to animate a picture
Make your bees fly around a page
Use ItemsPanelTemplate to bind controls to a Canvas
Congratulations! (But you're not done yet...)
III. C# Lab Invaders
A. Leftovers: The top 10 things we wanted to include in this book

#1. There's so much more to Windows Store
#2. The Basics
...more basics...
#3. Namespaces and assemblies
...so what did I just do?
Building a "Hello World" program from the command line
#4. Use BackgroundWorker to make your WinForms responsive
#5. The Type class and GetType()
#6. Equality, IEquatable, and Equals()
#7. Using yield return to create enumerable objects
#8. Refactoring
Extract a method
Rename a variable
Consolidate a conditional expression
#9. Anonymous types, anonymous methods, and lambda expressions
#10. LINQ to XML
Save and load XML files
Query your data
Read data from an RSS feed
Did you know that C# and the .NET Framework can...
B. Windows Presentation Foundation: WPF Learner's Guide to Head First C#
Why you should learn WPF
Build WPF projects in Visual Studio
How to use this appendix
Chapter 1

Start with a blank application
Set up the grid for your window
Add controls to your grid
Use properties to change how the controls look
Controls make the game work
You've set the stage for the game
What you'll do next
Add a method that does something
Use the IDE to create your own method
Fill in the code for your method
Finish the method and run your program
Here's what you've done so far
Add timers to manage the gameplay
Make the Start button work
Run the program to see your progress
Add code to make your controls interact with the player
Dragging humans onto enemies ends the game
Your game is now playable
Make your enemies look like aliens
Chapter 2
Use the debugger to see your variables change
Build an app from the ground up
Chapter 10
WPF applications use XAML to create UI objects
Redesign the Go Fish! form as a WPF application
Page layout starts with controls
Rows and columns can resize to match the page size
Use data binding to build Sloppy Joe a better menu
Use static resources to declare your objects in XAML
Use a data template to display objects
INotifyPropertyChanged lets bound objects send updates
Modify MenuMaker to notify you when the GeneratedDate property changes
Chapter 11
C# programs can use await to be more responsive
Stream some Guy objects to a file
Take your Guy Serializer for a test drive
Chapter 12

Brian’s code did something unexpected
Use the IDE’s debugger to ferret out exactly what went wrong in the Excuse Manager
Uh-oh — the code’s still got problems...
Handle exceptions with try and catch
What happens when a method you want to call is risky?
Use the debugger to follow the try/catch flow
If you have code that should ALWAYS run, use a finally block
Chapter 14
Build a WPF comic query application
Chapter 15
XAML controls use routed events
IsHitTestVisible determines if an element is “visible” to the pointer or mouse
Create an app to explore routed events
Chapter 16
Use the MVVM pattern to start building the basketball roster app
User controls let you create your own controls
Build the view for a simple stopwatch
Finish the stopwatch app
Converters automatically convert values for binding
Converters can work with many different types
Build an analog stopwatch using the same ViewModel
UI controls can be instantiated with C# code, too
Create a user control to animate a picture
Make your bees fly around a page
Use ItemsPanelTemplate to bind controls to a Canvas
Congratulations! (But you’re not done yet...)
Index
Special Upgrade Offer

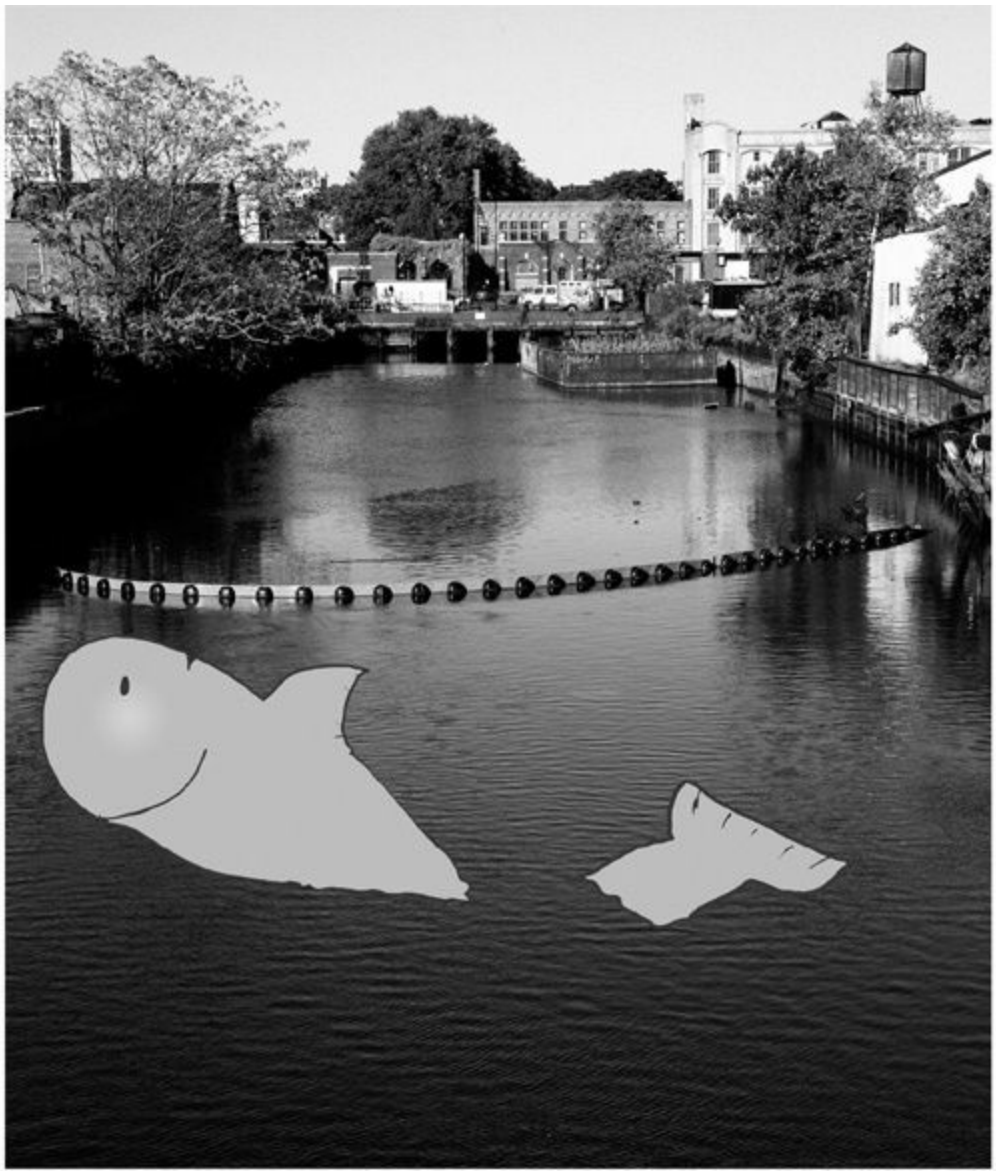
Head First C#

Jennifer Greene

Andrew Stellman

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

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



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

Special Upgrade Offer

If you purchased this ebook directly from oreilly.com, you have the following benefits:

- DRM-free ebooks — use your ebooks across devices without restrictions or limitations
- Multiple formats — use on your laptop, tablet, or phone
- Lifetime access, with free updates
- Dropbox syncing — your files, anywhere

If you purchased this ebook from another retailer, you can upgrade your ebook to take advantage of all these benefits for just $4.99. Click here to access your ebook upgrade.

Please note that upgrade offers are not available from sample content.

A Note Regarding Supplemental Files

Supplemental files and examples for this book can be found at http://examples.oreilly.com/0636920027812/. Please use a standard desktop web browser to access these files, as they may not be accessible from all ereader devices.

All code files or examples referenced in the book will be available online. For physical books that ship with an accompanying disc, whenever possible, we've posted all CD/DVD content. Note that while we provide as much of the media content as we are able via free download, we are sometimes limited by licensing restrictions. Please direct any questions or concerns to booktech@oreilly.com.

Advance Praise for *Head First C#*

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

— Shalewa Odusanya, Technical Account Manager, Google

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

— Johnny Halife, Chief Architect, *Mural.ly*

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

— Rebeca Duhn-Krahn, founding partner at Semphore Solutions

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

— Andy Parker, fledgling C# programmer

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

— Chris Burrows, developer for Microsoft's C# Compiler team

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

— Jay Hilyard, software developer, co-author of *C# 3.0 Cookbook*

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

— David Sterling, developer for Microsoft's Visual C# Compiler team

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

— Joseph Albahari, C# Design Architect at Egton Medical Information Systems, the UK's largest primary healthcare software supplier, co-author of *C# 3.0 in a Nutshell*

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

— Giuseppe Turitto, C# and ASP.NET developer for Cornwall Consulting Group

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

— Bill Mietelski, software engineer

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

— Krishna Pala, MCP

Praise for other *Head First books*

"I feel like a thousand pounds of books have just been lifted off of my head."

— Ward Cunningham, inventor of the Wiki and founder of the Hillside Group

"Just the right tone for the geeked-out, casual-cool guru coder in all of us. The right reference for practical development strategies — gets my brain going without having to slog through a bunch of tired stale professor-speak."

— Travis Kalanick, Founder of Scour and Red Swoosh Member of the MIT TR100

"There are books you buy, books you keep, books you keep on your desk, and thanks to O'Reilly and the Head First crew, there is the penultimate category, Head First books. They're the ones that are dog-eared, mangled, and carried everywhere. *Head First SQL* is at the top of my stack. Heck, even the PDF I have for review is tattered and torn."

— Bill Sawyer, ATG Curriculum Manager, Oracle

"This book's admirable clarity, humor and substantial doses of clever make it the sort of book that helps even non-programmers think well about problem-solving."

— Cory Doctorow, co-editor of Boing Boing Author, *Down and Out in the Magic Kingdom* and *Someone Comes to Town, Someone Leaves Town*

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

— Erich Gamma, IBM Distinguished Engineer, and co-author of *Design Patterns*

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

— Aaron LaBerge, VP Technology, ESPN.com

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

— Mike Davidson, CEO, Newsvine, Inc.

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

— Ken Goldstein, Executive Vice President, Disney Online

"Usually when reading through a book or article on design patterns, I'd have to occasionally stick myself in the eye with something just to make sure I was paying attention. Not with this book. Odd as it may sound, this book makes learning about design patterns fun.
"While other books on design patterns are saying 'Bueller... Bueller... Bueller...' this book is on the float belting out 'Shake it up, baby!'"

— Eric Wuehler

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

— Satish Kumar

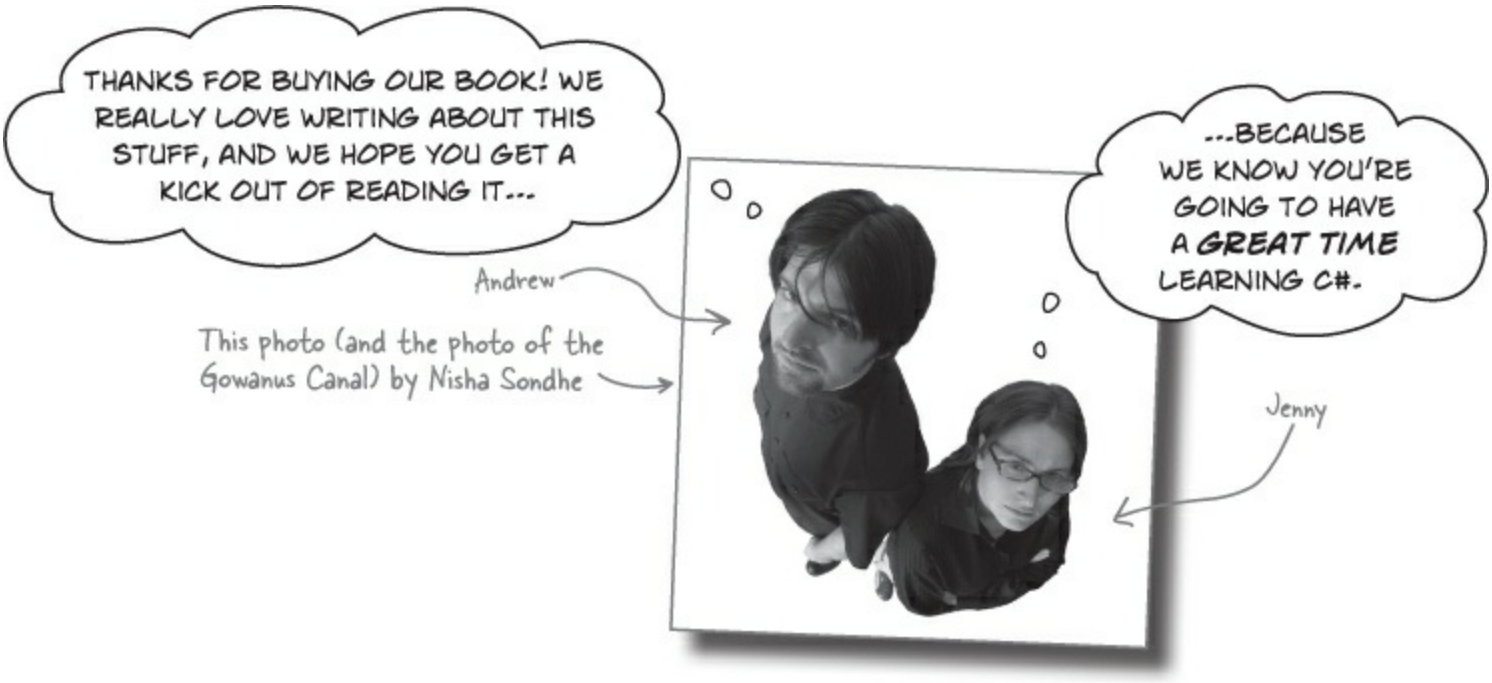
Other related books from O'Reilly

- Programming C# 4.0
- C# 4.0 in a Nutshell
- C# Essentials
- C# Language Pocket Reference

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

- Head First HTML5 Programming
- Head First iPhone and iPad Development
- Head First Mobile Web
- Head First Python

- Head First Web Design
- Head First WordPress
- Head First Java
- Head First Object-Oriented Analysis and Design (OOA&D)
- Head Rush Ajax
- Head First HTML with CSS and XHTML
- Head First Design Patterns
- Head First Servlets and JSP
- Head First EJB
- Head First PMP
- Head First SQL
- Head First Software Development
- Head First JavaScript
- Head First Ajax
- Head First Statistics
- Head First Physics
- Head First Programming
- Head First Ruby on Rails



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

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

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

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

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

She's traveled all over the world to work with different software teams and build all kinds of cool projects.

She loves traveling, watching Bollywood movies, reading the occasional comic book, playing PS3

games, and hanging out with her huge siberian cat, Sascha.

NOTE

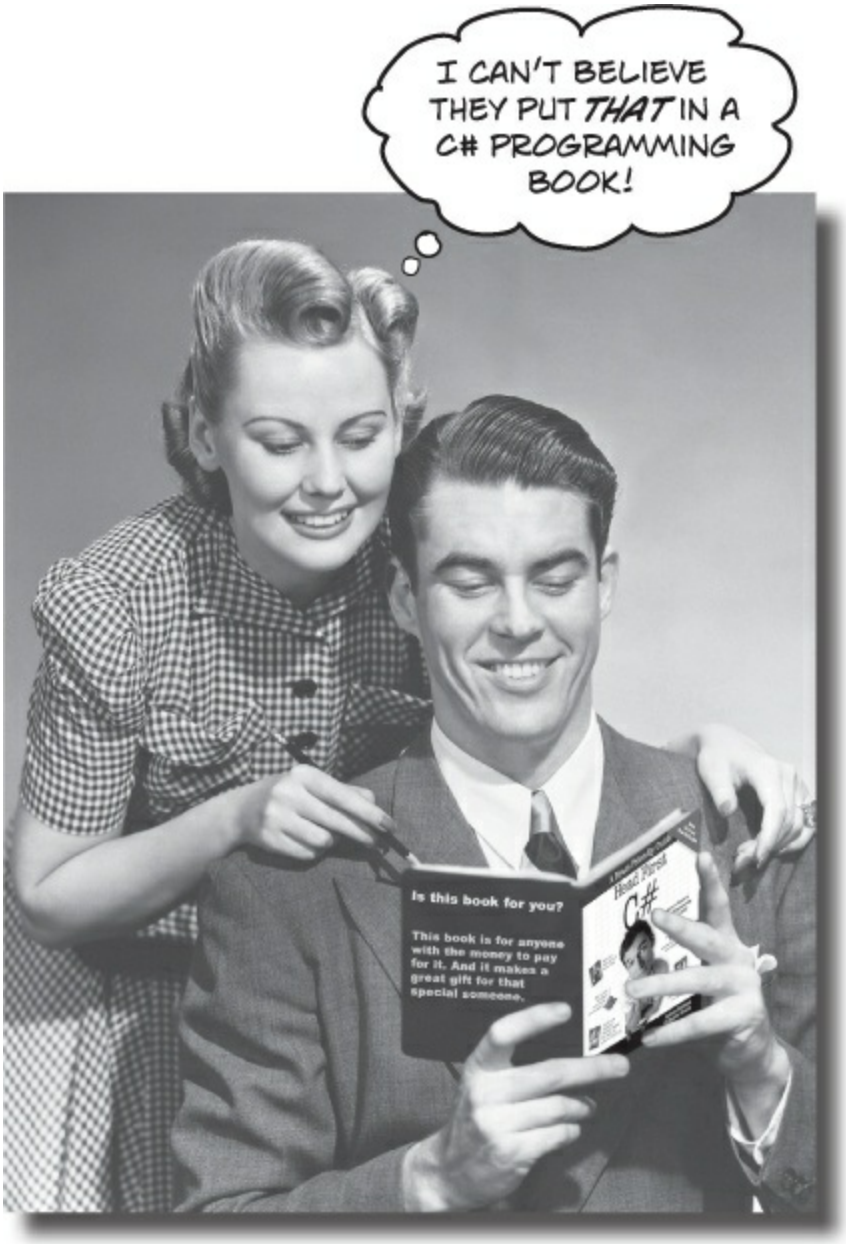
Jenny and Andrew have been building software and writing about software engineering together since they first met in 1998. Their first book, Applied Software Project Management, was published by O'Reilly in 2005. Other Stellman and Greene books for O'Reilly include Beautiful Teams (2009), and their first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

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

Check out their blog, Building Better Software: http://www.stellman-greene.com

Follow @AndrewStellman and @JennyGreene on Twitter

How to Use this Book: Intro



NOTE

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

Who is this book for?

If you can answer “yes” to all of these:

- ❶ Do you want to **learn** C#?
  Do you know another programming language, and now you need to ramp up on C#?
- ❷ Do you like to tinker — do you learn by doing, rather than just reading?
  Are you already a good C# developer, but you want to learn more about XAML, Model-View-ViewModel (MVVM), or Windows Store app development?
- ❸ Do you prefer **stimulating dinner party conversation** to **dry**, **dull**, **academic lectures**?
  Do you want to get practice writing lots of code?

this book is for you.

Who should probably back away from this book?

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

- ❶ Does the idea of doing projects and building programs make you bored and a little twitchy?
- ❷ Are you a really advanced C++ programmer looking for a dry reference book?
- ❸ Are you **afraid to try something different**? Would you rather have a root canal than mix stripes with plaid? Do you believe that a technical book can't be serious if C# concepts are anthropomorphized?

  If so, then lots of people just like you have used this book to do exactly those things!

> **NOTE**
>
> No programming experience is required to use this book... just curiosity and interest! Thousands of beginners with no programming experience have already used Head First C# to learn to code. That could be you!

this book is not for you.

> **READ THIS!**
>
> **ARE YOU USING WINDOWS 7 OR EARLIER? THEN THIS BOOK IS FOR YOU!**
>
> We need to keep our book up to date with the latest technology, so we based many projects in this book on Windows 8.1, the latest version of Microsoft Windows available at press time. However, **we *worked really hard* to support previous versions of Windows**. We included a special appendix with replacement pages for some of the book's projects. We did our best to minimize the amount of page flipping required. There's a complete replacement for most of Chapter 1, so you won't need to flip back to the book at all for the first project. Then there are just five replacement pages for Chapter 2. After that, the *you'll be able to use any version of Windows* (and even old versions of Visual Studio!) until you get to Chapter 10.

> **NOTE**
>
> Many readers have used this book Windows 7, Windows 2003, or other versions of Windows. We'll give you all the information you need to use any version of Windows at the end of this introduction.

We know what you're thinking.

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

"What's with all the graphics?"

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

And we know what your *brain* is thinking

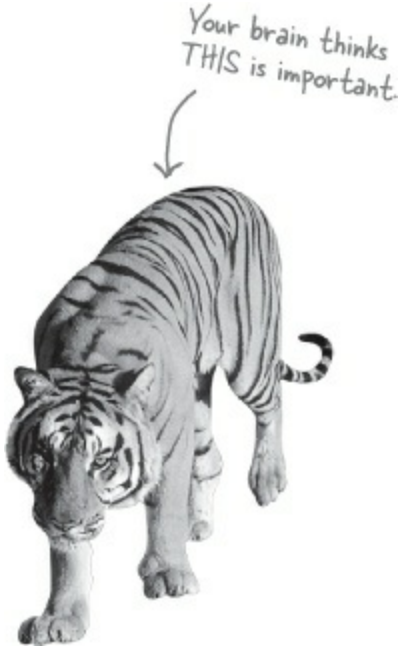
Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

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

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

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

And that's how your brain knows...

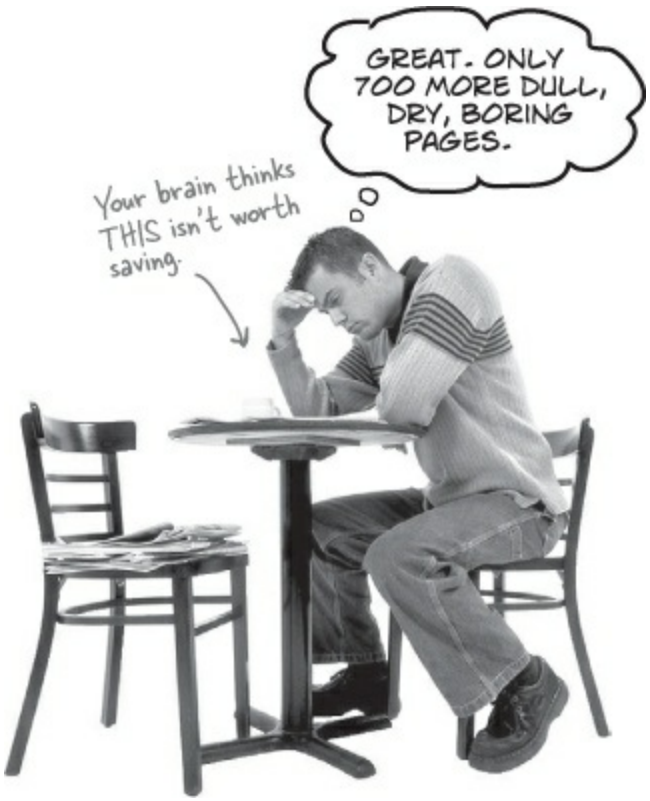


This must be important! Don't forget it!

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

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

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

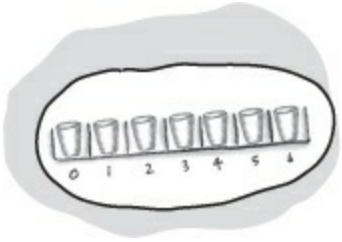


WE THINK OF A "HEAD FIRST" READER AS A LEARNER.

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

Some of the Head First learning principles:

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.



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

Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And

for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

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

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the...?", and the feeling of "I Rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm more technical than thou" Bob from engineering *doesn't*.

Metacognition: thinking about thinking

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

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

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

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

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

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

The faster way is to do ***anything that increases brain activity,*** especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words

within the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to makes sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

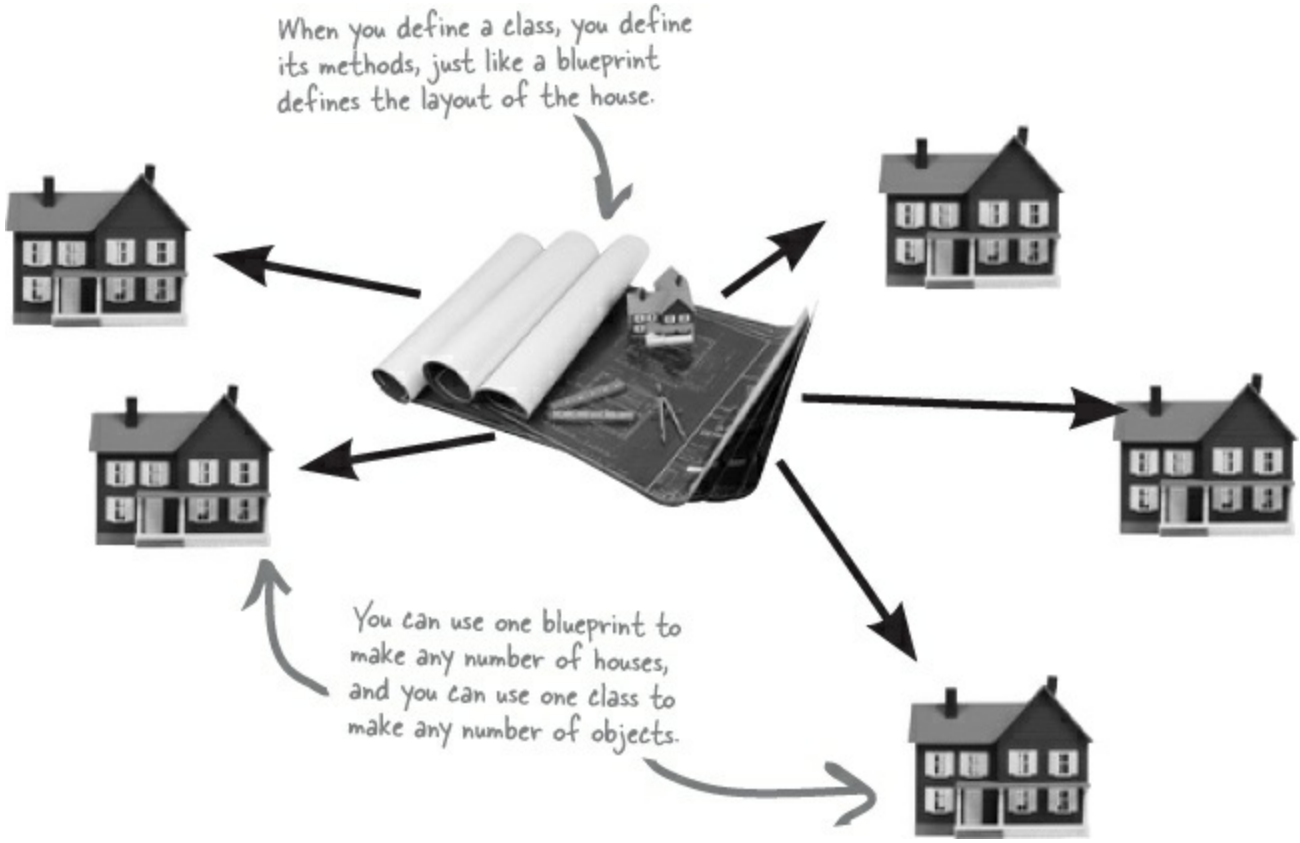
A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the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

But pictures and conversational style are just the beginning.

Here's what WE did

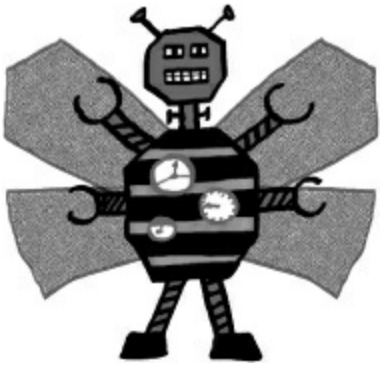
We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within*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

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



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

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



We included dozens of ***activities***, because your brain is tuned to learn and remember more when you

do things than when you *read* about things. And we made the paper puzzles and code exercises challenging-yet-do-able, because that's what most people prefer.

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

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

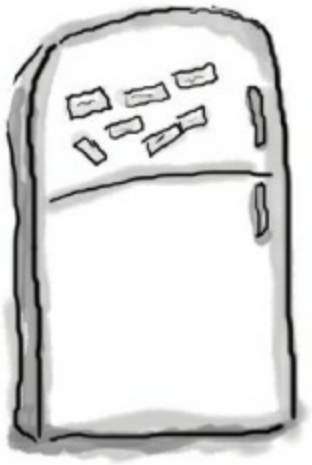
And we included ***stories*** and exercises that present ***more than one point of view,*** because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

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

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

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

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



Cut this out and stick it on your refrigerator.

- **❶ Slow down. The more you understand, the less you have to memorize.**
  Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.
- **❷ Do the exercises. Write your own notes.**
  We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.
- **❸ Read the "There are No Dumb Questions"**
  That means all of them. They're not optional sidebars — ***they're part of the core content!*** Don't skip them.
- **❹ Make this the last thing you read before bed. Or at least the last challenging thing.**
  Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.
- **❺ Drink water. Lots of it.**
  Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.
- **❻ Talk about it. Out loud.**
  Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.
- **❼ Listen to your brain.**
  Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.
- **❽ Feel something.**

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

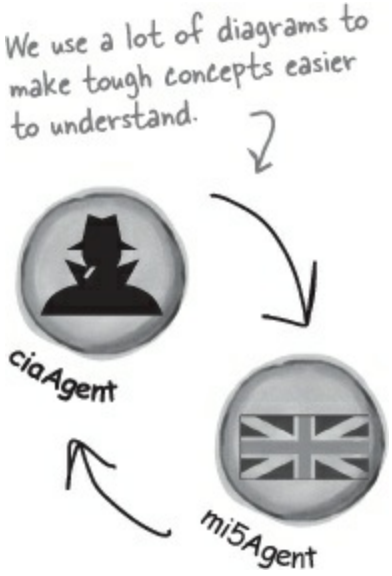
- **❾ Write a lot of software!**
  There's only one way to learn to program: **writing a lot of code**. And that's what you're going to do throughout this book. Coding is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has exercises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

Read me

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

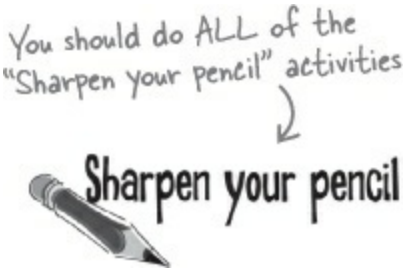
The activities are NOT optional.

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



The redundancy is intentional and important.

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



Do all the exercises!

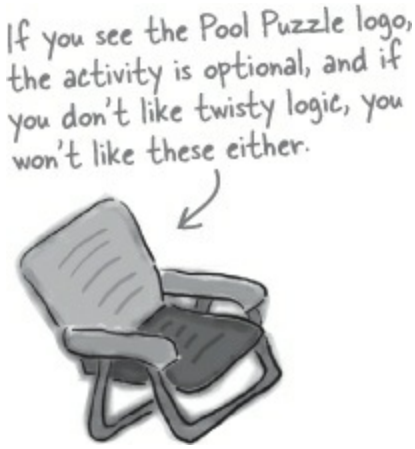
The one big assumption that we made when we wrote this book is that you want to learn how to program in C#. So we know you want to get your hands dirty right away, and dig right into the code. We gave you a lot of opportunities to sharpen your skills by putting exercises in every chapter. We've labeled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

try to solve the problem first.

We've also placed all the exercise solutions' source code on the web so you can download it. You'll find it at `http://www.headfirstlabs.com/books/hfcsharp/`

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

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



What version of Windows are you using?

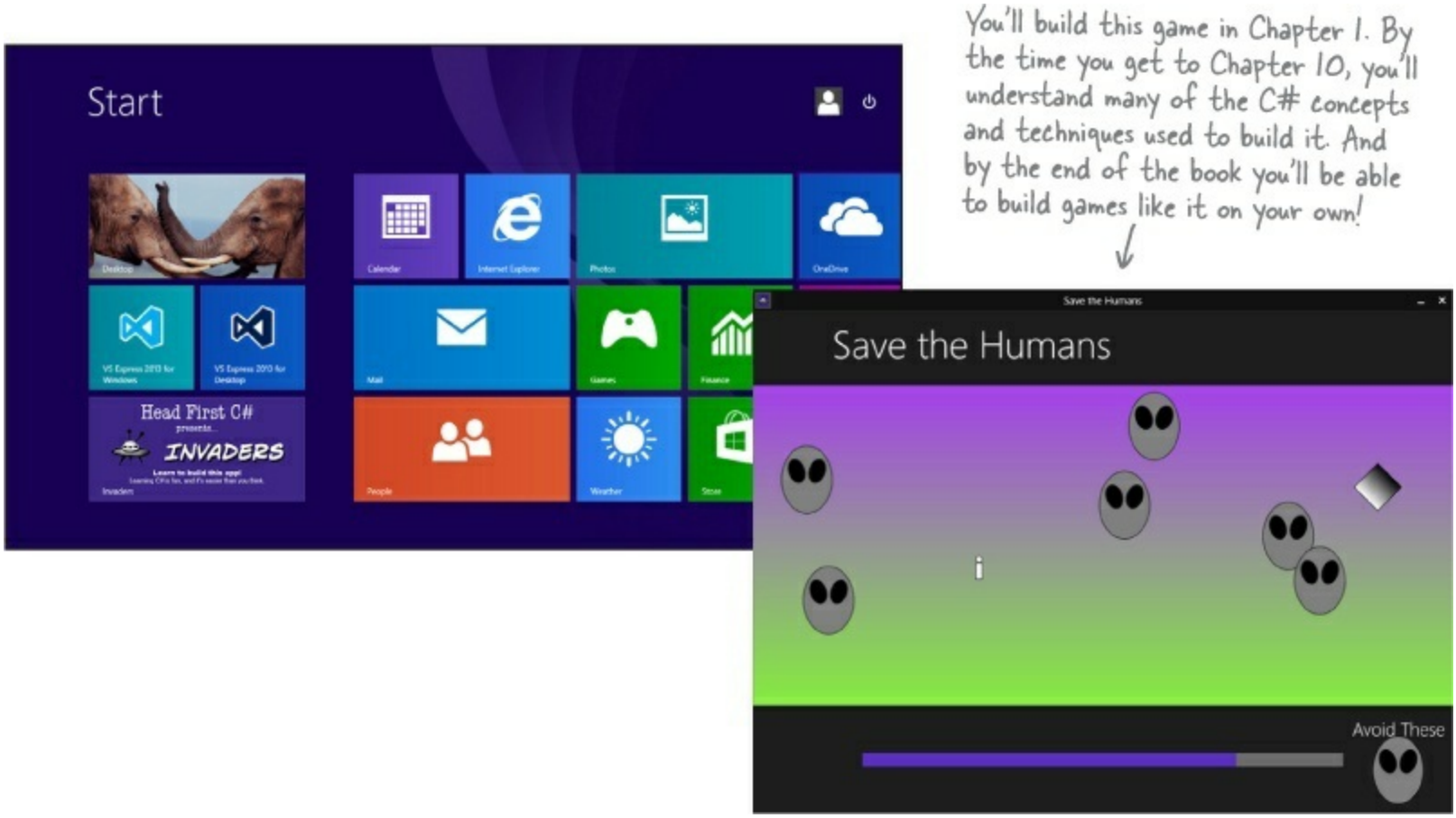
We wrote this book using **Visual Studio Express 2013 for Windows** and **Visual Studio Express 2013 for Windows Desktop**. All of the screenshots that you see throughout the book were taken from those two editions, so we recommend that you use them. You can also use Visual Studio 2013 Professional, Premium, Ultimate or Test Professional editions, but you'll see some small differences (but nothing that will cause problems with the coding exercises).

*We built this book using **Windows 8.1**, the latest version of Windows available when we went to press. We'll refer to it as "Windows 8" throughout the book. Visual Studio 2013 requires Windows 8.1, which is available as a free Windows Store update to Windows 8.*

Visual Studio 2013 can be installed on the same computer as other editions or older versions of Visual Studio without causing any problems.

Using Windows 8 or later? Then you'll start with Windows Store apps

Windows Store apps are programs built with the latest Microsoft technology. They get their name because they can be downloaded and sold through the Windows Store.



In the first two chapters, you'll build Windows Store apps, starting with a game called *Save the Humans*. This will be your first experience with Visual Studio, and will teach you the basic mechanics of creating a user interface and entering code — core skills that you'll use throughout the book as you use it to build many different projects.

NOTE

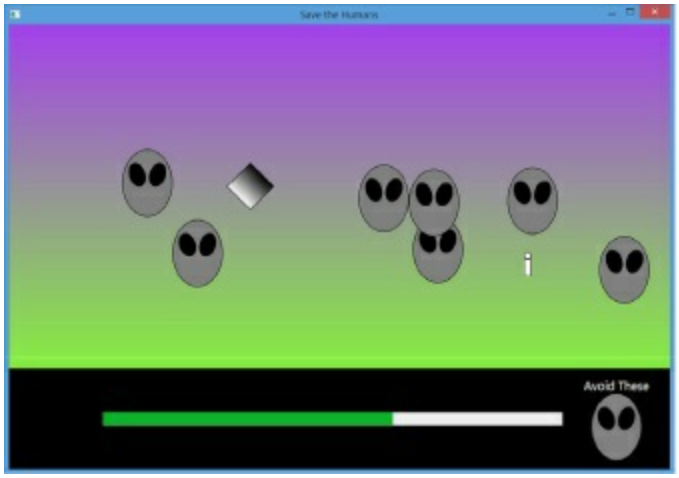
Some of the code in this book may not work with earlier or later editions of Visual Studio! But you can always download Visual Studio 2013 from Microsoft's website.

The screenshots in this book match Visual Studio 2013 Express Edition, the latest free version available at the time of this printing. We'll keep future printings up to date, but Microsoft typically makes older versions available for download. It's possible that some of the code for Windows Store apps may not work with future versions of Visual Studio. If the links on the next page don't work, search Microsoft.com for "Visual Studio 2013 Express update 3 download" — and also check the forum on http://headfirstlabs.com/hfcsharp.

Don't have Windows 8 or VS2013 yet? No problem — you'll start with WPF apps

There's another technology for building desktop apps called **Windows Presentation Foundation** (WPF) that works with previous versions of Windows. It's *very* important to us that you can use our book with Windows 7, Windows 2003, or other previous versions of Windows! **If you're one of these readers, we *worked very hard* to make our book easy for you to use**. We added an Appendix with alternate versions of *almost* every Windows Store project in this book that you'll build and run as WPF desktop applications. And if you're using an older version of Visual Studio, you'll be able to use it to build WPF apps too. Here's what you need to do:

- Flip to Appendix B, the **WPF Learner's Guide to Head First C#**. You'll find a complete replacement for the *Save the Humans* project in Chapter 1 and five replacement pages for Chapter 2 (which are all you need!).
- After that, Chapter 3 through Chapter 9 the first two labs do not require Windows 8 at all, because Windows Forms and Console applications work on all versions of Windows. You'll even be able to build them using Visual Studio 2012 (and even 2010 or 2008), although the Visual Studio screenshots may differ a bit from the book.
- For the rest of the book, you'll use the replacement pages in the Appendix to build WPF desktop apps instead of Windows Store apps. That way you'll still build lots of projects and learn the same important C# concepts.
- You can download a PDF of the appendix from the book's website (*http://headfirstlabs.com/hfcsharp*) in case you want to print out the replacement pages.
- And even if you're running the latest version of Windows, you should still have a look at the WPF Learner's Guide! Building the same projects with two different technologies is an excellent way to get C# into your brain.



WPF applications run in windows on the desktop, and work with older versions of Windows. Here's the WPF version of the game that you'll build in Chapter 1.

If you're running Windows 7 or earlier, you can still build all of the Windows Forms, Console, and WPF applications in this book.

If these download links don't work, go to microsoft.com and search for "Visual Studio Express 2013 Download"

Microsoft regularly releases updates to Visual Studio, and sometimes they make minor changes to its look and feel between updates. The screenshots in this book were taken from Visual Studio 2013 with Update 3. Here are direct links to the download pages for Visual Studio 2013 Express with Update 3:

VS2013 Express for Windows with Update 3: http://www.microsoft.com/en-us/download/details.aspx?id=43729

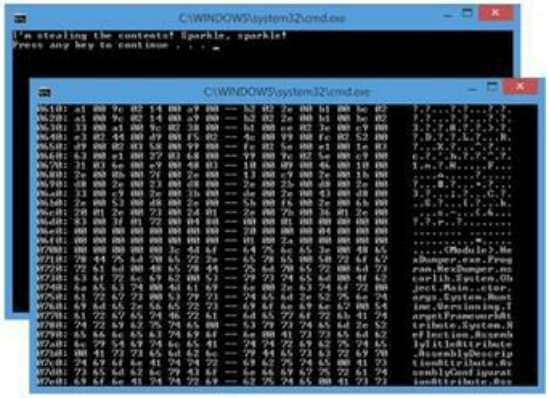
VS2013 Express for Windows Desktop with Update 3: http://www.microsoft.com/en-us/download/details.aspx?id=43733

The Visual Studio home page also has many useful download links: http://www.microsoft.com/visualstudio

You'll move on to create desktop applications

Chapter 1 and Chapter 2 focus on creating Windows Store (or WPF) apps. After that, you'll switch gears and create two different kinds of **desktop applications**. In the following few chapters you'll build **Windows Forms applications** and design user interfaces that are based on desktop windows. And later in the book you'll create **console applications** that use a command window for input and output. You'll mix Windows Store (or WPF) apps back in starting in Chapter 10.

Windows Forms applications are based on an older Microsoft technology. They use windows that pop up on the Windows desktop. They're a great tool for learning and experimenting with C#.

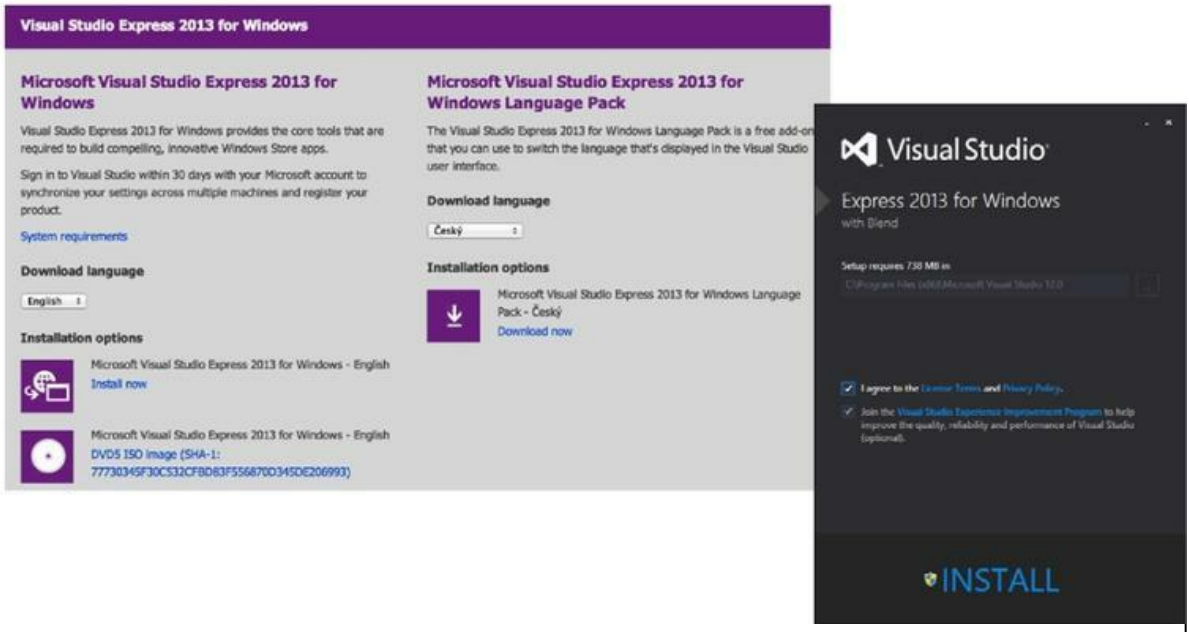


Console applications are text-only programs that don't display a graphical user interface.

SETTING UP VISUAL STUDIO 2013 EXPRESS EDITIONS

- You can download **Visual Studio Express 2013** for Windows for *free* from Microsoft's website. It installs cleanly alongside other editions of VS2013, as well as previous versions. You can download the edition from the **Visual Studio home page**.

Click the "Install Now" link to launch the web installer, which automatically downloads and installs Visual Studio.



NOTE

You'll also need to generate a product key, which is free for the Express editions (but requires you to create a Microsoft.com account).

- Once you've got it installed, you'll need to do the same thing for **Visual Studio Express 2013 for Windows Desktop**. You'll use this version to create Windows Forms Application and Console Application projects.
- If you have Visual Studio 2013 **Professional, Premium, or Ultimate** installed, then you can create all of the different types of applications with any of those editions. But you'll be able to do all of the projects in this book using the free editions.

The technical review team

Lisa Kellner

Rebeca Dunn-Krahn

Chris Burrows

Johnny Halife

David Sterling

NOTE

Not pictured (but just as awesome are the reviewers from previous editions): Joe Albahari, Jay Hilyard, Aayam Singh, Theodore, Peter Ritchie, Bill Meitelski Andy Parker, Wayne Bradney, Dave Murdoch, Bridgette Julie Landers, Nick Paldino, David Sterling. Special thanks to readers Alan Ouellette, Terry Graham, and our other readers who let us know about issues that slipped through QC. Thanks!!

Technical Reviewers:

The book you're reading has very few errors in it, and give a lot of credit for its high quality to some great technical reviewers. We're really grateful for the work that they did for this book — we would have gone to press with errors (including one or two big ones) had it not been for the most kick-ass review team EVER....

First of all, we really want to thank **Lisa Kellner** — this is our ninth (!) book that she's reviewed for us, and she made a huge difference in the readability of the final product. Thanks, Lisa! And special thanks to **Chris Burrows, Rebeca Dunn-Krahn,** and **David Sterling** for their enormous amount of technical guidance, and to **Joe Albahari** and **Jon Skeet** for their really careful and thoughtful review of the first edition, and **Nick Paladino** who did the same for the second edition.

Chris Burrows is a developer at Microsoft on the C# Compiler team who focused on design and implementation of language features in C# 4.0, most notably dynamic.

Rebeca Dunn-Krahn is a founding partner at Semaphore Solutions, a custom software shop in Victoria, Canada, that specializes in .NET applications. She lives in Victoria with her husband Tobias, her children, Sophia and Sebastian, a cat, and three chickens.

David Sterling has worked on the Visual C# Compiler team for nearly three years.

Johnny Halife is a Chief Architect & Co-Founder of Mural.ly (*http://murally.com*), a web start-up that allows people to create murals: collecting any content inside them and organizing it in a flexible and organic way in one big space. Johnny's a specialist on cloud and high-scalability solutions. He's also a passionate runner and sports fan.

Acknowledgments

Our editor:

We want to thank our editor, **Courtney Nash**, for editing this book. Thanks!

The O'Reilly team:

There are so many people at O'Reilly we want to thank that we hope we don't forget anyone. Special Thanks to production editor **Melanie Yarbrough**, indexer **Ellen Troutman-Zaig**, **Rachel Monaghan** for her sharp proofread, **Ron Bilodeau** for volunteering his time and preflighting expertise, and for offering one last sanity check — all of whom helped get this book from production to press in record time. And as always, we love **Mary Treseler**, and can't wait to work with her again! And a big shout out to our other friends and editors, **Andy Oram**, **Mike Hendrickson**, **Laurie Petryki**, **Tim O'Reilly**, and **Sanders Kleinfeld**. And if you're reading this book right now, then you can thank the greatest publicity team in the industry: **Marsee Henon**, **Sara Peyton**, and the rest of the folks at Sebastopol.

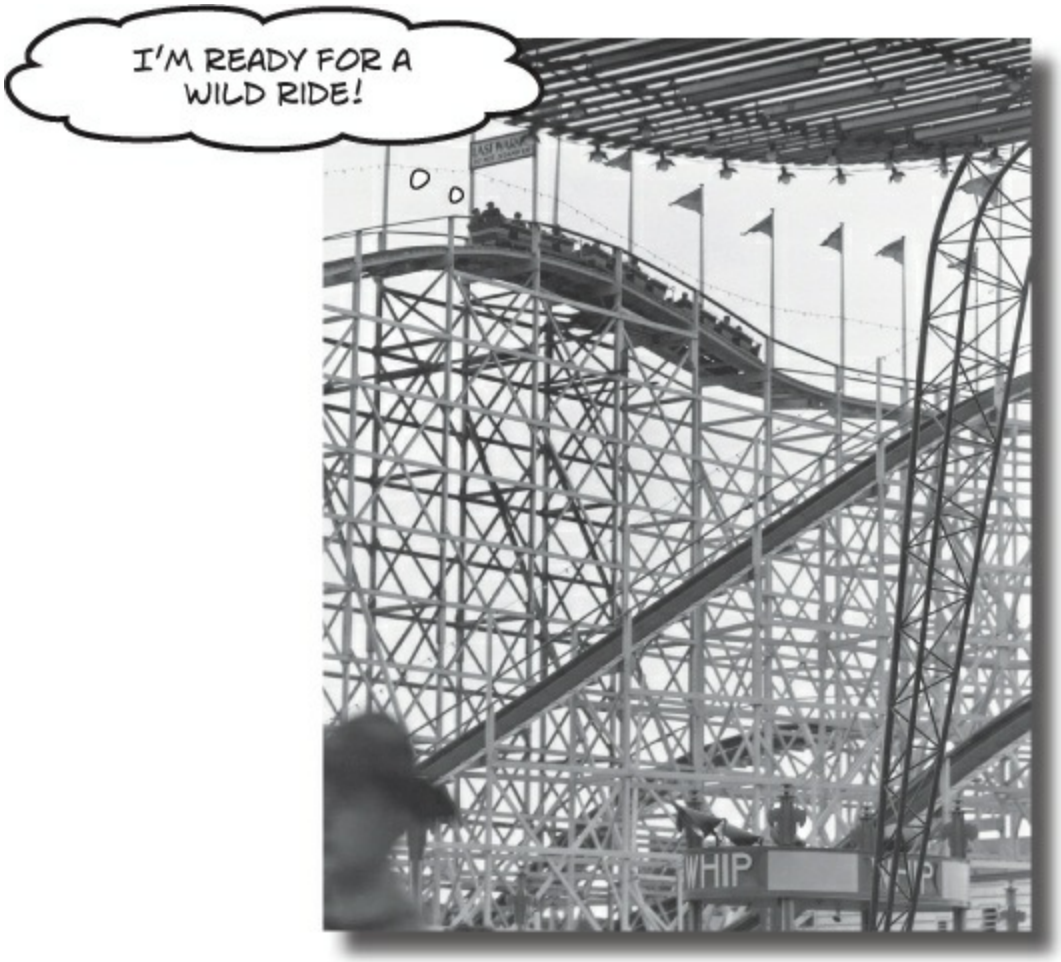
Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://safaribooksonline.com*.

Chapter 1. Start Building with C#: Build something cool, fast!



Want to build great apps really fast?

With C#, you've got a **great programming language** and a **valuable tool** at your fingertips. With the **Visual Studio IDE**, you'll never have to spend hours writing obscure code to get a button working again. Even better, you'll be able to **build really cool software**, rather than remembering which bit of code was for the *name* of a button, and which one was for its *label*. Sound appealing? Turn the page, and let's get programming.

Why you should learn C#

C# and the Visual Studio IDE make it easy for you to get to the business of writing code, and writing it fast. When you're working with C#, the IDE is your best friend and constant companion.

NOTE

The IDE — or Visual Studio Integrated Development Environment — is an important part of working in C#. It's a program that helps you edit your code, manage your files, and submit your apps to the Windows Store.

Here's what the IDE automates for you...

Every time you want to get started writing a program, or just putting a button on a page, your program needs a whole bunch of repetitive code.

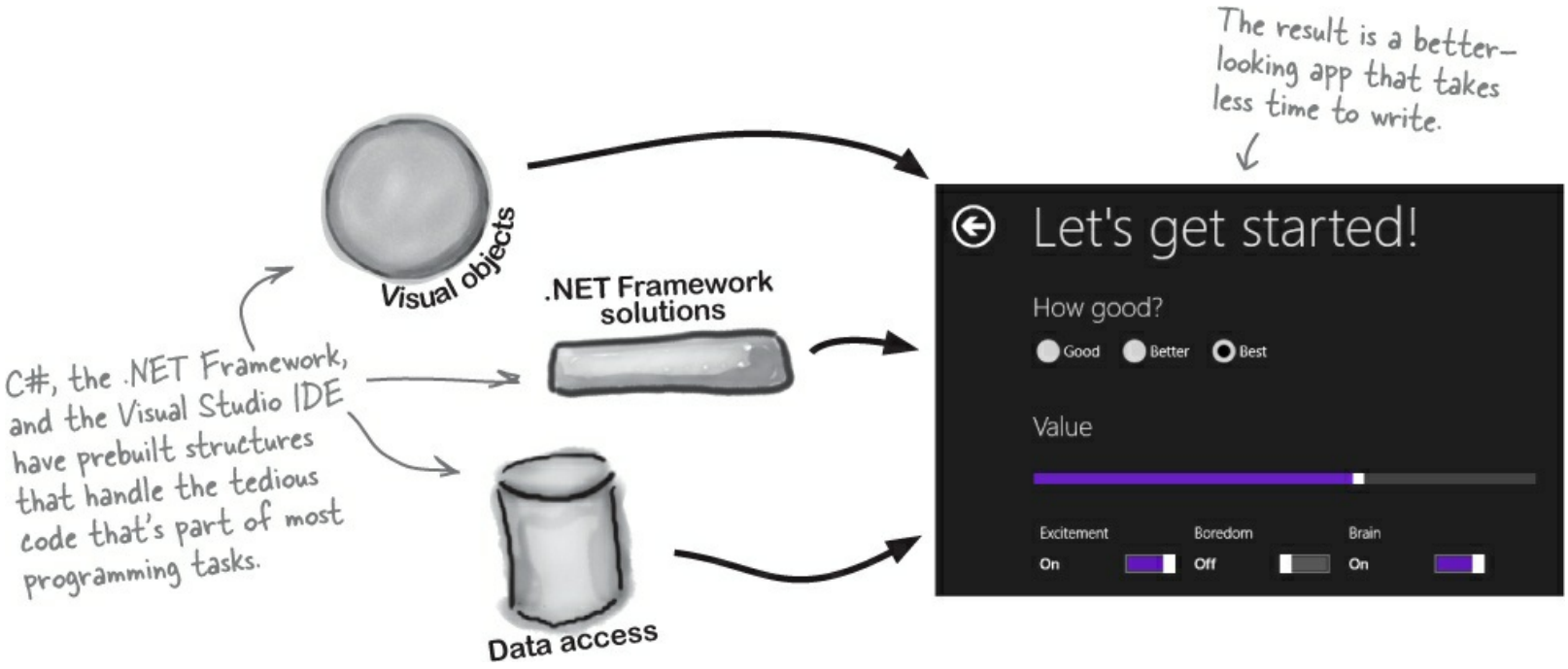
```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
namespace A_New_Program
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

```
private void InitializeComponent()
{
  this.button1 = new System.Windows.Forms.Button();
  this.SuspendLayout();
  //
  // button1
  //
  this.button1.Location = new System.Drawing.Point(105, 56);
  this.button1.Name = "button1";
  this.button1.Size = new System.Drawing.Size(75, 23);
  this.button1.TabIndex = 0;
  this.button1.Text = "button1";
  this.button1.UseVisualStyleBackColor = true;
  this.button1.Click += new System.EventHandler(this.button1_Click);
  //
  // Form1
  //
  this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 16F);
  this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
  this.ClientSize = new System.Drawing.Size(292, 267);
  this.Controls.Add(this.button1);
  this.Name = "Form1";
  this.Text = "Form1";
  this.ResumeLayout(false);
}
```

It takes all this code just to draw a button in a window. Adding a bunch of visual elements to a page could take 10 times as much code.

What you get with Visual Studio and C#...

With a language like C#, tuned for Windows programming, and the Visual Studio IDE, you can focus on what your program is supposed to **do** immediately:



C# and the Visual Studio IDE make lots of things easy

When you use C# and Visual Studio, you get all of these great features, without having to do any extra work. Together, they let you:

- ❶ **Build an application, FAST.** Creating programs in C# is a snap. The language is flexible and easy to learn, and the Visual Studio IDE does a lot of work for you automatically. You can leave mundane coding tasks to the IDE and focus on what your code should accomplish.
- ❷ **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 writing a graphical user interface entirely from scratch.
- ❸ **Build visually stunning programs.** When you combine C# with XAML, the visual markup language for designing user interfaces, 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.
- ❹ **Focus on solving your REAL problems.** The IDE does a lot for you, but ***you*** are still in control of what you build with C#. The IDE lets you just focus on your program, your work (or fun!), and your users. It handles all the grunt work for you:

  Keeping track of all your project files
  Making it easy to edit your project's code
  Keeping track of your project's graphics, audio, icons, and other resources
  Helping you manage and interact with your data

  All this means you'll have all the time you would've spent doing this routine programming to put into **building and sharing killer apps**.

> **NOTE**
>
> You're going to see exactly what we mean next.

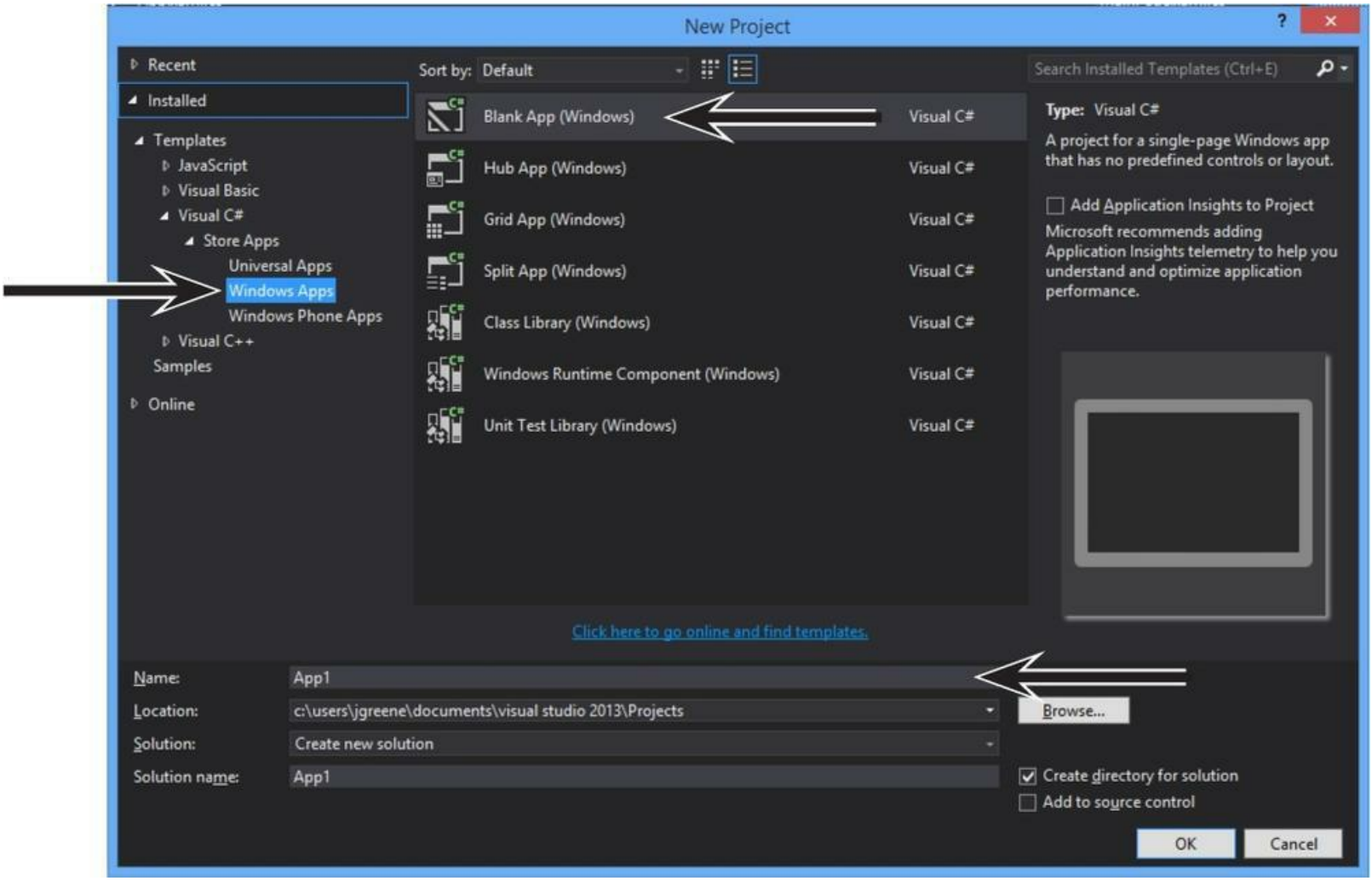
\
apps

What you do in Visual Studio...

Go ahead and start up Visual Studio 2013 for Windows, if you haven't already. Skip over the start page and select New Project from the **File** menu. There are several project types to choose from. Expand **Visual C#→Windows Store→Windows App**, and select **Blank App (Windows)**. The IDE will create a folder called *Visual Studio 2013* in your *Documents* folder, and put your applications in a *Projects* folder under it (you can use the Location box to change this).

NOTE

If you don't see this option, you might be running Visual Studio 2013 for Windows Desktop. You'll need to exit that IDE and launch Visual Studio Express 2013 for Windows.



WATCH IT!

Things may look a bit different in your IDE.

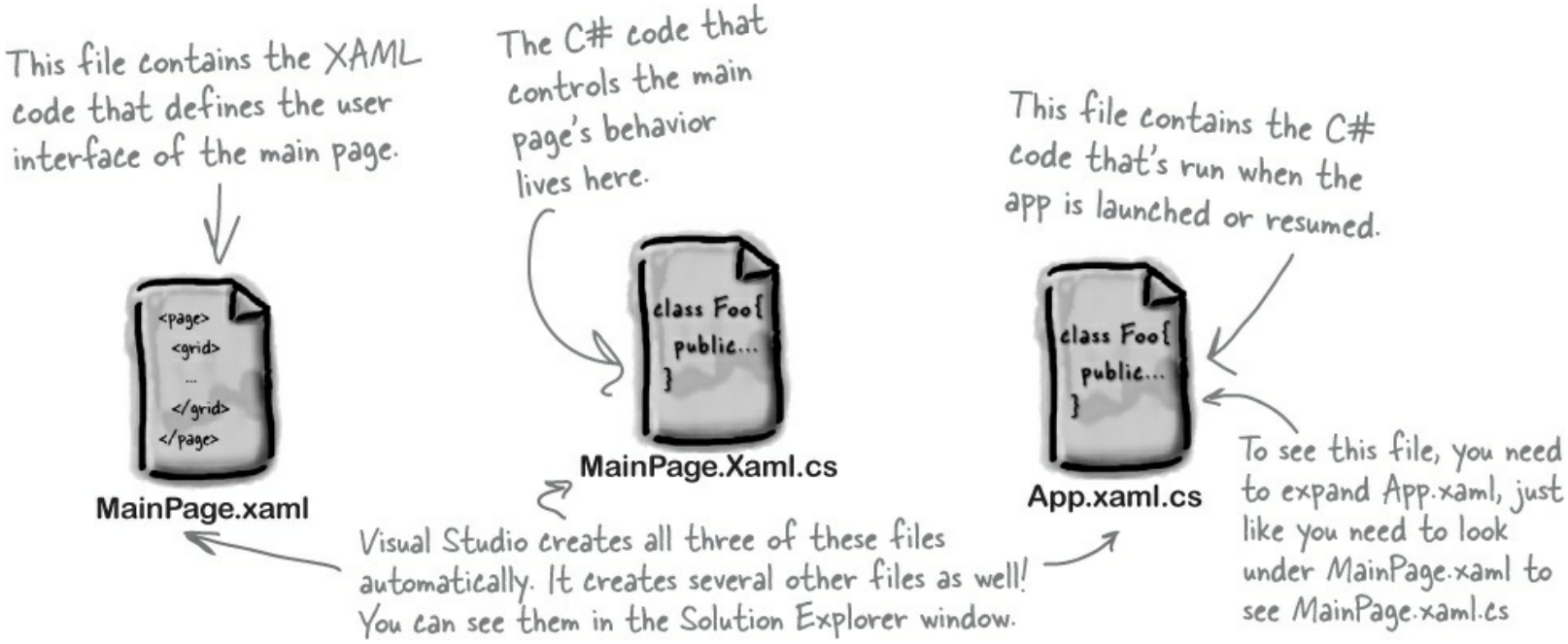
*This is what the New Project window looks like in **Visual Studio 2013 Express for Windows**. If you're using the Professional or Team Foundation edition, it might be a bit different. But don't worry, everything still works exactly the same.*

What Visual Studio does for you...

As soon as you save the project, the IDE creates a bunch of files, including *MainPage.xaml*, *MainPage.Xaml.cs*, and *App.xaml.cs*, when you create a new project. It adds these to the Solution Explorer window, and by default, puts those files in the *Projects\App1\App1* folder.

NOTE

Make sure that you save your project as soon as you create it by selecting Save All from the File menu — that'll save all of the project files out to the folder. If you select Save, it just saves the one you're working on.

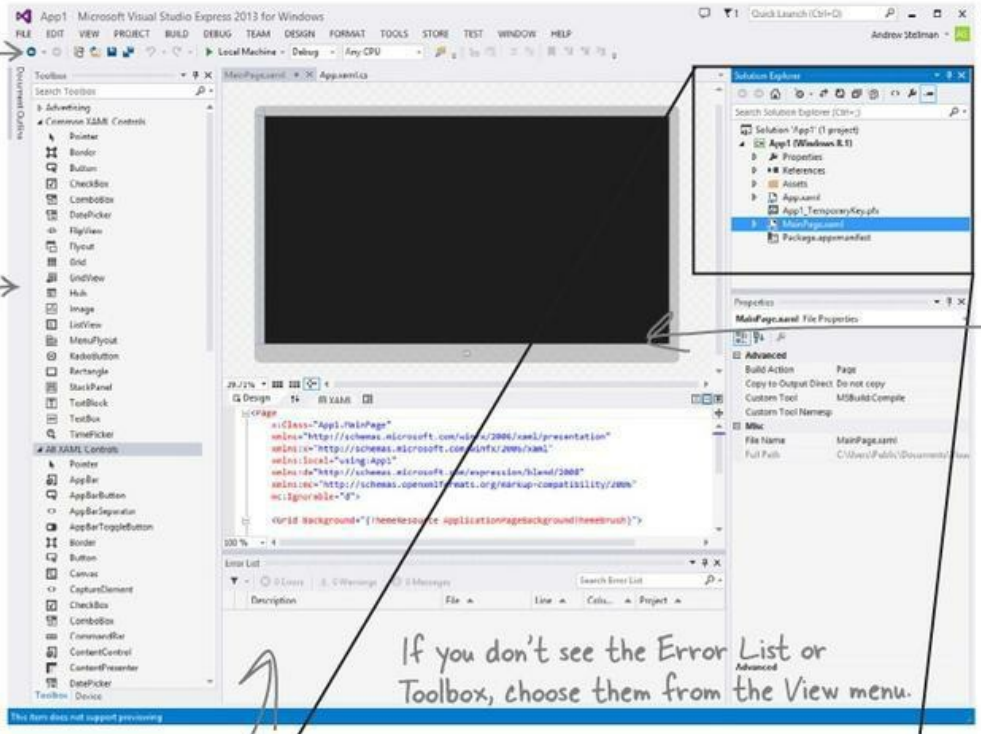


SHARPEN YOUR PENCIL

Just a couple more steps and your screen will match the picture below. First, **open *MainPage.xaml*** by double-clicking on it in the Solution Explorer window. Next, select the **Light color theme from the Options menu**. Finally, make sure you open the Toolbox and Error List windows by **choosing them from the View menu**. You should be able to figure out the purpose of many of these windows and files based on what you already know. Then, in each of the blanks, try to fill in an annotation saying what that part of the IDE does. We've done one to get you started. See if you can guess what all of these things are for.

This toolbar has buttons that apply to what you're currently doing in the IDE.

We've blown up this window below so you have more room.

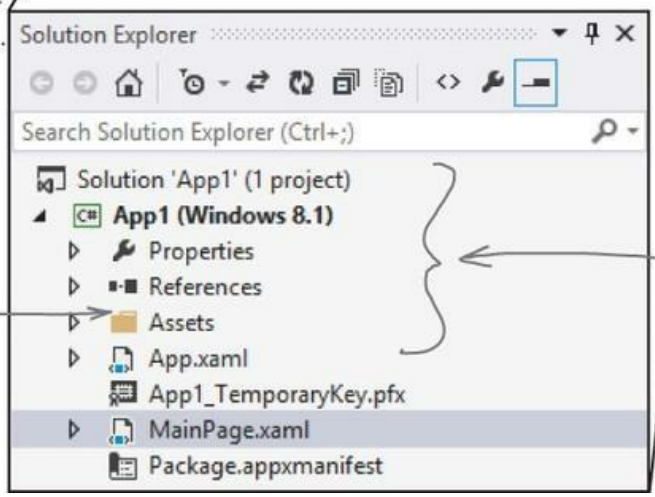


The designer lets you edit the user interface by dragging controls onto it.

If you don't see the Error List or Toolbox, choose them from the View menu.

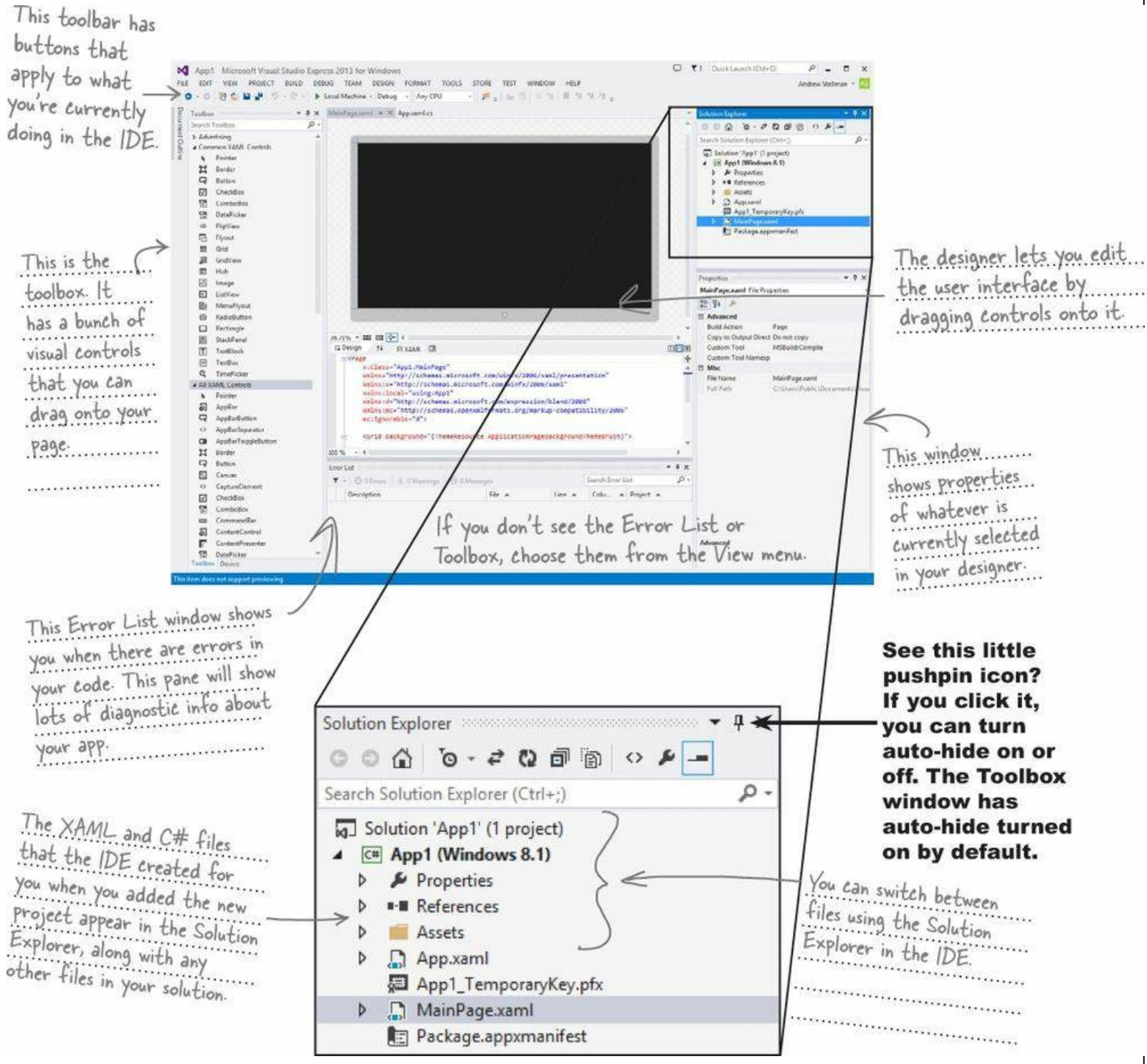
The screenshot on page 4 is in the Dark color theme.

We switched to the Light color theme because it's easier to see light screenshots in a book. If you like it, pick "Options..." from the Tools menu, expand Environment, and click on General to change it (feel free to change back).



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 you should have been able to figure out the basics of what each window and section of the IDE is used for.



THERE ARE NO DUMB QUESTIONS

Q: **Q: So if the IDE writes all this code for me, is learning C# just a matter of learning how to use the IDE?**

A: **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 on your pages. 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: **Q: What if the IDE creates code I don't want in my project?**

A: **A:** You can change 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-touse interface in the IDE.

Q: **Q: Is it OK that I downloaded and installed Visual Studio Express? 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: **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 Express and the other editions aren't going to get in the way of writing C# and creating fully functional, complete applications.

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

A: **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 full-page Windows Store apps. XAML is based on XML (which you'll also learn about later in the book), 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"/>
```

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 the different XAML tags throughout the book.

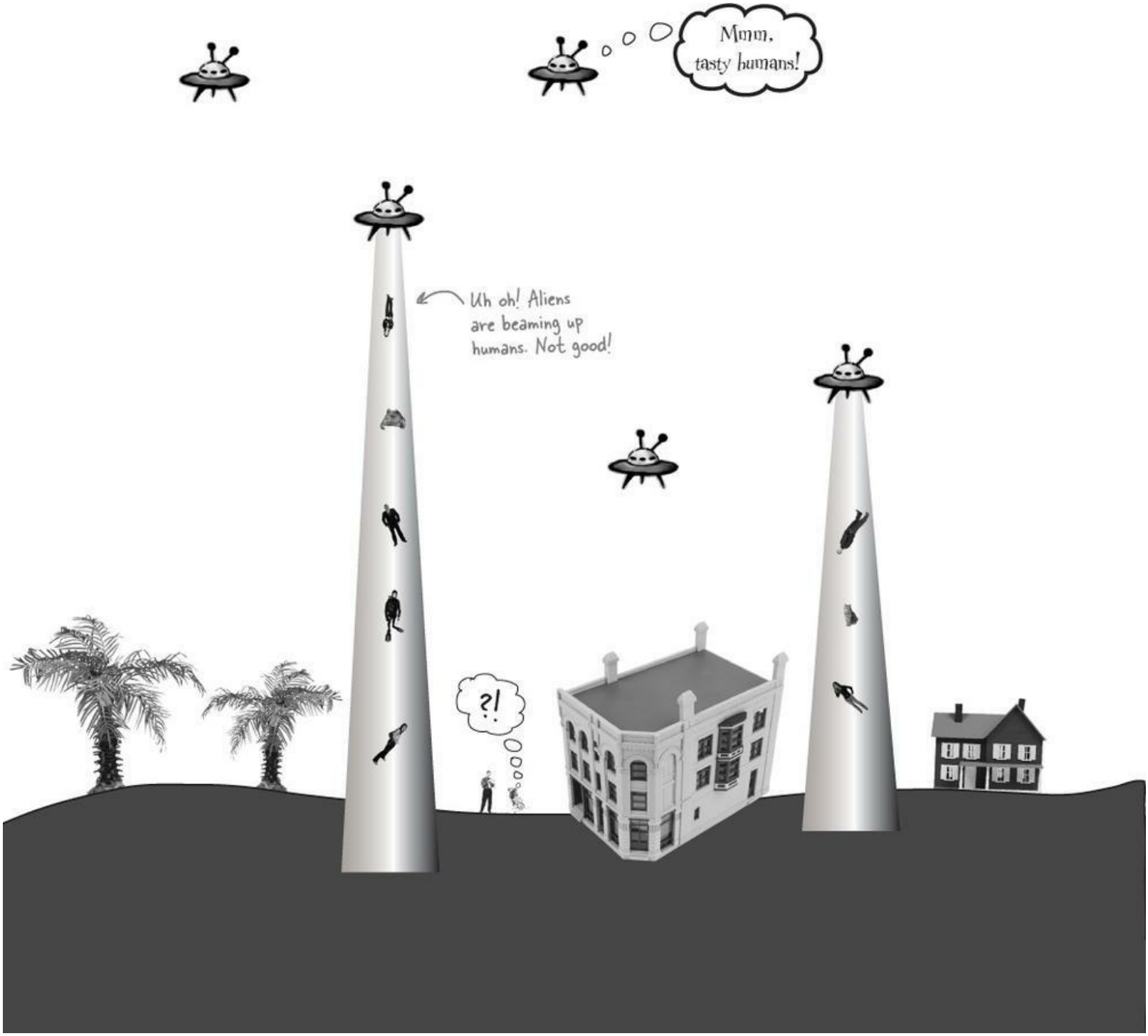
Q: **Q: I'm looking at the IDE right now, but my screen doesn't look like yours! It's missing some of the windows, and others are in the wrong place. What gives?**

A: **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 you can use the View→Other Windows menu to make your screen look just like the ones in this chapter.

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

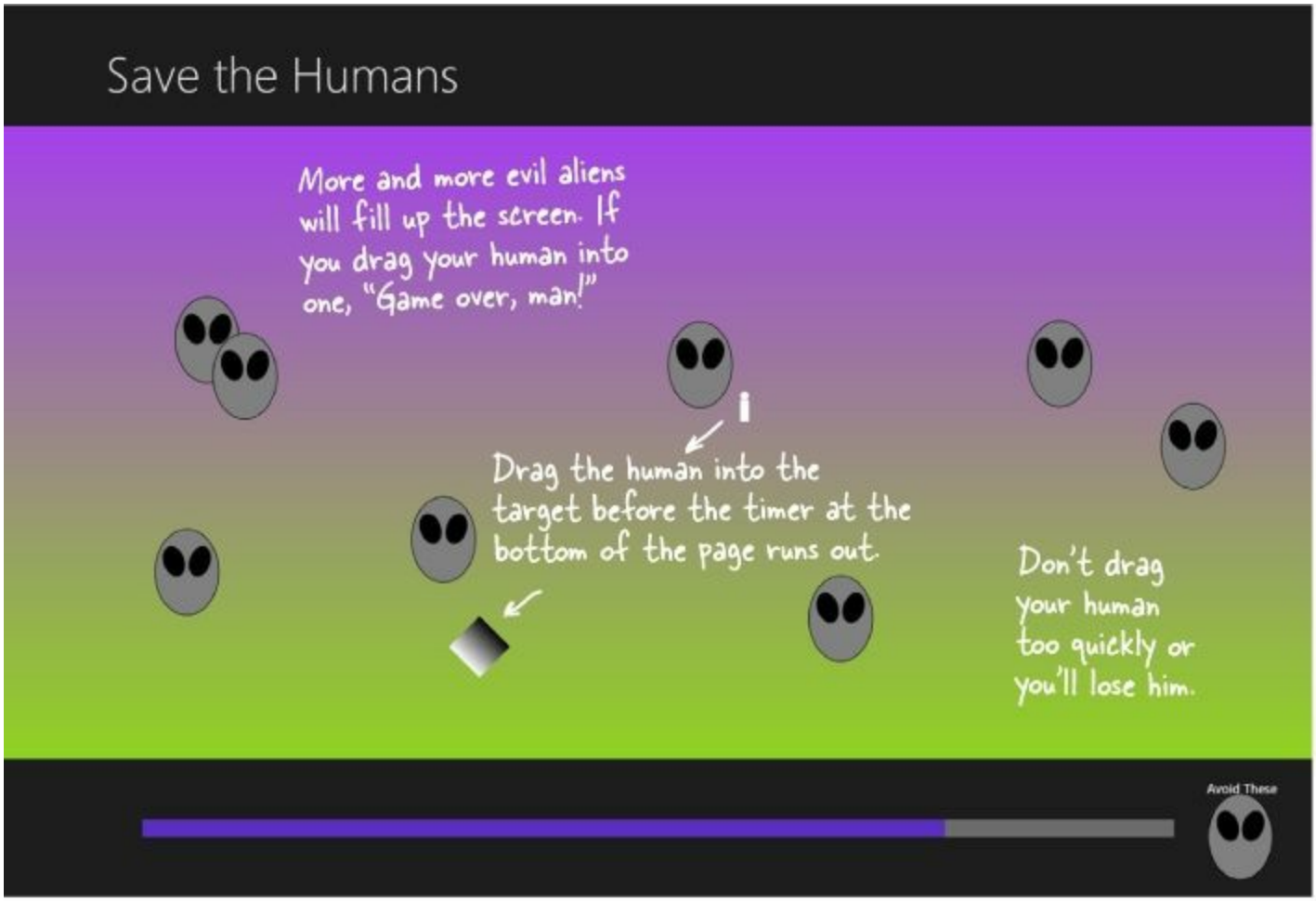
Aliens attack!

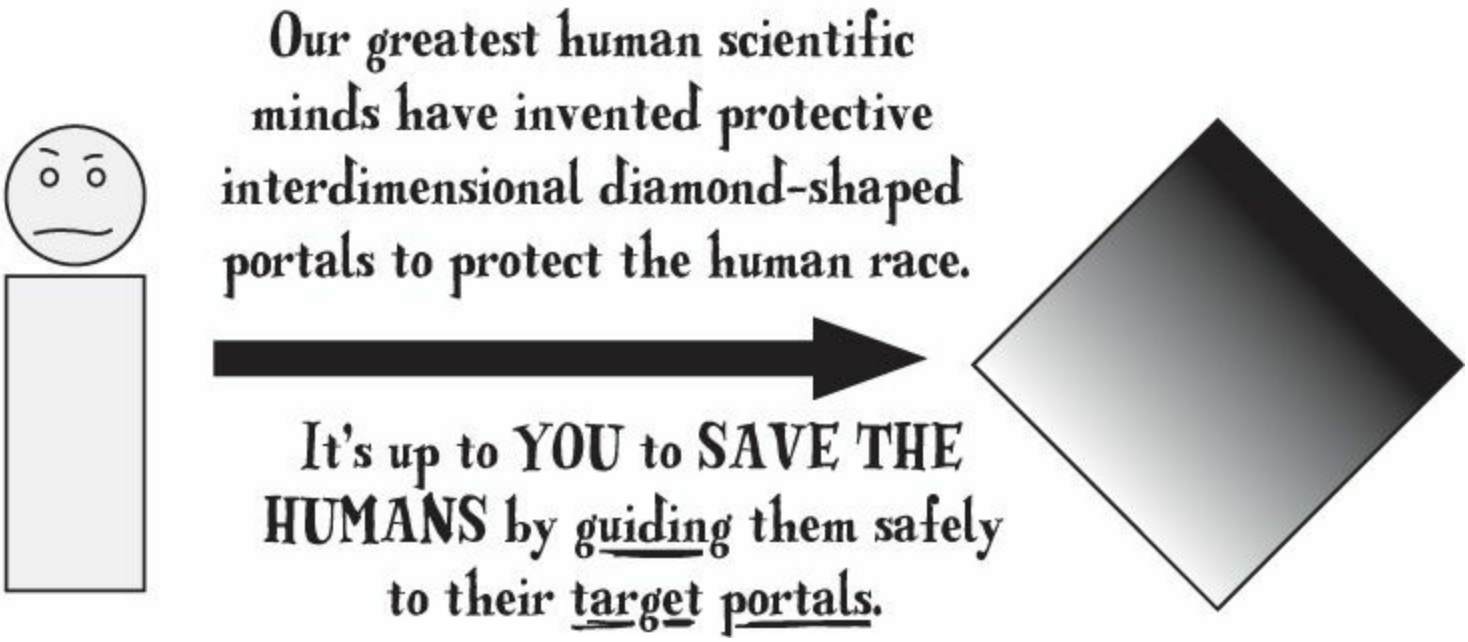
Well, there's a surprise: vicious aliens have launched a full-scale attack on planet Earth, abducting humans for their nefarious and unspeakable gastronomical experiments. Didn't see that coming!



Only <u>you</u> can help save the Earth

The last hopes of humanity rest on your shoulders! The people of planet Earth need you to **build an awesome C# app** to coordinate their escape from the alien menace. Are you up to the challenge?

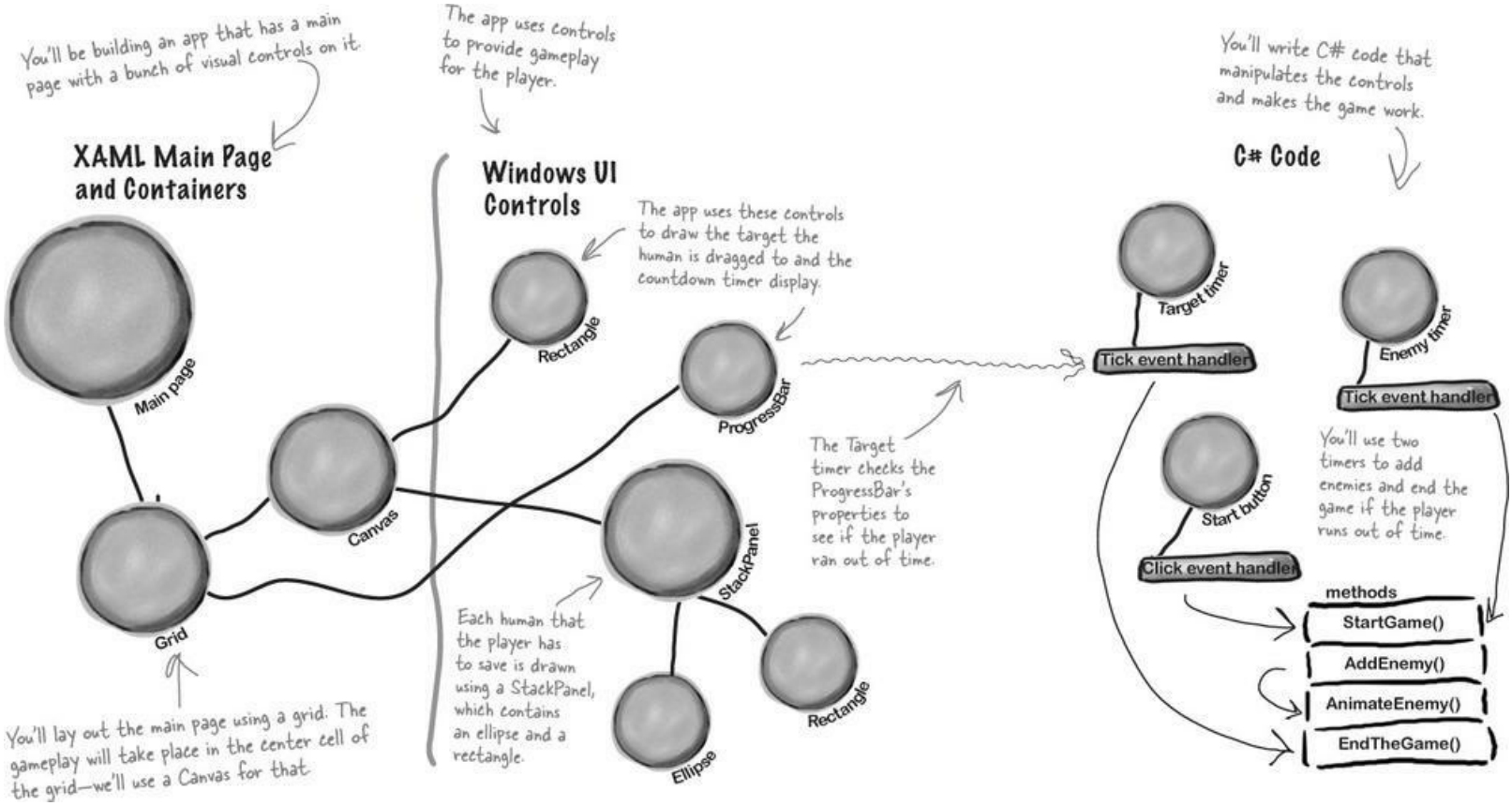




Here's what you're going to build

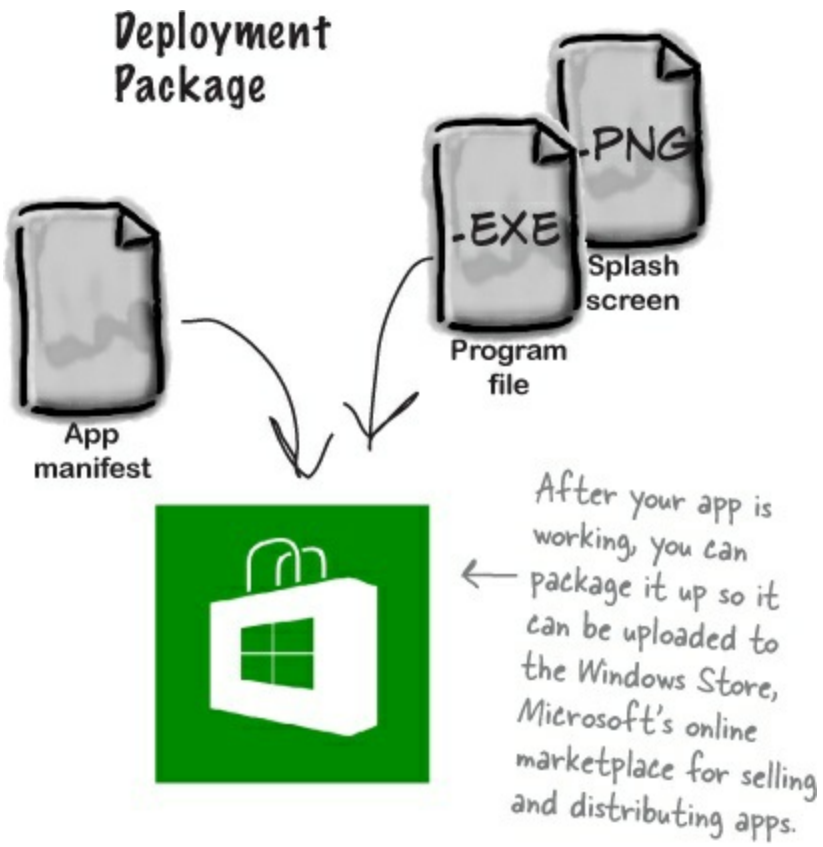
You're going to need an application with a graphical user interface, objects to make the game work, and an executable to run. It sounds like a lot of work, but you'll build all of this over the rest of the chapter, and by the end you'll have a pretty good handle on how to use the IDE to design a page and add C# code.

Here's the structure of the app we're going to create:



Save the Humans is a Windows Store app — you need Windows 8 to build and run it. Don't have Windows 8? The WPF Learner's Guide to Head First C# appendix at the end of this book shows you how to build this project as a desktop app.

You'll be building an app with two different kinds of code. First you'll design the user interface using XAML (Extensible Application Markup Language), a really flexible design language. Then you'll add C# code to make the game actually work. You'll learn a lot more about XAML throughout the second half of the book.



NOTE

It's not unusual for computers in an office to be running an operating system as old as Windows 2003, and may have an old version of Visual Studio. With WPF you can still do the projects in the book.

RELAX

No Windows 8? No problem.

The first two chapters and the last half of this book have many projects that are built with *Visual Studio 2013 for Windows*, but many readers aren't running Windows 8 yet. Luckily, almost all of the Windows Store apps in this book **can also be built as desktop apps** using Windows Presentation Foundation (WPF), which is compatible with earlier operating systems. Flip back to the last few pages of the Introduction, to the section called "What version of Windows are you using?" to learn more.

NOTE

The WPF Guide appendix contains complete replacement pages for the rest of this chapter, and then just five replacement pages for Chapter 2. After that, you won't need to use the WPF Learner's Guide appendix again until Chapter 10.

We worked really hard to build the WPF Learner's Guide appendix so that you keep page flipping to a minimum, while still letting you use an earlier version of Windows and even a

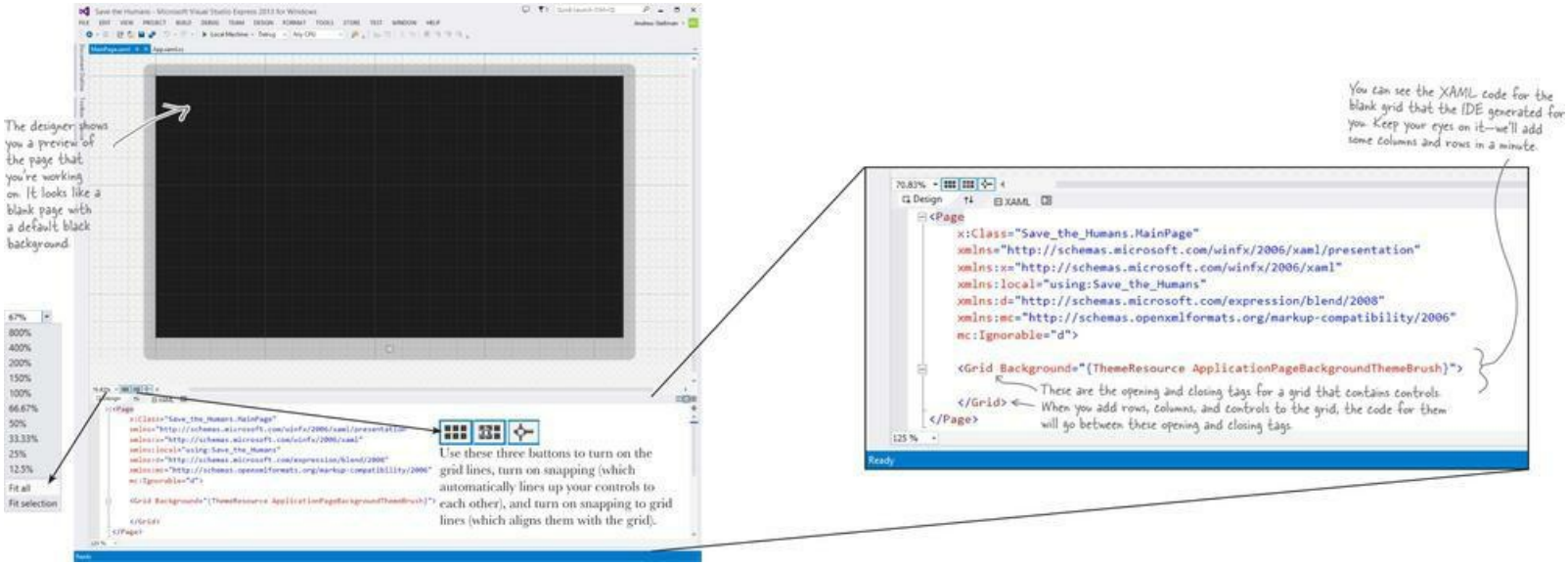
previous edition of Visual Studio to learn all of the same important C# concepts.

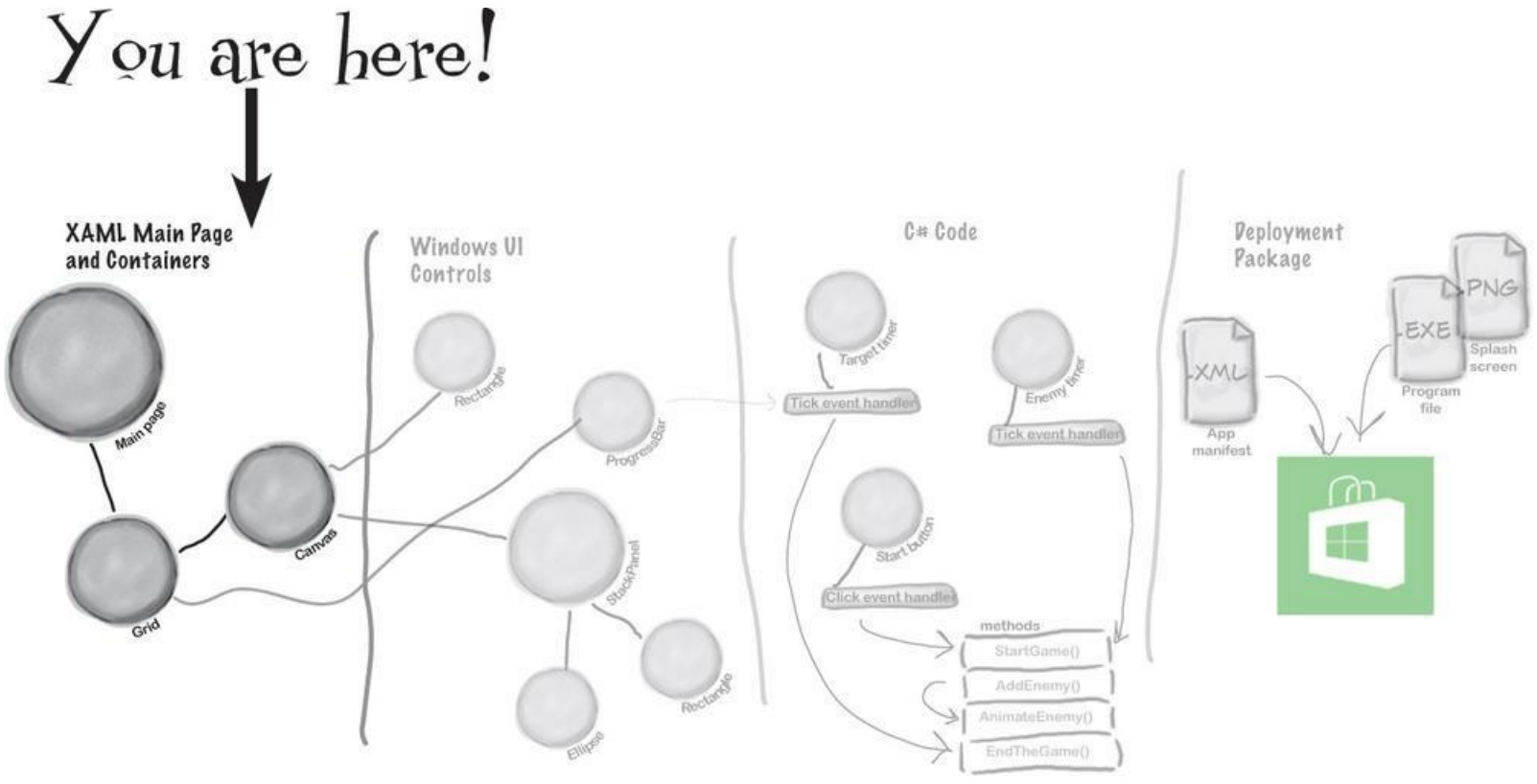
Start with a blank application

Every great app starts with a new project. Choose New Project from the File menu. Make sure you have Visual C#→Window Store selected and choose **Blank App (XAML)** as the project type. Type `Save the Humans` as the project name.

> **NOTE**
>
> If your code filenames don't end in ".cs" you may have accidentally created a JavaScript, Visual Basic, or Visual C++ program. You can fix this by closing the solution and starting over. If you want to keep the project name "Save the Humans," then you'll need to delete the previous project folder.

- ❶ Your starting point is the **Designer window**. Double-click on *MainPage.xaml* in the Solution Explorer to bring it up. Find the zoom drop-down in the lower-left corner of the designer and choose "Fit all" to zoom it out.



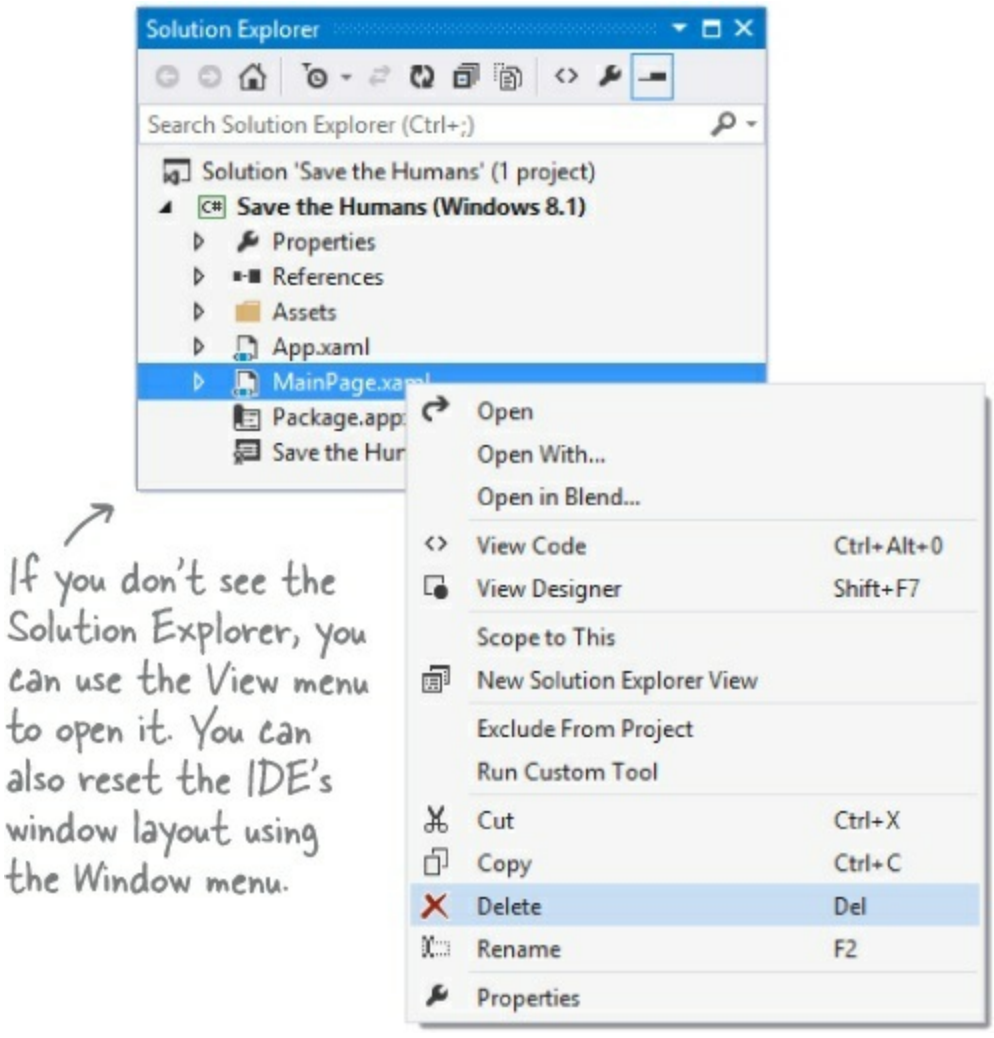


The bottom half of the Designer window shows you the XAML code. It turns out your "blank" page isn't blank at all — it contains a **XAML grid**. The grid works a lot like a table in an HTML page or Word document. We'll use it to lay out our pages in a way that lets them grow or shrink to different screen sizes and shapes.

> **LOOKING TO LEARN WPF? LOOK NO FURTHER!**
>
> *Most* of the Windows Store apps in this book **can be built with WPF** (Windows Presentation Foundation), which is compatible with Windows 7 and earlier operating systems and Visual Studio versions. Flip back to the last few pages of the Introduction, to the section called **"What version of Windows are you using?"** to learn more.

This part of the project has steps numbered ① to ⑤. Flip the page to keep going! ⟶

- ❷ Your page is going to need a title, right? And it'll need margins, too. You can do this all by hand with XAML, but there's an easier way to get your app to look like a normal Windows Store app.
  Go to the Solution Explorer window and find ▷ MainPage.xaml. Right-click on it and choose Delete to **delete the *MainPage.xaml* page**:

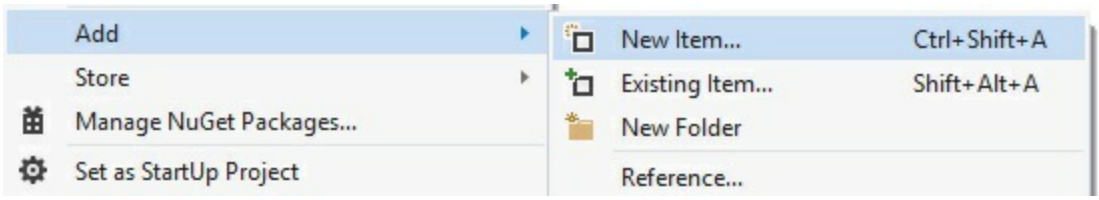


Over the next few pages you'll explore a lot of different features in the Visual Studio IDE, because we'll be using the IDE as a powerful tool for learning and teaching. You'll use the IDE throughout the book to explore C#. That's a really effective way to get it into your brain!

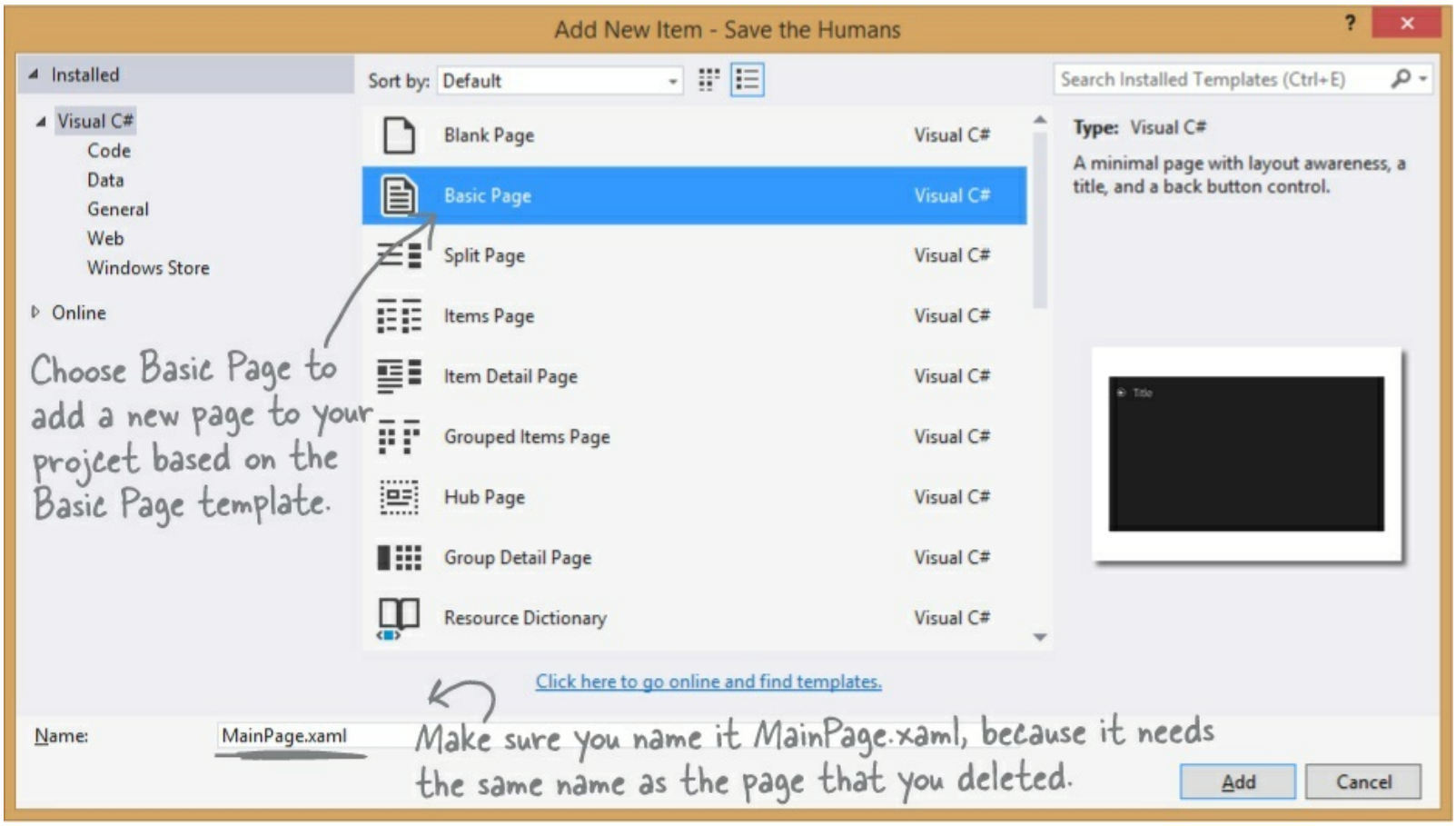
When you start a Windows Store app, you'll often replace the main page with one of the templates that Visual Studio provides.

- ❸ Now you'll need to replace the main page. Go back to the Solution Explorer and right-click on Save the Humans (it should be the second item in the Solution Explorer) to select the project. Then choose **Add→New Item...** from the menu:

> **NOTE**
>
> If you chose a different name when you created your project, you'll see that name instead of "Save the Humans" in the Solution Explorer.



The IDE will pop up the Add New Item window for your project. Choose **Basic Page** and give it the name **MainPage.xaml**. Then click the **Add** button to add the replacement page to your project.



The IDE will prompt you to add missing files — **choose *Yes* to add them**. Wait for the designer to finish loading. It might display either Invalid Markup or Build the Project to update Design view.. Choose **Rebuild Solution** from the Build menu to bring the IDE's Designer window up to date. Now you're ready to roll!

> **NOTE**
>
> When you replace MainPage.xaml with the new Basic Page item, the IDE needs to add additional files. Rebuilding the solution brings everything up to date so it can display the page in the designer.

Let's explore your newly added *MainPage.xaml file*. Scroll through the XAML pane in the designer window until you find this XAML code. This is the grid you'll use as the basis for your program:

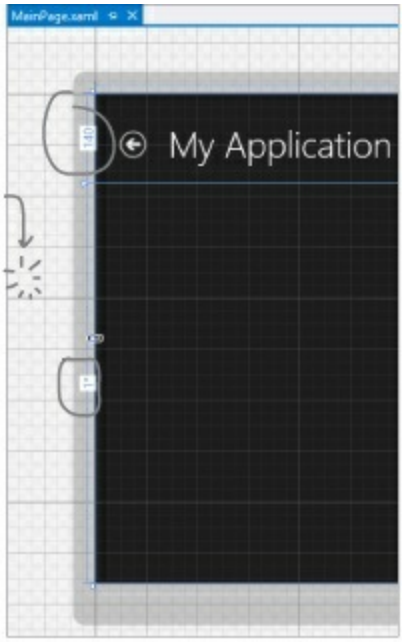
```
<!--
    This grid acts as a root panel for the page that defines two rows:
    * Row 0 contains the back button and page title
    * Row 1 contains the rest of the page layout
-->

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.ChildrenTransitions>
        <TransitionCollection>
            <EntranceThemeTransition/>
        </TransitionCollection>
    </Grid.ChildrenTransitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="140"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <!-- Back button and page title -->
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="120"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Button x:Name="backButton" Margin="39,59,39,0" Command="{Binding NavigationHelper.GoBackCom
```

You'll use the IDE to lay out your app by modifying this grid.

Your page should be displayed in the designer. If it isn't, double-click on MainPage.xaml in the Solution Explorer.

Notice how there's a whole separate grid, with its own starting <Grid> and ending </Grid> tags? That's the page header that displays the app name. This grid is also contained inside the root grid that you'll be adding controls to. The inner grid contains a Button and a TextBlock.

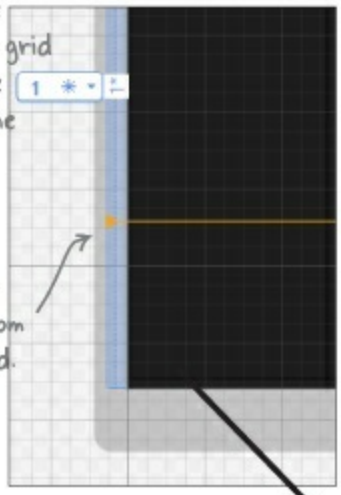
- ❹ Your app will be a grid with two rows and three columns (plus the header row that came with the blank page template), with one big cell in the middle that will contain the play area. Start defining rows by hovering over the border until a line and triangle appear:



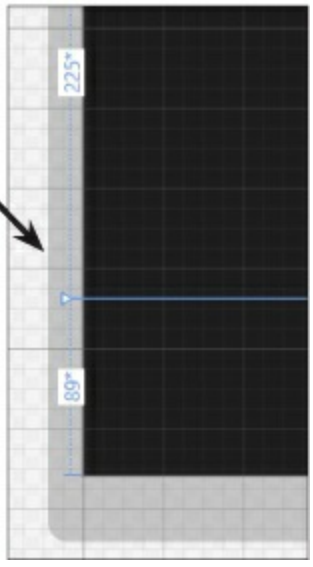
If you don't see the numbers 140 and 1* along the border of your page, click outside the page.

Hover over the border of the grid until an orange triangle and line appear...

...then click to create a bottom row in the grid.



After the row is added, the line will change to blue and you'll see the row height in the border. The height of the center row will change from 1* to a larger number followed by a star.



Windows Store apps need to look right on any screen, from tablets to laptops to giant monitors, in portrait or landscape.

Laying out the page using a grid's columns and rows allows your app to automatically adjust to the display.

THERE ARE NO DUMB QUESTIONS

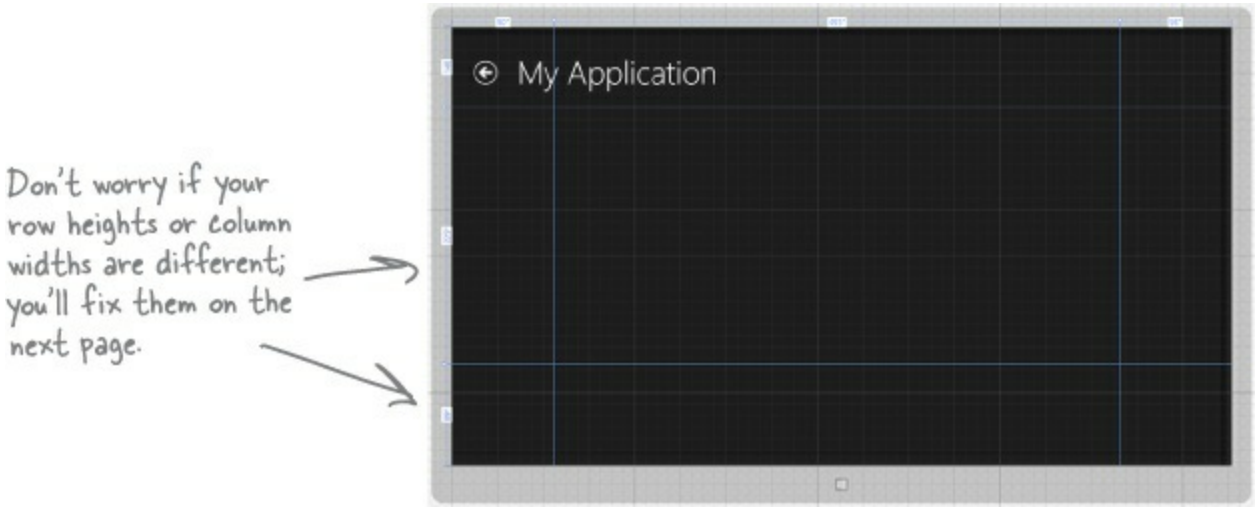
Q: **Q: But it looks like I already have many rows and and columns in the grid. What are those gray lines?**

A: **A:** The gray lines were just Visual Studio giving you a grid of guidelines to help you lay your controls out evenly on the page. You can turn them on and off with the button. None of the lines you see in the designer show up when you run the app outside of Visual Studio. But when you clicked and created a new row, you actually altered the XAML, which will change the way the app behaves when it's compiled and executed.

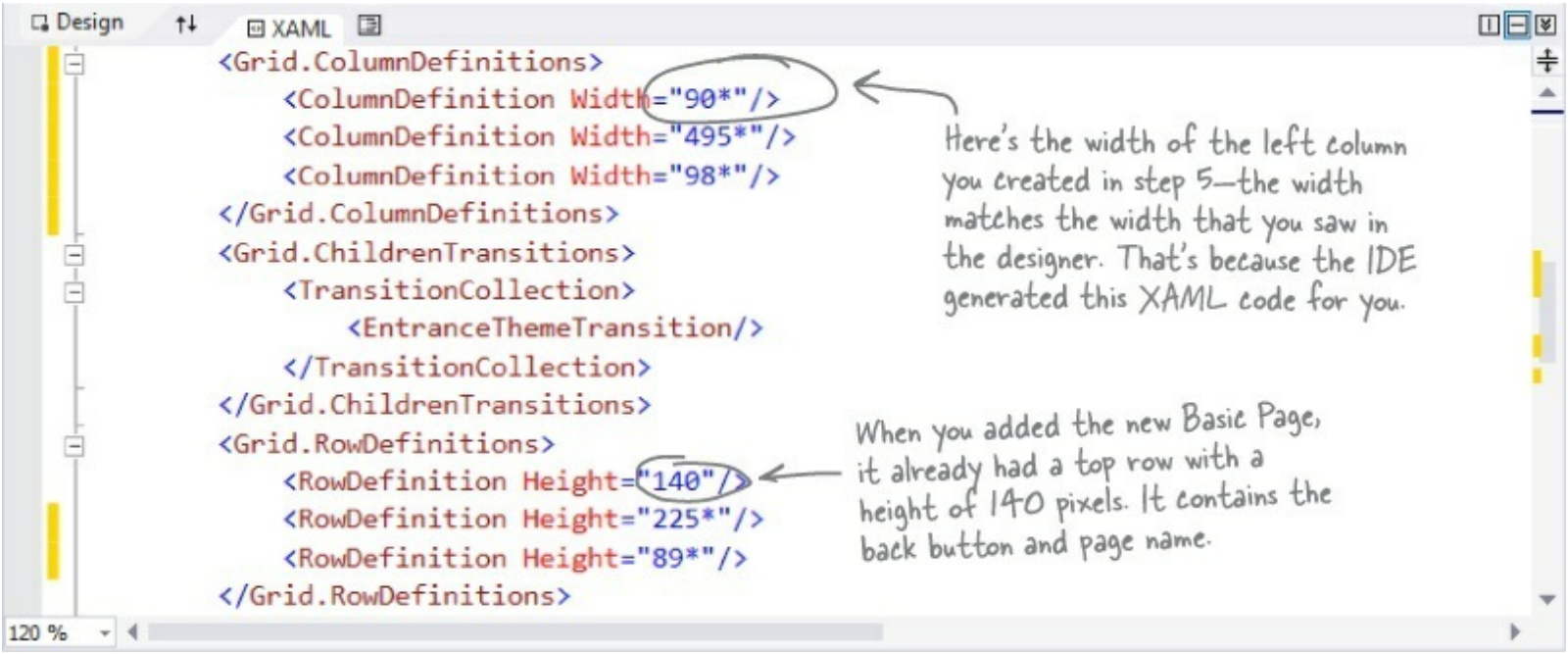
Q: **Q: Wait a minute. I wanted to learn about C#. Why am I spending all this time learning about XAML?**

A: **A:** Because Windows Store apps built in C# almost always start with a user interface that's designed in XAML. That's also why Visual Studio has such a good XAML editor — to give you the tools you need to build stunning user interfaces. Throughout this book, you'll learn how to build two other types of programs with C#, desktop applications and console applications, neither of which use XAML. Seeing all three of these will give you a deeper understanding of programming with C#.

- ❺ Do the same thing along the top border of the page — except this time create two columns, a small one on the lefthand side and another small one on the righthand side. Don't worry about the row heights or column widths — they'll vary depending on where you click. We'll fix them in a minute.

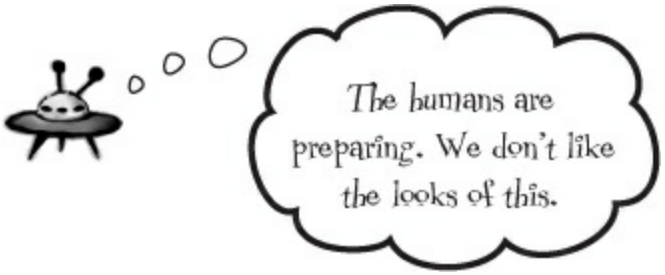


When you're done, look in the XAML window and go back to the same grid from the previous page. Now the column widths and row heights match the numbers on the top and side of your page.



Your grid rows and columns are now added!

XAML grids are **container controls**, which means they hold other controls. Grids consist of rows and columns that define cells, and each cell can hold other XAML controls that show buttons, text, and shapes. A grid is a great way to lay out a page, because you can set its rows and columns to resize themselves based on the size of the screen.

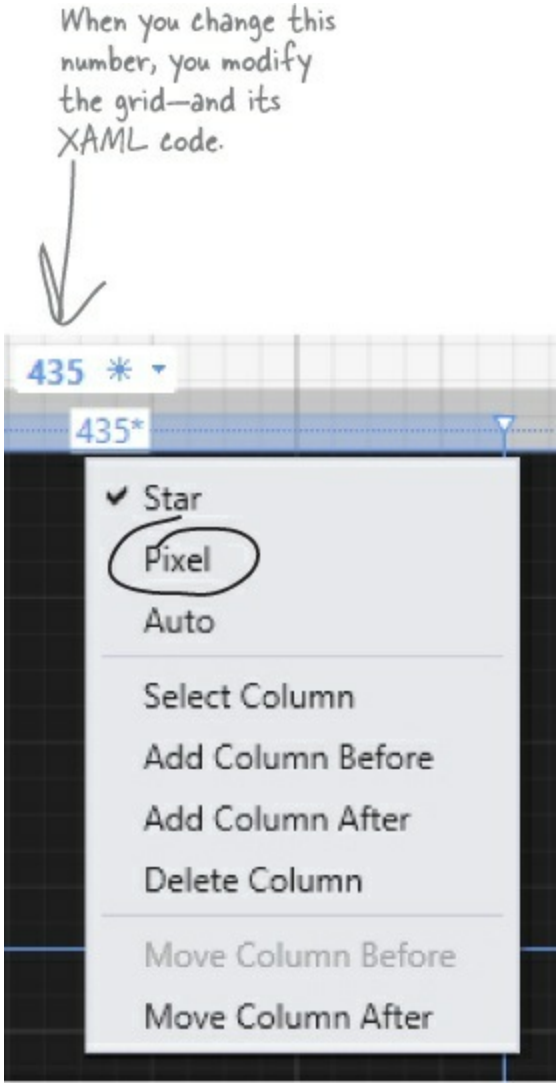


Set up the grid for your page

Your app needs to be able to work on a wide range of devices, and using a grid is a great way to do that. You can set the rows and columns of a grid to a specific pixel height. But you can also use the **Star** setting, which keeps them the same size proportionally — to each other and also to the page — no matter how big the display or what its orientation is.

- **❶ SET THE WIDTH OF THE LEFT COLUMN.**
  Hover over the number above the first column until a drop-down menu appears. Choose Pixel to change the star to a lock, then click on the number to change it to `160`. Your column's number should now look like this:

- **❷ REPEAT FOR THE RIGHT COLUMN AND THE BOTTOM ROW.**
  Make the right column and the bottom row 160 by choosing Pixel and typing `160` into the box.



Set your columns or rows to Pixel to give them a fixed width or height. The Star setting lets a row or column grow or shrink proportionally to the rest of the grid. Use this setting in the designer to alter the Width or Height property in the XAML. If you remove the Width or Height property, it's the same as setting the property to 1*.

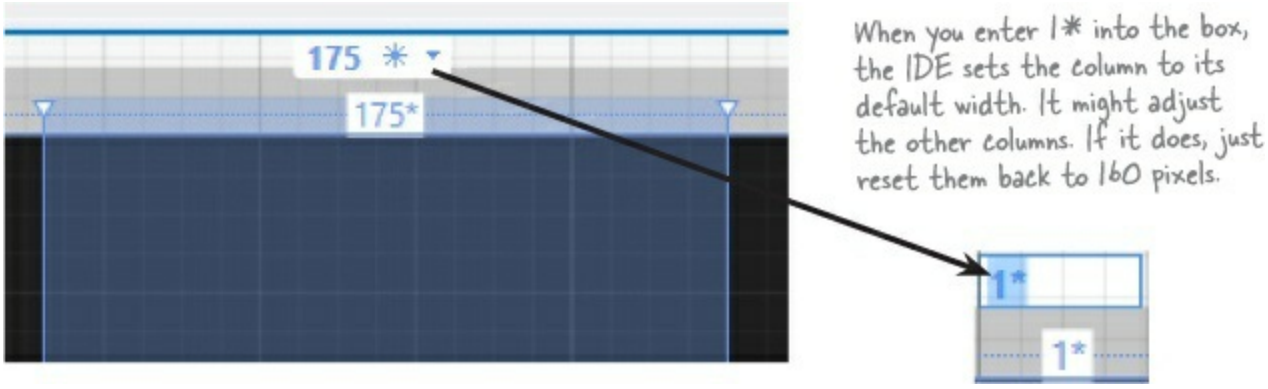
RELAX

It's OK if you're not a pro at app design...*yet*.

We'll talk a lot more about what goes into designing a good app later on. For now, we'll walk you through building this game. By the end of the book, you'll understand exactly what all of these things do!

- **❸ MAKE THE CENTER COLUMN AND CENTER ROW THE DEFAULT SIZE 1* (IF THEY AREN'T ALREADY).**
  Click on the number above the center column and enter 1. Don't use the drop-down (leave it Star) so it looks like the picture below. Then make sure to look back at the other columns to make sure the IDE didn't resize them. If it did, just change them back to `160`.

XAML and C# are case sensitive! Make sure your uppercase and lowercase letters match example code.



- **❹ LOOK AT YOUR XAML CODE!**
  Click on the grid to make sure it's selected, then look in the XAML window to see the code that you built.

```
<!--
    This grid acts as a root panel for the page that defines two rows:
    * Row 0 contains the back button and page title
    * Row 1 contains the rest of the page layout
-->
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="160"/>
        <ColumnDefinition/>
        <ColumnDefinition Width="160"/>
    </Grid.ColumnDefinitions>
    <Grid.ChildrenTransitions>
        <TransitionCollection>
            <EntranceThemeTransition/>
        </TransitionCollection>
    </Grid.ChildrenTransitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="140"/>
        <RowDefinition/>
        <RowDefinition Height="160"/>
    </Grid.RowDefinitions>
```

The <Grid .. > line at the top means everything that comes after it is part of the grid.

This is how a column is defined for a XAML grid. You added three columns and three rows, so there are three ColumnDefinition tags and three RowDefinition tags.

This top row with a height of 140 pixels is part of the Basic Page template you added.

You used the column and row drop-downs to set the Width and Height properties.

In a minute, you'll be adding controls to your grid, which will show up here, after the row and column definitions.

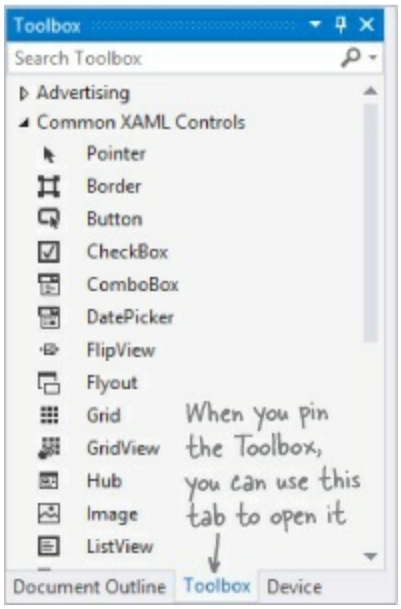
Add controls to your grid

Ever notice how apps are full of buttons, text, pictures, progress bars, sliders, drop-downs, and menus? Those are called **controls**, and it's time to add some of them to your app — *inside* the cells defined by your grid's rows and columns.

- ❶ Expand the Common XAML Controls section of the toolbox and drag a Button into the **bottom-left cell** of the grid.

Then look at the bottom of the Designer window and have a look at the **XAML tag** that the IDE generated for you. You'll see something like this — your margin numbers will be different depending on where in the cell you dragged it, and the properties might be in a different order.

If you don't see the toolbox in the IDE, you can open it using the View menu. Use the pushpin to keep it from collapsing.

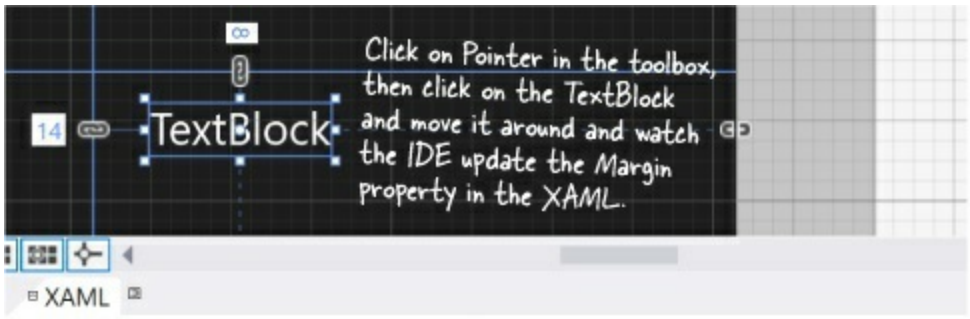


The XAML for the button starts here, with the opening tag.

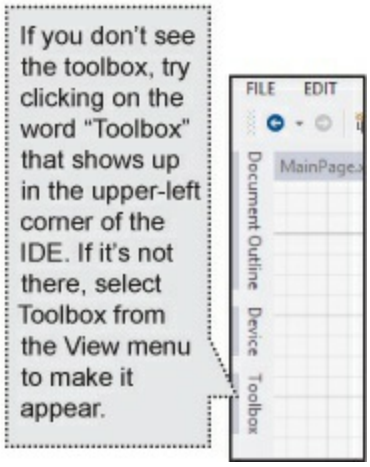
These are properties. Each property has a name, followed by an equals sign, followed by its value.

```
<Button Content="Button" HorizontalAlignment="Left"
        Margin="60,72,0,0" Grid.Row="2" VerticalAlignment="Top"/>
```

- ❷ Drag a TextBlock into the **lower-right cell** of the grid. Your XAML will look something like this. See if you can figure out how it determines which row and column the controls are placed in.



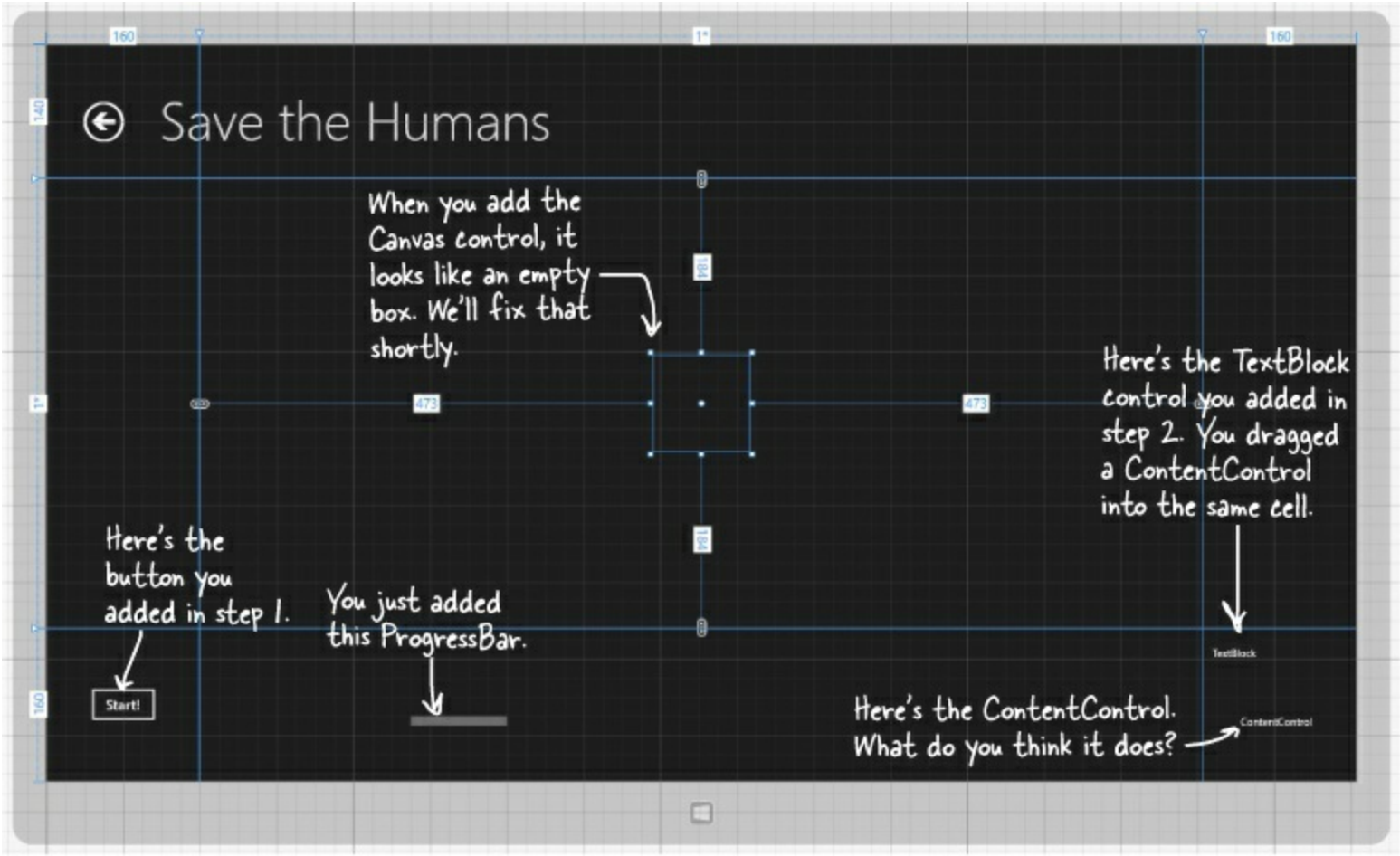

```
<TextBlock Grid.Column="2" HorizontalAlignment="Left"
           Margin="14,8,0,0" Grid.Row="2" TextWrapping="Wrap"
           Text="TextBlock" VerticalAlignment="Top"/>
```



NOTE

We added line breaks to make the XAML easier to read. You can add line breaks too. Give it a try!

- ❸ Next, expand the All XAML Controls section of the toolbox. Drag a ProgressBar into the bottom-center cell, a ContentControl into the bottom-right cell (make sure it's **below** the TextBlock you already put in that cell), and a Canvas into the center cell. Your page should now have controls on it (don't worry if they're placed differently than the picture below; we'll fix that in a minute):



- ❹ You've got the Canvas control currently selected, since you just added it. (If not, use the pointer to select it again.) Look in the XAML window:

```
<Canvas Grid.Column="1" Grid.Row="1" HorizontalAlignment="Left" Height="100"
```

It's showing you the XAML tag for the Canvas control. It starts with `<Canvas` and ends with `/>`, and between them it has properties like `Grid.Column="1"` (to put the Canvas in the center column) and `Grid.Row="1"` (to put it in the center row). Try clicking *in both the grid and the XAML window* to select different controls.

When you drag a control out of the toolbox and onto your page, the IDE automatically generates XAML to put it where you dragged it.

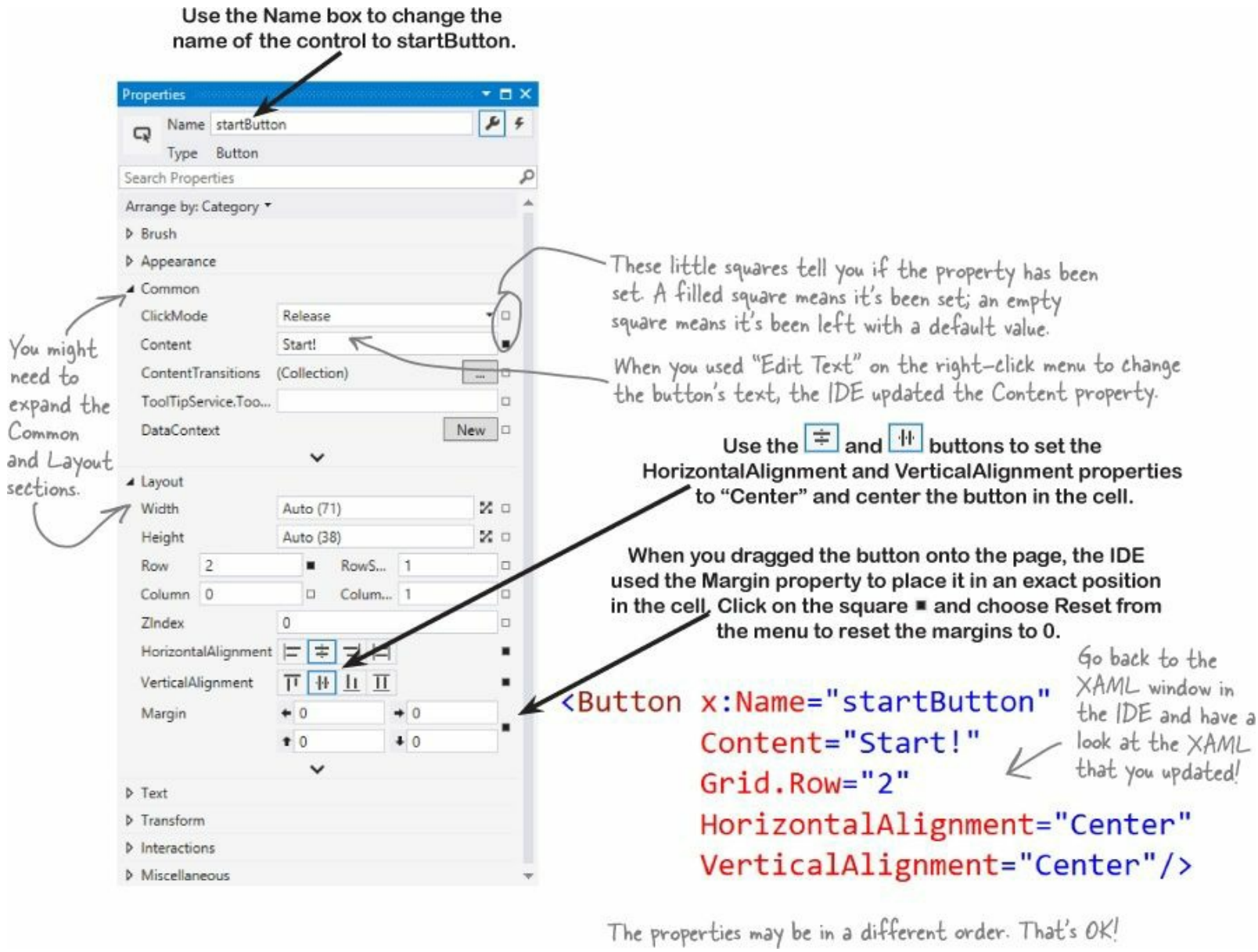
Use properties to change how the controls look

The Visual Studio IDE gives you fine control over your controls. The **Properties window** in the IDE lets you change the look and even the behavior of the controls on your page.

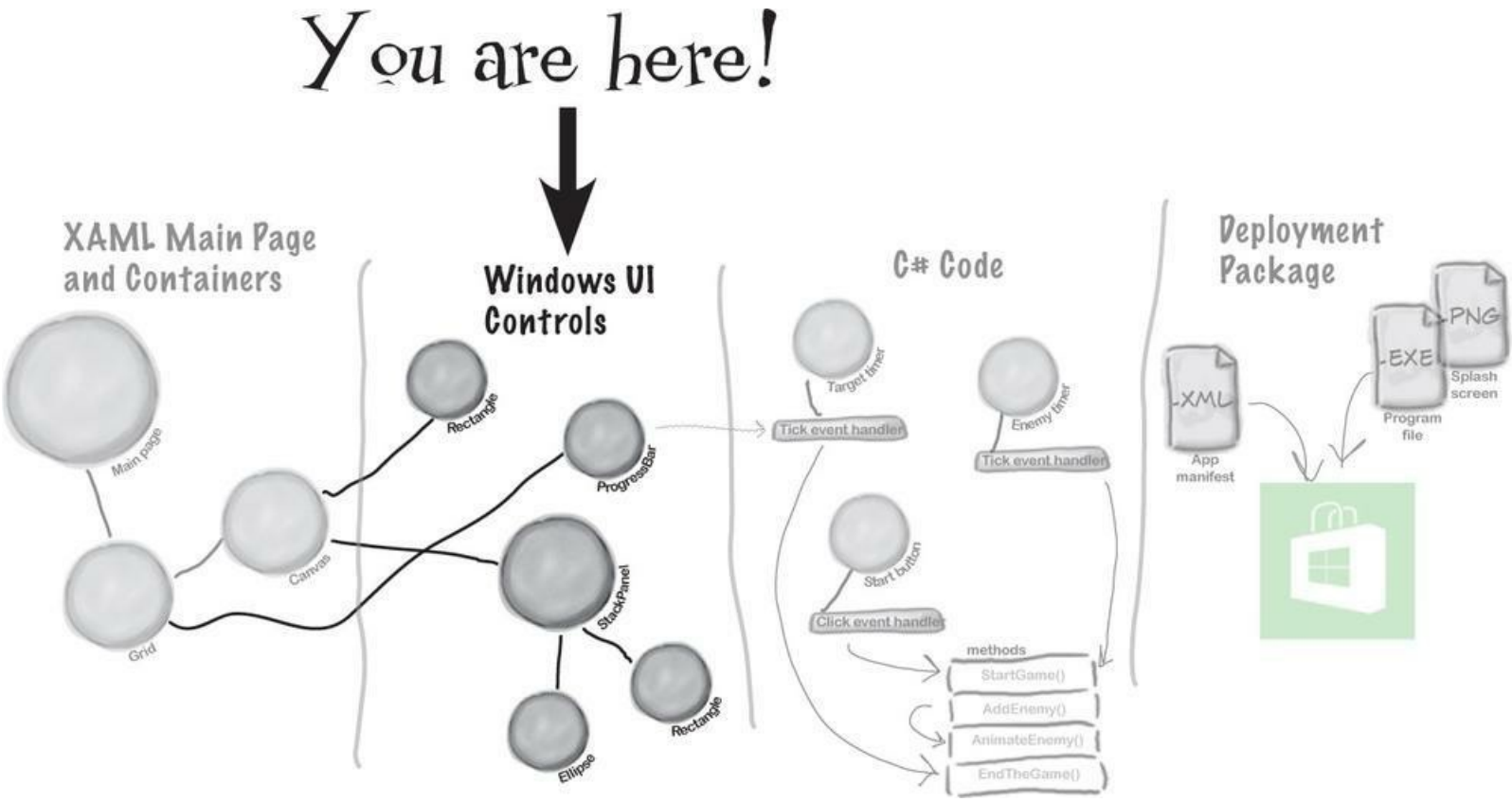
- **❶ Change the text of the button.**
  Right-click on the button control that you dragged onto the grid and choose **Edit Text** from the menu. Change the text to: `Start!` and see what you did to the button's XAML:

> **NOTE**
>
> When you're editing text, use the Escape key to finish. This works for other things in the IDE, too.

- **❷ Use the Properties window to modify the button.**
  Make sure the button is selected in the IDE, then look at the Properties window in the lower-right corner of the IDE. Use it to change the name of the control to `startButton` and center the control in the cell. Once you've got the button looking right, **right-click on it and choose View Source** to jump straight to the `<Button>` tag in the XAML window.



You can use Edit→Undo (or Ctrl-Z) to undo the last change. Do it several times to undo the last few changes. If you selected the wrong thing, you can choose Select None from the Edit menu to deselect. You can also hit Escape to deselect the control. If it's living inside a container like a StackPanel or Grid, hitting Escape will select the container, so you may need to hit it a few times.



- **❸ Change the page header text.**
  Right-click on the page header ("My Application") and choose View Source to jump to the XAML for the text block. Scroll in the XAML window until you find the `Text` property:

```
Text="{StaticResource AppName}"
```

  Wait a minute! That's not text that says "My Application" — what's going on here?
  The Blank Page template uses a **static resource** called `AppName` for the name that it displays at the top of the page. Scroll to the top of the XAML code until you find a `<Page.Resources>` section that has this XAML code in it:

```
<x:String x:Key="AppName">My Application</x:String>
```

  Replace "My Application" with the name of your application:

```
<x:String x:Key="AppName">Save the Humans</x:String>
```

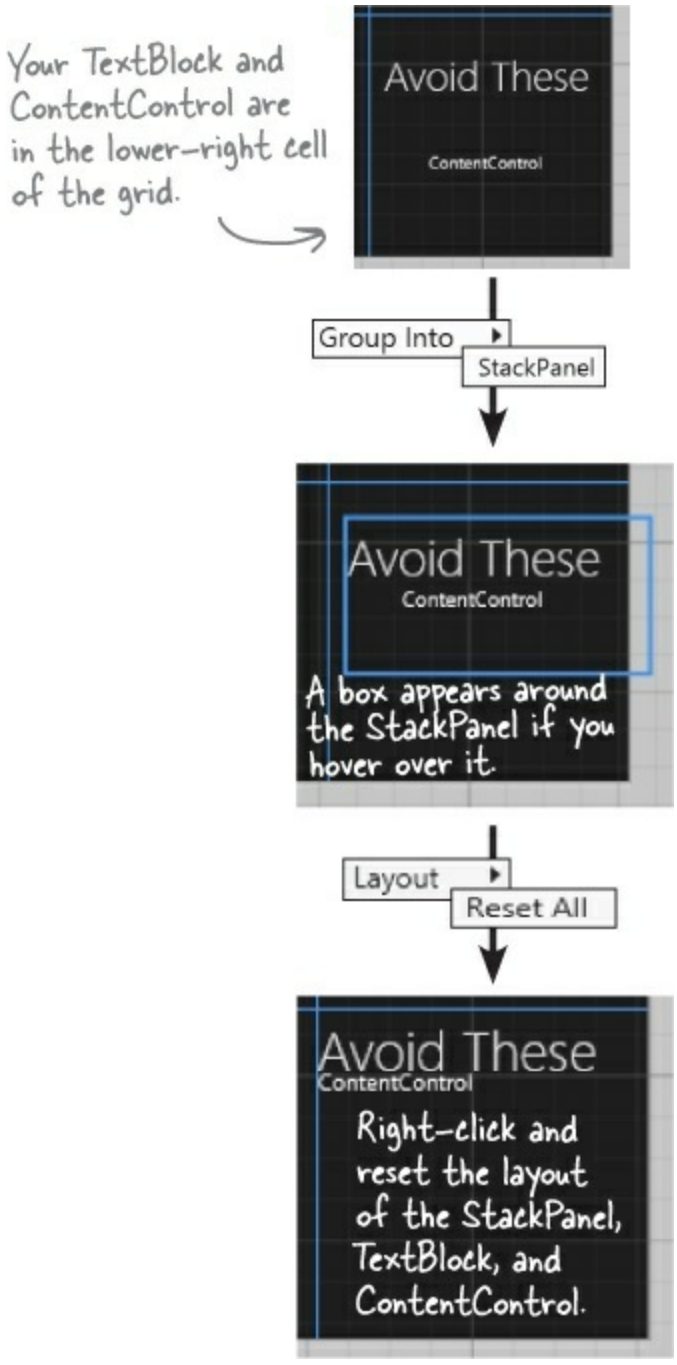
  Now you should see the correct text at the top of the page:

Don't worry about that back button. You'll learn all about how to use it in Chapter 14. You'll also learn about static resources.

- **❹ Update the TextBlock to change its text and its style.**
  Use the Edit Text right-mouse menu option to change the TextBlock so it says `Avoid These` (hit Escape to finish editing the text). Then right-click on it, choose the Edit Style menu item, and then choose the Apply Resource submenu and select **`SubheaderTextBlockStyle`** to make its text

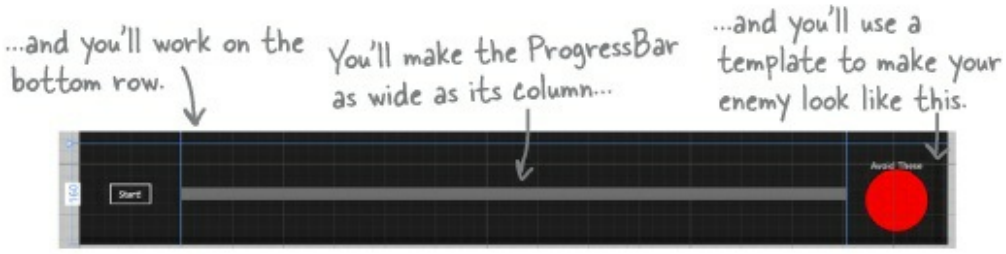
bigger.

- **❺ Use a StackPanel to group the TextBlock and ContentControl.**
  Make sure that the TextBlock is near the top of the cell, and the ContentControl is near the bottom. **Click and drag to select both the TextBlock and ContentControl, and then right-click**. Choose Group Into from the pop-up menu, then choose StackPanel. This adds a new control to your page: a **StackPanel control**. You can select the StackPanel by clicking between the two controls.
  The StackPanel is a lot like the Grid and Canvas: its job is to hold other controls (it's called a "container"), so it's not visible on the page. But since you dragged the TextBlock to the top of the cell and the ContentControl to the bottom, the IDE created the StackPanel so it fills up most of the cell. Click in the middle of the StackPanel to select it, then right-click and choose Layout and Reset All to quickly reset its properties, which will set its vertical and horizontal alignment to Stretch. Finally, right-click on the TextBox and ContentControl to reset their layouts as well. While you have the ContentControl selected, set its vertical and horizontal alignments to Center.



Controls make the game work

Controls aren't just for decorative touches like titles and captions. They're central to the way your game works. Let's add the controls that players will interact with when they play your game. Here's what you'll build next:

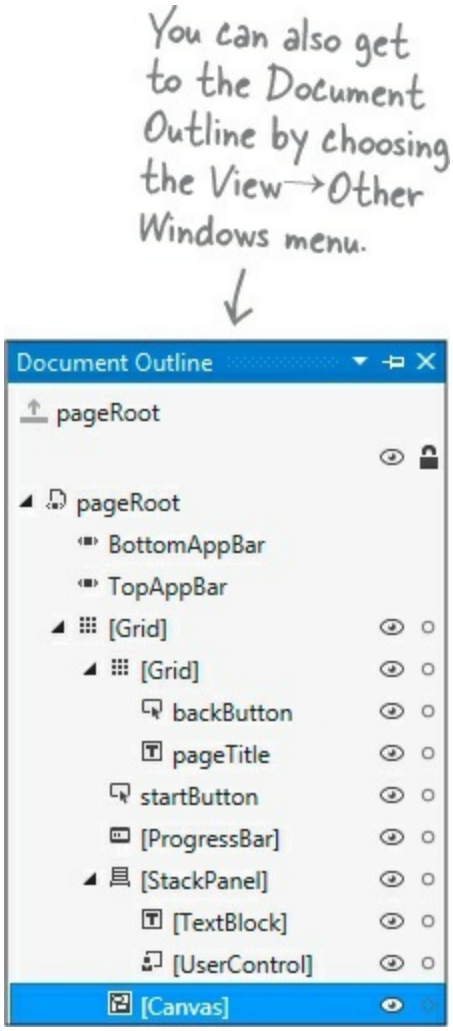


- **❶ Update the ProgressBar.**
  Right-click on the ProgressBar in the bottom-center cell of the grid, choose the **Layout** menu option, and then choose **Reset All** to reset all of the properties to their default values. Use the Height box in the Layout section of the Properties window to set the `Height` to `20`. The IDE stripped all of the layout-related properties from the XAML, and then added the new `Height`:

```
<ProgressBar Grid.Column="1" Grid.Row="2" Height="20"/>
```

- **❷ Turn the Canvas control into the gameplay area.**
  Remember that Canvas control that you dragged into the center square? It's hard to see it right now because a Canvas control is invisible when you first drag it out of the toolbox, but there's an easy way to find it. Click the very small button above the XAML window to bring up the **Document Outline**. Click on [Canvas] to select the Canvas control.
  Make sure the Canvas control is selected, then **use the Name box** in the Properties window to set the name to `playArea`.

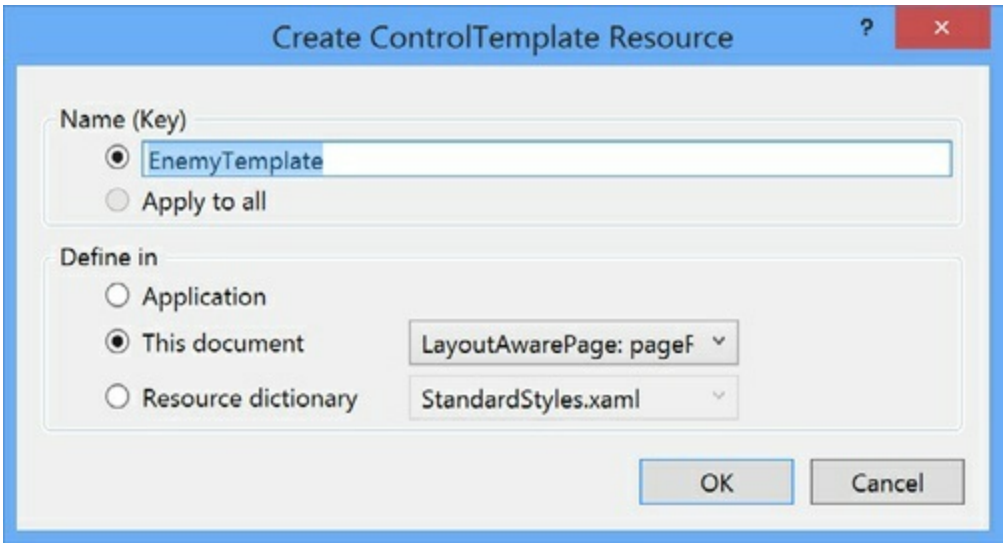
> **NOTE**
>
> Once you change the name, it'll show up as playArea instead of [Canvas] in the Document Outline window.



After you've named the Canvas control, you can close the Document Outline window. Then use the and buttons in the Properties window to set its vertical and horizontal alignments to Stretch, **reset the margins**, and click both buttons to set the `Width` and `Height` to Auto. Then set its Column to 0, and its ColumnSpan (next to Column) to 3.

Finally, open the **Brush** section of the Properties window and use the button to give it a **gradient**. Choose the starting and ending colors for the gradient by clicking each of the tabs at the bottom of the color editor and then clicking on a color.

- **❸ Create the enemy template.**
  Your game will have a lot of enemies bouncing around the screen, and you're going to want them to all look the same. Luckily, XAML gives us **templates**, which are an easy way to make a bunch of controls look alike.
  Next, right-click on the ContentControl in the Document Outline window. Choose **Edit Template**, then choose **Create Empty...** from the menu. Name it `EnemyTemplate`. The IDE will add the template to the XAML.



NOTE

You're "flying blind" for this next bit — the designer won't display anything for the template until you add a control and set its height and width so it shows up. Don't worry; you can always undo and try again if something goes wrong.

NOTE

You can also use the Document Outline window to select the grid if it gets deselected.

Your newly created template is currently selected in the IDE. Collapse the Document Outline window so it doesn't overlap the Toolbox. *Your template is **still invisible**, but you'll change that in the next step. If you accidentally click out of the control template, **you can always get back to it** by opening the Document Outline, right-clicking on the Content Control, and choosing Edit Template→Edit Current.*

- **❹ Edit the enemy template.**

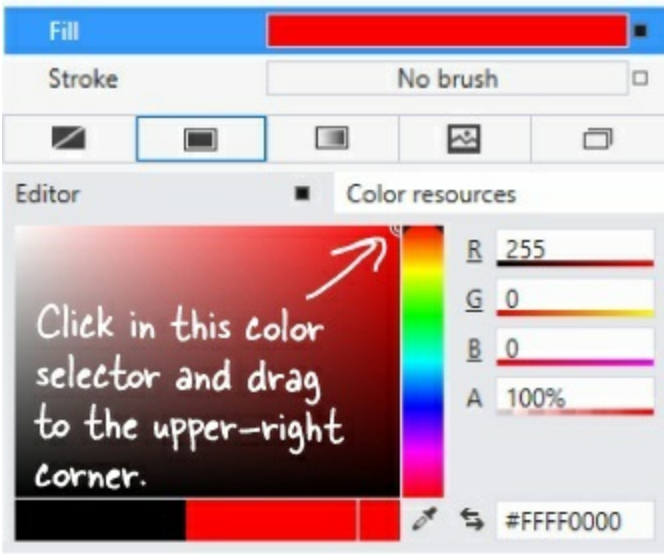
  Add a red circle to the template:

  Double-click on Ellipse in the toolbox to add an ellipse.

NOTE

— Make sure you don't click anywhere else in the designer until you see the ellipse. That will keep the template selected.

Set the ellipse's `Height` and `Width` properties to `100`, which will cause the ellipse to be displayed in the cell.
Reset the `HorizontalAlignment`, `VerticalAlignment`, and `Margin` properties by clicking on their squares and choosing Reset.
Go to the Brush section of the Properties window and click on ■ to select a solid-color brush.
Color your ellipse red by clicking in the color bar and dragging to the top, then clicking in the color sector and dragging to the upper-right corner.



The XAML for your ContentControl now looks like this:

```
<ContentControl Content="ContentControl"  HorizontalAlignment="Center"
                VerticalAlignment="Center" Template="{StaticResource EnemyTemplate}"/>
```

Scroll around your page's XAML window and see if you can find where the EnemyTemplate is defined. It should be right below the AppName resource.

- **❺ Use the Document Outline to modify the StackPanel and TextBlock controls.**
  Go back to the Document Outline (if you see EnemyTemplate (ContentControl Template) at the top of the Document Outline window, just click to get back to the Page outline). Select the StackPanel control, make sure its vertical and horizontal alignments are set to center, and clear the margins. Then do the same for the TextBlock.
  You're almost done laying out the page! Flip the page for the last steps... →
- **❻ Add the human to the Canvas.**
  You've got two options for adding the human. The first option is to follow the next three paragraphs. The second, quicker option is to just type the four lines of XAML into the IDE. It's your choice!
  Select the Canvas control, then open the **All XAML Controls** section of the toolbox and double-click on Ellipse to add an Ellipse control to the Canvas. Select the Canvas control again and double-click on Rectangle. The Rectangle will be added right on top of the Ellipse, so drag the Rectangle below it.
  Hold down the Shift key and click on the Ellipse so both controls are selected. Right-click on the Ellipse, choose **Group Into**, and then **StackPanel**. Select the Ellipse, use the solid brush property to change its color to white, and set its `Width` and `Height` properties to 10. Then select the Rectangle, make it white as well, and change its `Width` to 10 and its `Height` to 25.
  Use the Document Outline window to select the Stack Panel (make sure you see Type StackPanel at the top of the Properties window). **Click both buttons to set the `Width` and `Height` to Auto.** Then use the Name box at the top of the window to set its name to `human`. Here's the XAML you generated:

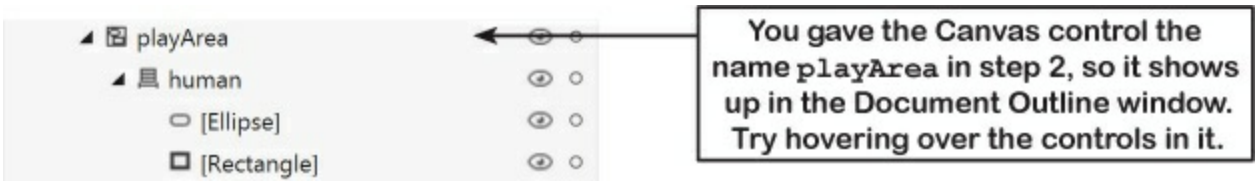
If you used the designer to create your human, make sure its source matches this XAML.

```
<StackPanel x:Name="human" Orientation="Vertical">
    <Ellipse Fill="White" Height="10" Width="10"/>
    <Rectangle Fill="White" Height="25" Width="10"/>
</StackPanel>
```

NOTE

If you choose to type this into the XAML window of the IDE, make sure you do it directly above the </Canvas> tag. That's how you indicate that the human is contained in the Canvas.

You might also see a Stroke property on the Ellipse and Rectangle set to `"Black"`. (If you don't see one, try adding it. What happens?)
Go back to the Document Outline window to see how your new controls appear:



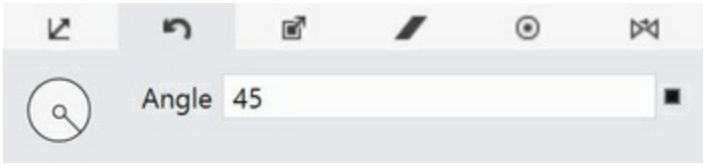
If `human` isn't indented underneath `playArea`, click and drag `human` onto it.

- **❼ Add the Game Over text.**
  When your player's game is over, the game will need to display a Game Over message. You'll do it by adding a TextBlock, setting its font, and giving it a name:

  Select the Canvas, then drag a TextBlock out of the toolbox and onto it.
  Use the Name box in the Properties window to change the TextBlock's name to `gameOverText`.
  Use the Text section of the Properties window to change the font to Arial Black, change the size to 100 px, and make it Bold and Italic.
  Click on the TextBlock and drag it to the middle of the Canvas.
  Edit the text so it says `Game Over`.

  > **When you drag a control around a Canvas, its Left and Top properties are changed to set its position. If you change the Left and Top properties, you move the control.**

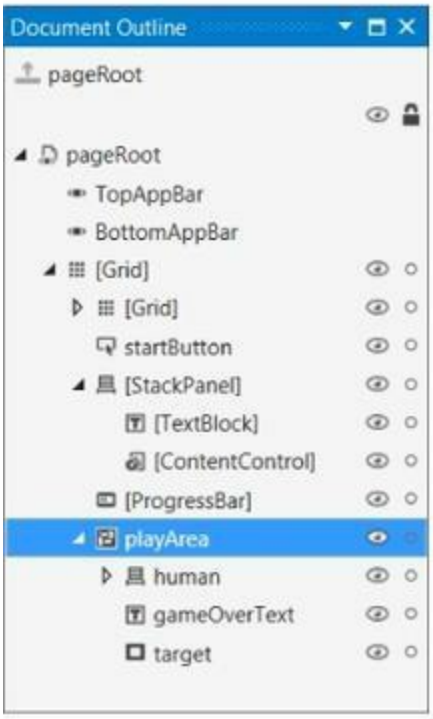
- **❽ Add the target portal that the player will drag the human onto.**
  There's one last control to add to the Canvas: the target portal that your player will drag the human into. (It doesn't matter where in the Canvas you drag it.)
  Select the Canvas control, then drag a Rectangle control onto it. Use the button in the Brushes section of the Properties window to give it a gradient. Set its `Height` and `Width` properties to **`50`**.
  Turn your rectangle into a diamond by rotating it 45 degrees. Open the Transform section of the Properties window to rotate the Rectangle 45 degrees by clicking on and setting the angle to **`45`**.



  Finally, use the Name box in the Properties window to give it the name `target`.

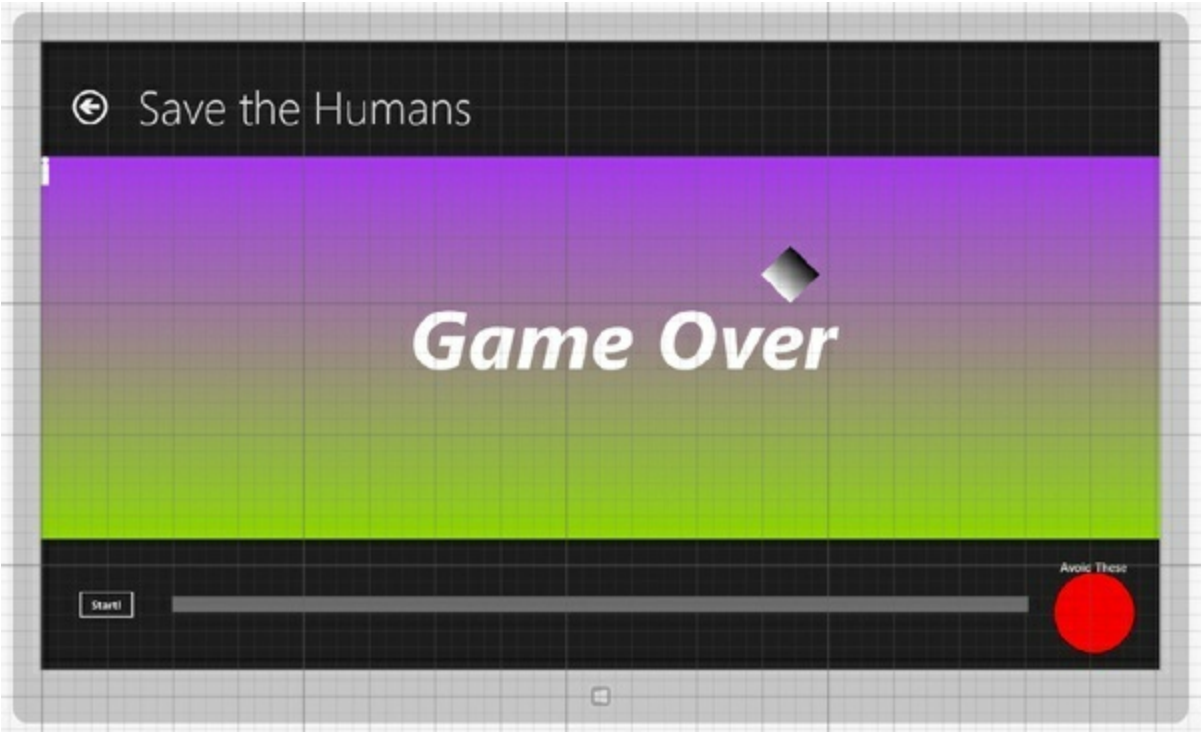
- **❾ Take a minute and double-check a few things.**
  Open the Document Outline window and make sure that the `human` StackPanel, `gameOverText` TextBlock, and `target` Rectangle are indented underneath the `playArea` Canvas control, which is indented under the second [Grid]. (If you see EnemyTemplate (ContentControl Template) at the top of the Document Outline window, just click to get back to the Page outline, which has pageRoot at the top.) Select the `playArea` Canvas control and make sure its Height and Width are set to Auto. These are all things that **could cause bugs** in your game that will be difficult to track down.

When you open the Document Outline for your page, it should look like this.



We collapsed human to make it obvious that it's indented underneath playArea, along with gameOverText and target. It's okay if the controls are in a different order (you can even drag them up an down!), as long as the indenting is correct—that's how you know which controls are inside other container controls.

Congratulations — you've finished building the main page for your app!



WHO DOES WHAT?

Now that you've built a user interface, you should have a sense of what some of the controls do, and you've used a lot of different properties to customize them. See if you can work out which property does what, and where in the Properties window in the IDE you find it.

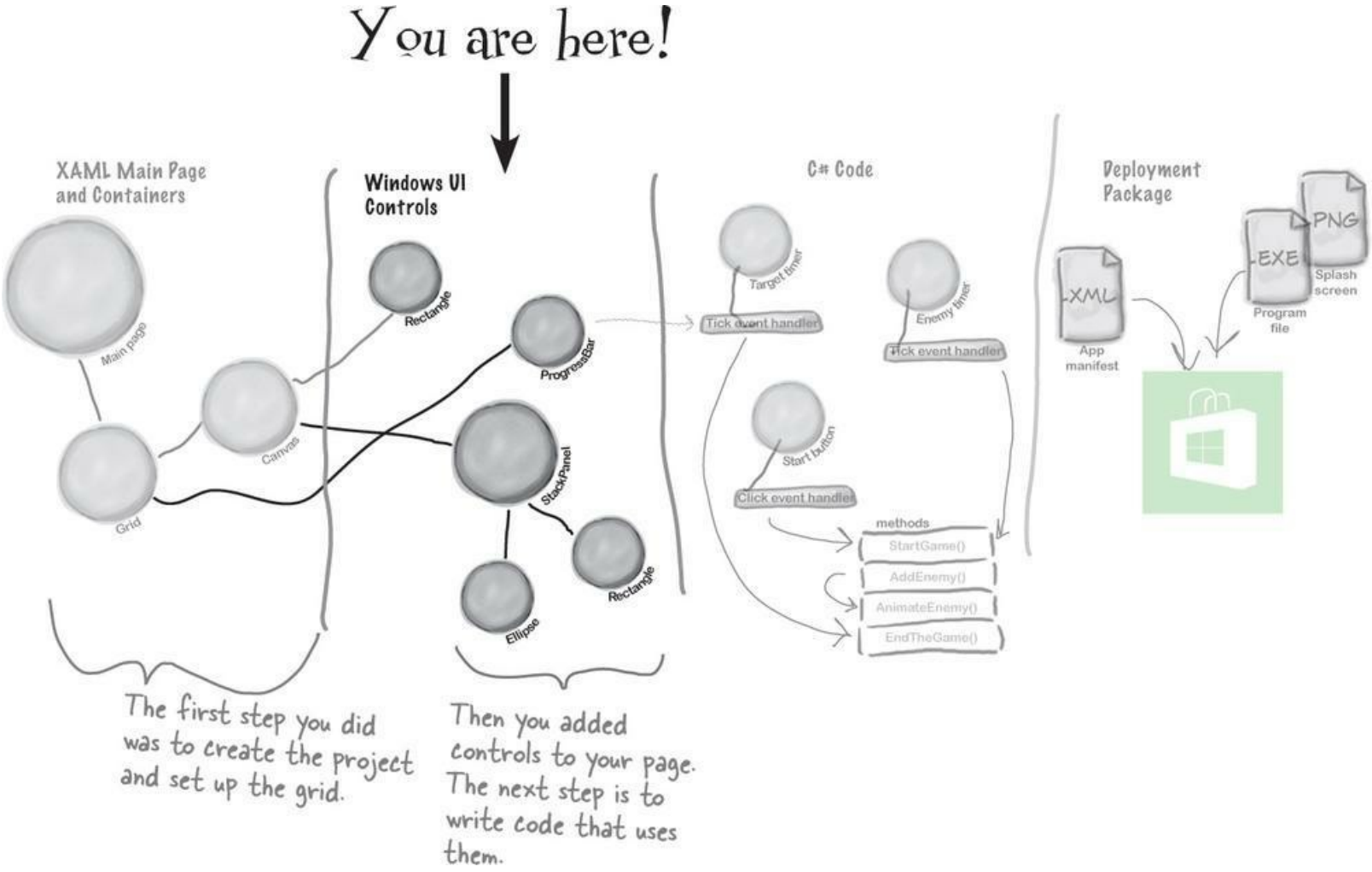
XAML property	Where to find it in the Properties window in the IDE	What it does
`Content`	At the top	Determines how tall the control should be
`Height`	▷ Brush	Sets the angle that the control is turned
`Rotation`	▷ Appearance	You use this in your C# code to manipulate a specific control
`Fill`	▷ Common	The color of the control
`x:Name`	▷ Layout	Use this when you want to change text displayed inside your control
	▷ Transform	

Solution in Who Does What? Solution →

Here's a hint: you can use the Search box in the Properties window to find properties — but some of these properties aren't on every type of control.

You've set the stage for the game

Your page is now all set for coding. You set up the grid that will serve as the basis of your page, and you added controls that will make up the elements of the game.



Visual Studio gave you useful tools for laying out your page, but all it really did was help you create XAML code. You're the one in charge!

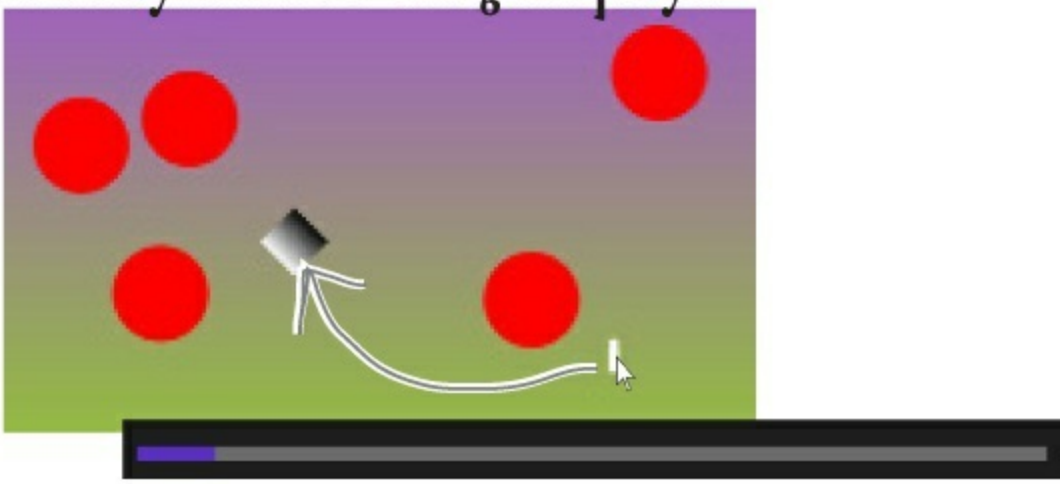
What you'll do next

Now comes the fun part: adding the code that makes your game work. You'll do it in three stages: first you'll animate your enemies, then you'll let your player interact with the game, and finally you'll add polish to make the game look better.

First you'll animate the enemies...

The first thing you'll do is add C# code that causes enemies to shoot out across the play area every time you click the Start button.

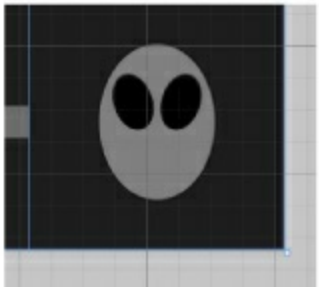
...then you'll add the gameplay...



To make the game work, you'll need the progress bar to count down, the human to move, and the game to end when the enemy gets him or time runs out.

You used a template to make the enemies look like red circles. Now you'll update the template to make them look like evil alien heads.

...and finally, you'll make it look good.



A lot of programmers build their code in small increments, making sure one piece works before moving on to the next one. That's how you'll build the rest of this program. You'll start by creating a method called `AddEnemy()` that adds an animated enemy to the Canvas control. First you'll hook it up to the Start button so you can fill your page up with bouncing enemies. That will lay the groundwork to build out the rest of the game.

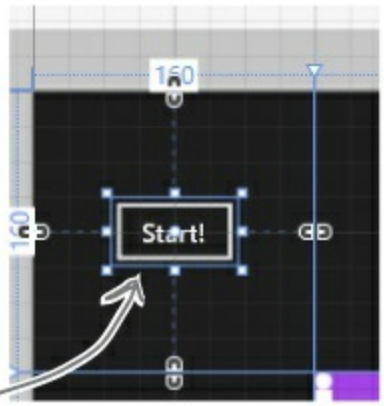
```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    AddEnemy();
}
```

The red squiggly line is the IDE telling you there's a problem, and the blue box is the IDE telling you that it might have a solution.

Add a method that does something

It's time to start writing some C# code, and the first thing you'll do is add a **method** — and the IDE can give you a great starting point by generating code.

When you're editing a page in the IDE, double-clicking on any of the controls on the page causes the IDE to automatically add code to your project. Make sure you've got the page designer showing in the IDE, and then double-click on the Start button. The IDE will add code to your project that gets run any time a user clicks on the button. You should see some code pop up that looks like this:



When you double-clicked on the Button control, the IDE created this method. It will run when a user clicks the "Start!" button in the running application.

```
private void startButton_Click(object sender, RoutedEventArgs e)
{

}
```

```
Click="startButton_Click"
```

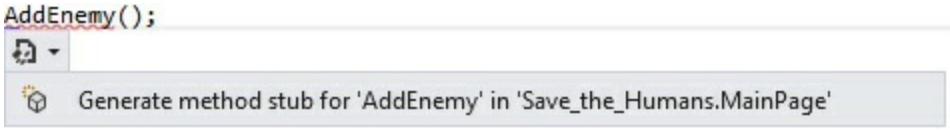
The IDE also added this to the XAML. See if you can find it. You'll learn more about what this is in Chapter 2.

Use the IDE to create your own method

Click between the { } brackets and type this, including the parentheses and semicolon:

Notice the red squiggly line underneath the text you just typed? That's the IDE telling you that something's wrong. If you click on the squiggly line, a blue box appears, which is the IDE's way of telling you that it might be able to help you fix the error.

Hover over the blue box and click the icon that pops up. You'll see a box asking you to generate a method stub. What do you think will happen if you click it? Go ahead and click it to find out!



THERE ARE NO DUMB QUESTIONS

Q: **Q: What's a method?**

A: **A:** A **method** is just a *named block of code*. We'll talk a lot more about methods in Chapter 2.

Q: **Q: And the IDE generated it for me?**

A: **A:** Yes...for now. A method is one of the basic building blocks of programs — you'll write a lot of them, and you'll get used to writing them by hand.

WATCH IT!

C# code must be added exactly as you see it here.

It's really easy to throw off your code. When you're adding C# code to your program, the capitalization has to be exactly right, and make sure you get all of the parentheses, commas, and semicolons. If you miss one, your program won't work!

Fill in the code for your method

It's time to make your program ***do something***, and you've got a good starting point. The IDE generated a **method stub** for you: the starting point for a method that you can fill in with code.

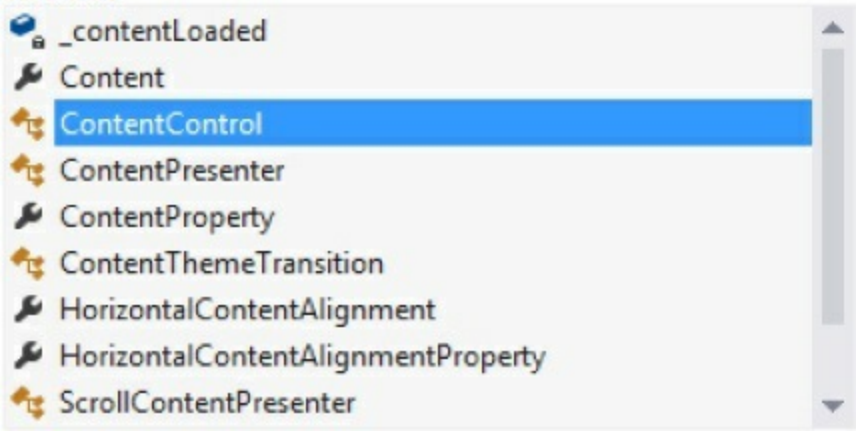
- ❶ Delete the contents of the method stub that the IDE generated for you.

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

Select this and delete it. You'll learn about exceptions in Chapter 12.

- ❷ Start adding code. Type the word Content into the method body. The IDE will pop up a window called an **IntelliSense Window** with suggestions. Choose `ContentControl` from the list.

```
private void AddEnemy()
{
    Content
}
```



- ❸ Finish adding the first line of code. You'll get another IntelliSense window after you type `new`.

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
}
```

This line creates a new ContentControl object. You'll learn about objects and the new keyword in Chapter 3, and reference variables like enemy in Chapter 4.

Make sure each XAML control has the right name, and all properties (like Width and Height) are correct! If not, your program might crash.

- ❹ Before you fill in the `AddEnemy()` method, you'll need to add a line of code near the top of the file. Find the line that starts with **`public sealed partial class MainPage`** and add this line after the bracket (`{`) and before the first line of code (`private NavigationHelper navigationHelper;`):

```
/// <summary>
/// A basic page that provides characteristics common to most applications.
/// </summary>
public sealed partial class MainPage : Page
{
    Random random = new Random();

    private NavigationHelper navigationHelper;
    private ObservableDictionary defaultViewModel = new ObservableDictionary();
```

This is called a field. You'll learn more about how it works in Chapter 4.

- ❺ Finish adding the method. You'll see some squiggly red underlines. The ones under `AnimateEnemy()` will go away when you generate its method stub.

Do you see a squiggly underline under playArea? Go back to the XAML editor and sure you set the name of the Canvas control to playArea.

This line adds your new enemy control to a collection called Children. You'll learn about collections in Chapter 8.

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
    enemy.Template = Resources["EnemyTemplate"] as ControlTemplate;
    AnimateEnemy(enemy, 0, playArea.ActualWidth - 100, "(Canvas.Left)");
    AnimateEnemy(enemy, random.Next((int)playArea.ActualHeight - 100),
        random.Next((int)playArea.ActualHeight - 100), "(Canvas.Top)");
    playArea.Children.Add(enemy);
}
```

If you need to switch between the XAML and C# code, use the tabs at the top of the window.

- ❻ Use the blue box and the button to generate a method stub for `AnimateEnemy()`, just like you did for `AddEnemy()`. This time it added four **parameters** called `enemy`, `p1`, `p2`, and `p3`. Edit the top line of the method to change the last three parameters. Change the parameter `p1` to **`from`**, the parameter `p2` to **`to`**, and the parameter `p3` to **`propertyToAnimate`**. Then change any `int` types to **`double`**.

You'll learn about methods and parameters in Chapter 2.

```
private void AnimateEnemy(ContentControl enemy, int p1, double p2, string p3)
{
    throw new NotImplementedException();
}
```

```
private void AnimateEnemy(ContentControl enemy, double from, double to, string propertyToAnimate)
```

The IDE may generate the method stub with "int" types. Change them to "double". You'll learn about types in Chapter 4.

Flip the page to see your program run!

Finish the method and run your program

Your program is almost ready to run! All you need to do is finish your `AnimateEnemy()` method. Don't panic if things don't quite work yet. You may have missed a comma or some parentheses — when you're programming, you need to be really careful about those things!

RELAX

Still seeing red? The IDE helps you track down problems.

If you still have some of those red squiggly lines, don't worry! You probably just need to track down a typo or two. If you're still seeing squiggly red underlines, it just means you didn't type in some of the code correctly. We've tested this chapter with a lot of different people, and we didn't leave anything out. All of the code you need to get your program working is in these pages.

- **❶ Add a using statement to the top of the file.**
  Scroll all the way to the top of the file. The IDE generated several lines that start with `using`. Add one more to the bottom of the list:

Statements like these let you use code from .NET libraries that come with C#. You'll learn more about them in Chapter 2.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

using Windows.UI.Xaml.Media.Animation;
```

You'll need this line to make the next bit of code work. You can use the IntelliSense window to get it right—and don't forget the semicolon at the end.

This using statement lets you use animation code from the .NET Framework in your program to move the enemies on your screen.

- **❷ Add code that creates an enemy bouncing animation.**
  You generated the method stub for the `AnimateEnemy()` method on the previous page. Now you'll add its code. It makes an enemy start bouncing across the screen.

You'll learn about object initializers like this in Chapter 4.

```
private void AnimateEnemy(ContentControl enemy, double from, double to, string propertyToAnimate)
{
    Storyboard storyboard = new Storyboard() { AutoReverse = true, RepeatBehavior = RepeatBehavior.Forever };
    DoubleAnimation animation = new DoubleAnimation()
    {
        From = from,
        To = to,
        Duration = new Duration(TimeSpan.FromSeconds(random.Next(4, 6)))
    };
    Storyboard.SetTarget(animation, enemy);
    Storyboard.SetTargetProperty(animation, propertyToAnimate);
    storyboard.Children.Add(animation);
    storyboard.Begin();
}
```

And you'll learn about animation in Chapter 16.

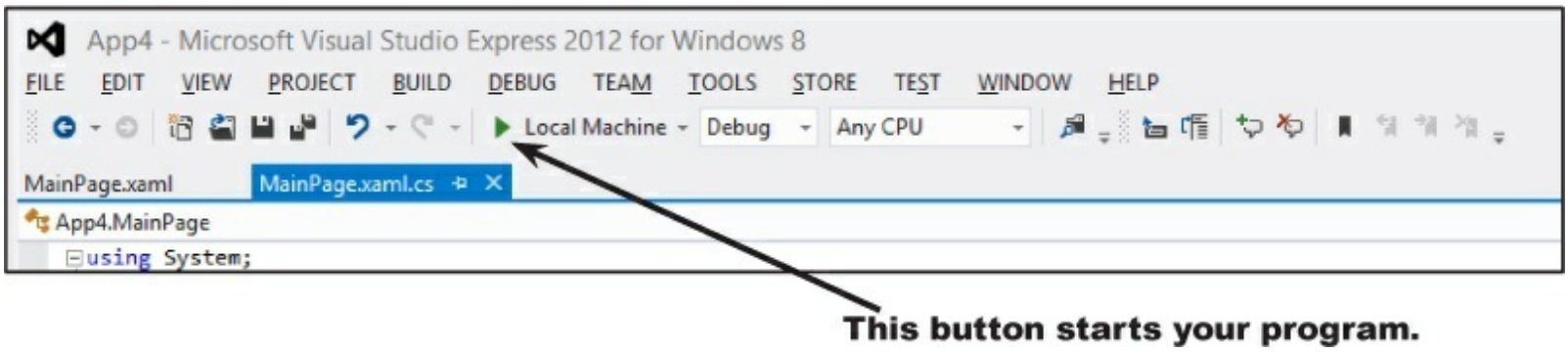
This code makes the enemy you created move across playArea. If you change 4 and 6, you can make the enemies move slower or faster.

- **❸ Look over your code.**
  You shouldn't see any errors, and your Error List window should be empty. If not, double-click on the error in the Error List. The IDE will jump your cursor to the right place to help you track down the problem.

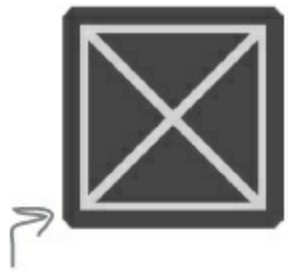
> **NOTE**
>
> If you can't see the Error List window, choose Error List from the View menu to show it. You'll learn more about using the error window and debugging your code in Chapter 2.

> **Here's a hint: if you move too many windows around your IDE, you can always reset by choosing Reset Window Layout from the Window menu.**

- **❹ Start your program.**
  Find the ▶ button at the top of the IDE. This starts your program running.

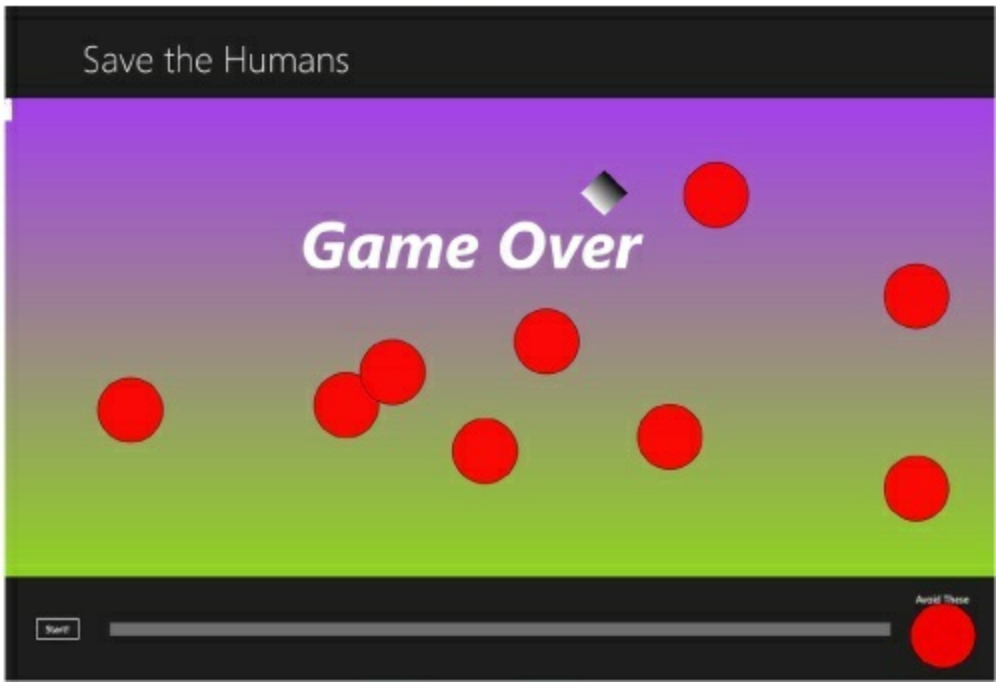


- **❺ Now your program is running!**
  First, a big X will be displayed for a few seconds, and then your main page will be displayed. Click the "Start!" button a few times. Each time you click it, a circle is launched across your canvas.



This big X is the splash screen. You'll make your own splash screen at the end of the chapter.

You built something cool! And it didn't take long, just like we promised. But there's more to do to get it right.



If the enemies aren't bouncing, or if they leave the play area, double-check the code. You may be missing parentheses or keywords.

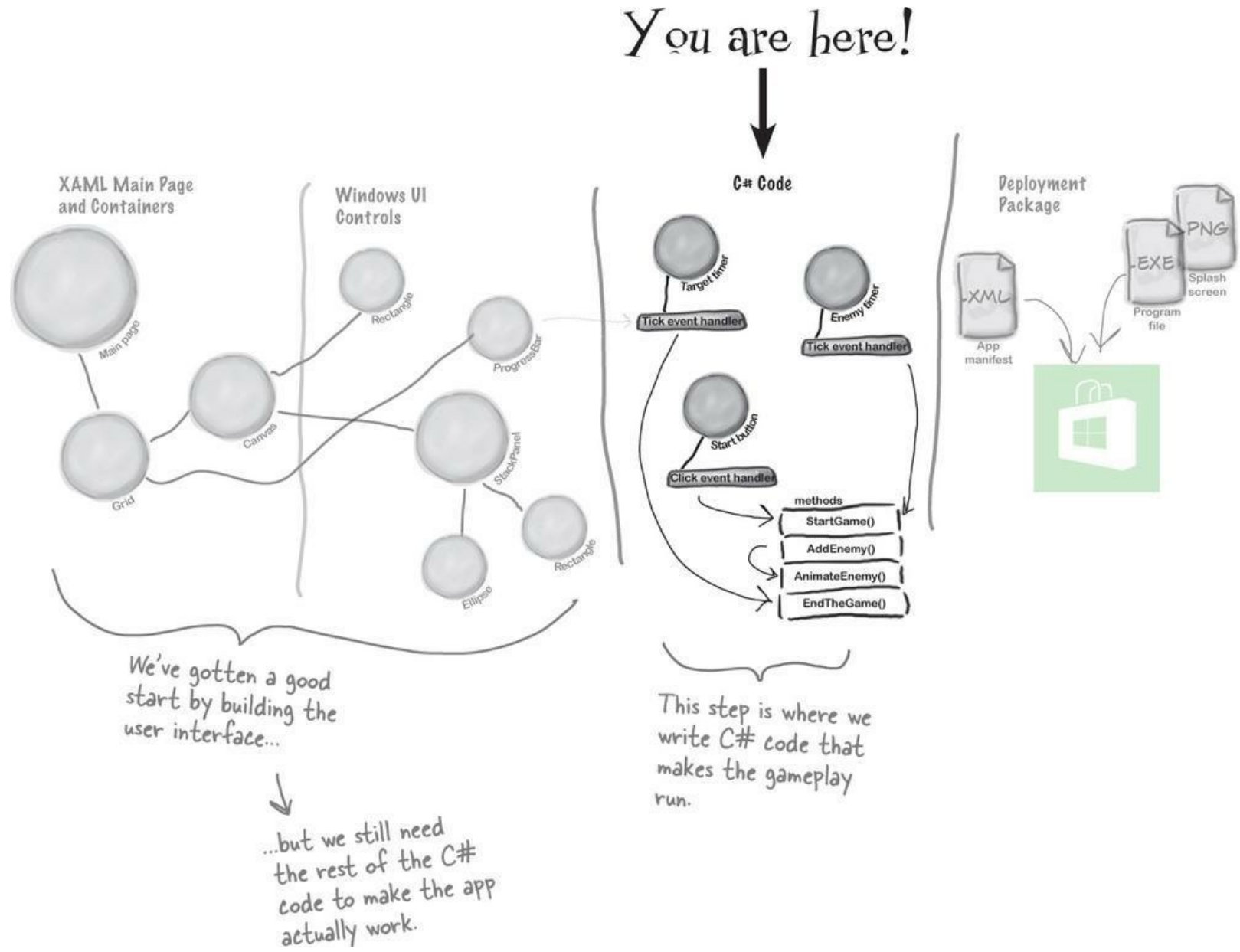
- **❻ Stop your program.**
  Press Alt-Tab to switch back to the IDE. The ▶ button in the toolbar has been replaced with ⏸ ■ ↻ to break, stop, and restart your program. Click the square to stop the program running.

Do you see numbers in the upper corners of the page? Those are frame rate counters.

> **NOTE**
>
> You'll learn more about them in Chapter 10.

Here's what you've done so far

Congratulations! You've built a program that actually does something. It's not quite a playable game, but it's definitely a start. Let's look back and see what you built.

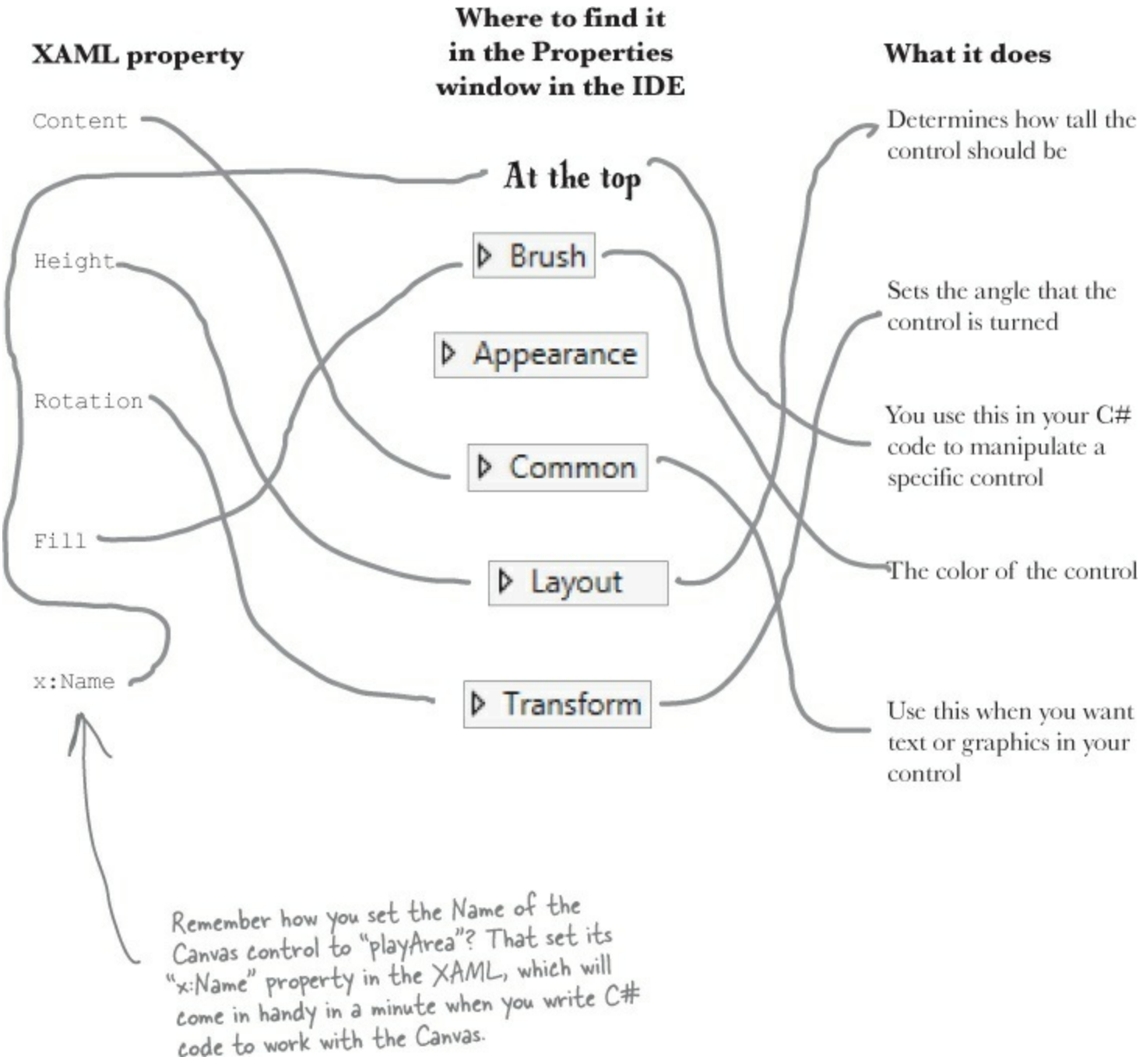


Visual Studio can generate code for you, but you need to know what you want to build <u>BEFORE</u> you start building it. It won't do that for you!

Here's the solution for the "Who Does What" exercise in Who Does What?. We'll give you the anwers to the pencil-and-paper puzzles and exercises, but they won't always be on the next page.

WHO DOES WHAT? SOLUTION

Now that you've built a user interface, you should have a sense of what some of the controls do, and you've used a lot of different properties to customize them. See if you can work out which property does what, and where in the Properties window in the IDE you find it.



Add timers to manage the gameplay

Let's build on that great start by adding working gameplay elements. This game adds more and more enemies, and the progress bar slowly fills up while the player drags the human to the target. You'll use **timers** to manage both of those things.

- **❶ ADD MORE LINES TO THE TOP OF YOUR C# CODE.**

> **NOTE**
>
> The MainPage.Xaml.cs file you've been editing contains the code for a class called MainPage. You'll learn about classes in Chapter 3.

Go up to the top of the file where you added that `Random` line. Add three more lines:

```
/// <summary>
/// A basic page that provides characteristics common to most applications.
/// </summary>
public sealed partial class MainPage : Page
{
    Random random = new Random();
    DispatcherTimer enemyTimer = new DispatcherTimer();
    DispatcherTimer targetTimer = new DispatcherTimer();
    bool humanCaptured = false;
```

Add these three lines below the one you added before. These are fields, and you'll learn about them in Chapter 4.

- **❷ ADD A METHOD FOR ONE OF YOUR TIMERS.**

Find this code that the IDE generated:

```
public MainPage()
{
    this.InitializeComponent();
    this.navigationHelper = new NavigationHelper(this);
    this.navigationHelper.LoadState += navigationHelper_LoadState;
    this.navigationHelper.SaveState += navigationHelper_SaveState;
}
```

Put your cursor right after the last semicolon, hit Enter two times, and type `enemyTimer.` (including the period). As soon as you type the dot, an IntelliSense window will pop up. Choose `Tick` from the IntelliSense window and type the following text. As soon as you enter += the IDE pops up a box:

```
enemyTimer.Tick +=
```

enemyTimer_Tick; (Press TAB to insert)

Press the Tab key. The IDE will pop up another box:

```
enemyTimer.Tick +=enemyTimer_Tick;
```

Press TAB to generate handler 'enemyTimer_Tick' in this class

Press Tab one more time. Here's the code the IDE generated for you:

```
    ...
    enemyTimer.Tick += enemyTimer_Tick;
}

void enemyTimer_Tick(object sender, object e)
{
    throw new NotImplementedException();
}
```

The IDE generated a method for you called an event handler. You'll learn about event handlers in Chapter 15.

Timers "tick" every time interval by calling methods over and over again. You'll use one timer to add enemies every few seconds, and the other to end the game when time expires.

- **❸ FINISH THE MAINPAGE() METHOD.**
  You'll add another `Tick` event handler for the other timer, and you'll add two more lines of code. Here's what your finished `MainPage()` method and the two methods the IDE generated for you should look like:

```
public MainPage()
{
    this.InitializeComponent();
    this.navigationHelper = new NavigationHelper(this);
    this.navigationHelper.LoadState += navigationHelper_LoadState;
    this.navigationHelper.SaveState += navigationHelper_SaveState;

    enemyTimer.Tick += enemyTimer_Tick;
    enemyTimer.Interval = TimeSpan.FromSeconds(2);

    targetTimer.Tick += targetTimer_Tick;
    targetTimer.Interval = TimeSpan.FromSeconds(.1);
}

void targetTimer_Tick(object sender, object e)
{
    throw new NotImplementedException();
}

void enemyTimer_Tick(object sender, object e)
{
    throw new NotImplementedException();
}
```

Try changing these numbers once your game is finished. How does that change the gameplay?

The IDE generated these lines as placeholders when you pressed Tab to add the Tick event handlers. You'll replace them with code that gets run every time the timers tick.

BRAIN POWER

Right now your Start button adds bouncing enemies to the play area. What do you think you'll need to do to make it start the game instead?

- **❹ ADD THE ENDTHEGAME() METHOD.**
  Go to the new `targetTimer_Tick()` method, delete the line that the IDE generated, and add the following code. The IntelliSense window might not seem quite right:

Did the IDE keep trying to capitalize the P in progressBar? That's because there was no lowercase-P progressBar, and the closest match it could find was the type of the control.

```
void targetTimer_Tick(object sender, object e)
{
    progressBar.Value += 1;
    if (progressBar.Value >= progressBar.Maximum)
        EndTheGame();
}
```

Notice how `progressBar` has an error? That's OK. We did this on purpose (and we're not even sorry about it!) to show you what it looks like when you try to use a control that doesn't have a name, or has a typo in the name. Go back to the XAML code (it's in the other tab in the IDE), find the ProgressBar control that you added to the bottom row, and change its name to `progressBar`.

NOTE

If you closed the Designer tab that had the XAML code, double-click on MainPage. xaml in the Solution Explorer window to bring it up.

Next, go back to the code window and generate a method stub for `EndTheGame()`, just like you did a few pages ago for `AddEnemy()`. Here's the code for the new method:

```
private void EndTheGame()
{
    if (!playArea.Children.Contains(gameOverText))
    {
        enemyTimer.Stop();
        targetTimer.Stop();
        humanCaptured = false;
        startButton.Visibility = Visibility.Visible;
        playArea.Children.Add(gameOverText);
    }
}
```

If gameOverText comes up as an error, it means you didn't set the name of the "Game Over" TextBlock. Go back and do it now.

Don't forget the exclamation point in the first line of code! Without it, the *Game Over* text won't show up.

This method ends the game by stopping the timers, making the Start button visible again, and adding the GAME OVER text to the play area.

Make the Start button work

Remember how you made the Start button fire circles into the Canvas? Now you'll fix it so it actually starts the game.

- **❶ Make the Start button start the game.**
  Find the code you added earlier to make the Start button add an enemy. Change it so it looks like this:

```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    StartGame();
}
```

When you change this line, you make the Start button start the game instead of just adding an enemy to the playArea Canvas.

- **❷ Add the StartGame() method.**

> **NOTE**
>
> It's normal to add parentheses () when writing about a method.

Generate a method stub for the `StartGame()` method. Here's the code to fill into the stub method that the IDE added:

```
private void StartGame()
{
    human.IsHitTestVisible = true;
    humanCaptured = false;
    progressBar.Value = 0;
    startButton.Visibility = Visibility.Collapsed;
    playArea.Children.Clear();
    playArea.Children.Add(target);
    playArea.Children.Add(human);
    enemyTimer.Start();
    targetTimer.Start();
}
```

You'll learn about IsHitTestVisible in Chapter 15.

Did you forget to set the names of the target Rectangle or the human StackPanel? You can look a few pages back to make sure you set the right names for all of the controls.

- **❸ Make the enemy timer add the enemies.**
  Find the `enemyTimer_Tick()` method that the IDE added for you and replace its contents with this:

```
void enemyTimer_Tick(object sender, object e)
{
    AddEnemy();
}
```

READY BAKE CODE

We're giving you a lot of code to type in.

By the end of the book, you'll know what all of this code does — in fact, you'll be able to write code just like it on your own.

For now, your job is to make sure you enter each line accurately, and to follow the instructions exactly. This will get you used to entering code, and will help give you a feel for the ins and outs of the IDE.

NOTE

Once you're used to working with code, you'll be good at spotting those missing parentheses, semicolons, etc.

Are you seeing errors in the Error List window that don't make sense? One misplaced comma or semicolon can cause two, three, four, or more errors to show up. Double check your code.

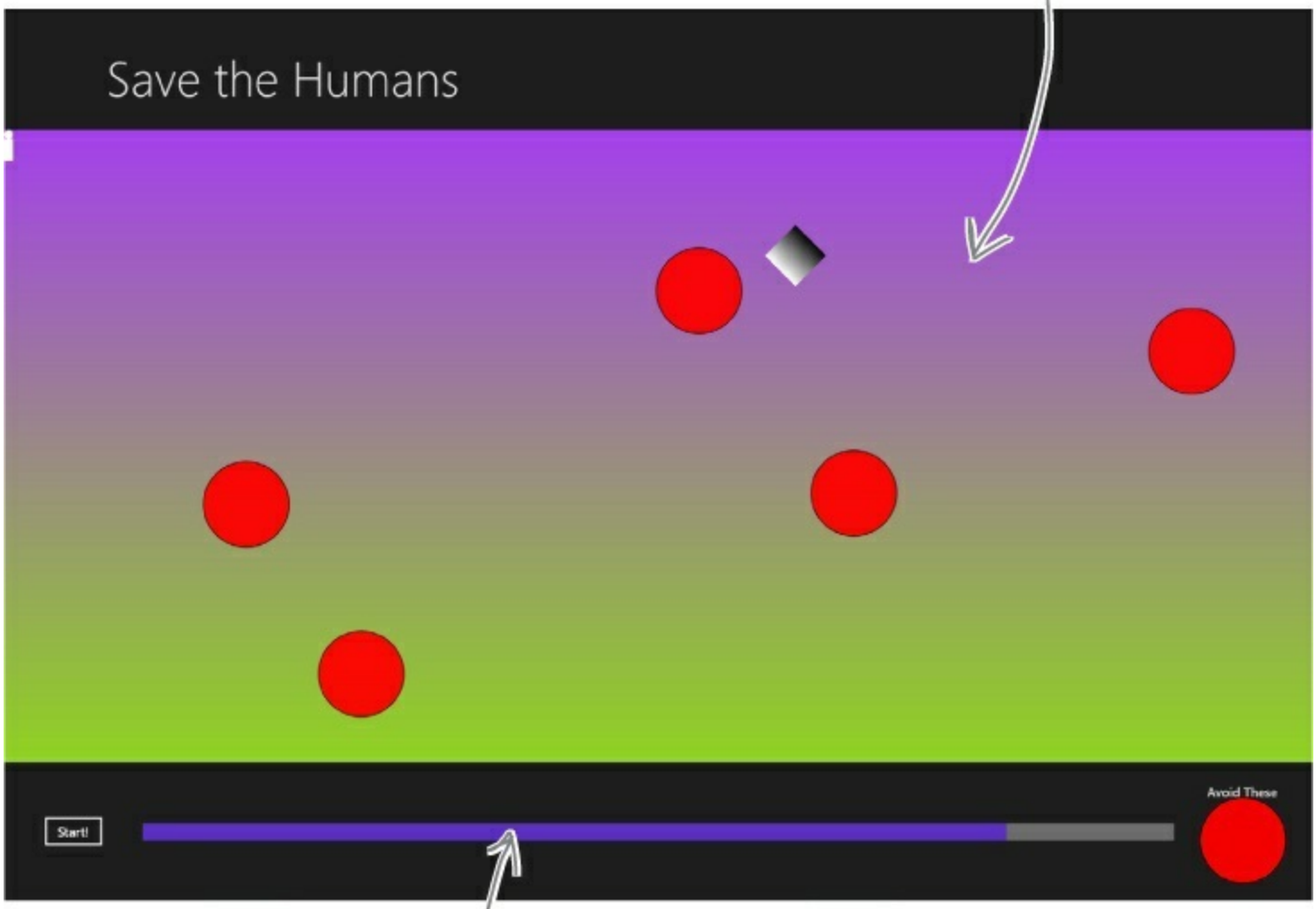
Run the program to see your progress

Your game is coming along. Run it again to see how it's shaping up.

NOTE

When you press the "Start!" button, it disappears, clears the enemies, and starts the progress bar filling up.

The play area slowly starts to fill up with bouncing enemies.



When the progress bar at the bottom fills up, the game ends and the Game Over text is displayed.

The target timer should fill up slowly, and the enemies should appear every two seconds. If the timing is off, make sure you added all of the lines to the MainPage() method.

BRAIN POWER

What do you think you'll need to do to get the rest of your game working?

Flip the page to find out! ⟶

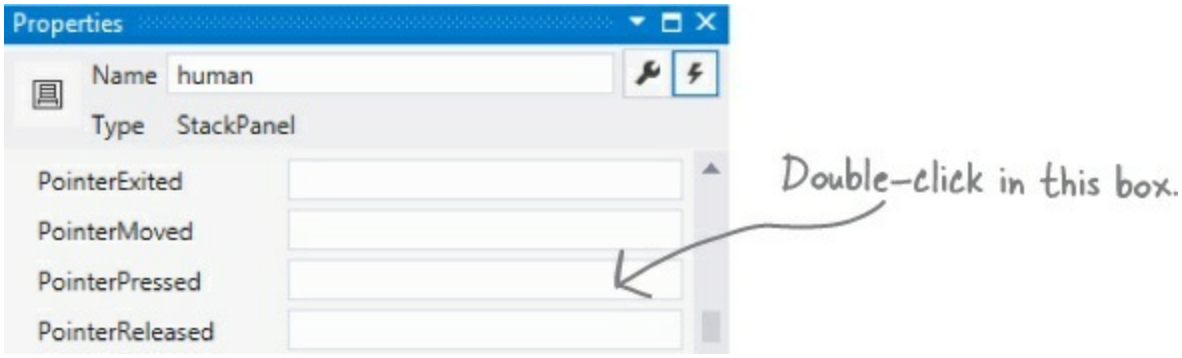
Add code to make your controls interact with the player

Make sure you switch back to the IDE and stop the app before you make more changes to the code.

You've got a human that the player needs to drag to the target, and a target that has to sense when the human's been dragged to it. It's time to add code to make those things work.

- ❶ Go to the XAML designer and use the Document Outline window to select `human` (remember, it's the StackPanel that contains a Circle and a Rectangle). Then go to the Properties window and press the ⚡ button to switch it to show event handlers. Find the PointerPressed row and double-click in the empty box.

> **NOTE**
>
> You'll learn more about the event handlers in the Properties window in Chapter 4.
>
> The Document Outline may have collapsed [Grid], playArea, and other lines. If it did, just expand them to find the human control.



Now go back and check out what the IDE added to your XAML for the StackPanel:

```
<StackPanel x:Name="human" Orientation="Vertical" PointerPressed="human_PointerPressed">
```

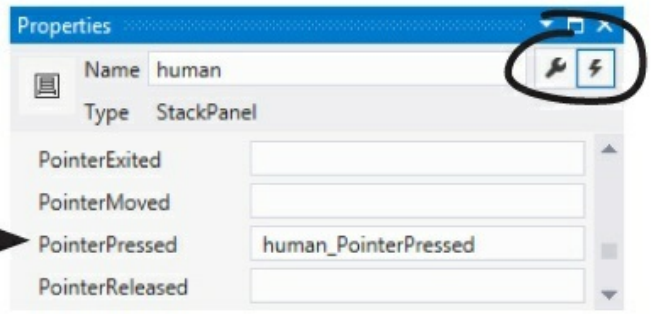
It also generated a method stub for you. Right-click on `human_PointerPressed` in the XAML and choose "Navigate to Event Handler" to jump straight to the C# code:

```
private void human_PointerPressed(object sender, PointerRoutedEventArgs e)
{

}
```

- ❷ Fill in the C# code:

```
private void human_PointerPressed(object sender, PointerRoutedEventArgs e)
{
    if (enemyTimer.IsEnabled)
    {
        humanCaptured = true;
        human.IsHitTestVisible = false;
    }
}
```

You can use these buttons to switch between showing properties and event handlers in the Properites window.



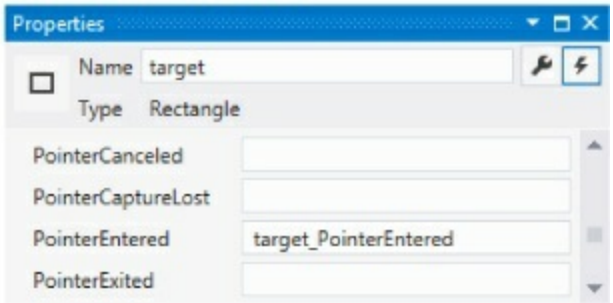
If you go back to the designer and click on the StackPanel again, you'll see that the IDE filled in the name of the new event handler method. You'll be adding more event handler methods the same way.

- ❸ Use the Document Outline window to select the Rectangle named `target`, then use the event handlers view of the Properties window to add a `PointerEntered` event handler. Here's the code for the method:

> **NOTE**
>
> Make sure you add the right event handler! You added a PointerPressed event handler to the human, but now you're adding a PointerEntered event handler to the target.

> **When the Properties window is in the mode where it displays event handlers, double-clicking on an empty event handler box causes the IDE to add a method stub for it.**

```
private void target_PointerEntered(object sender, PointerRoutedEventArgs e)
{
    if (targetTimer.IsEnabled && humanCaptured)
    {
        progressBar.Value = 0;
        Canvas.SetLeft(target, random.Next(100, (int)playArea.ActualWidth - 100));
        Canvas.SetTop(target, random.Next(100, (int)playArea.ActualHeight - 100));
        Canvas.SetLeft(human, random.Next(100, (int)playArea.ActualWidth - 100));
        Canvas.SetTop(human, random.Next(100, (int)playArea.ActualHeight - 100));
        humanCaptured = false;
        human.IsHitTestVisible = true;
    }
}
```



- ❹ Now you'll add two more event handlers, this time to the `playArea` Canvas control. You'll need to find the right `[Grid]` in the Document Outline (there are two of them — use the child grid that's indented under the main grid for the page) and set its name to `grid`. Then you can add these

event handlers to playArea:

> **NOTE**
>
> You'll need to switch your Properties window back to show properties instead of event handlers.

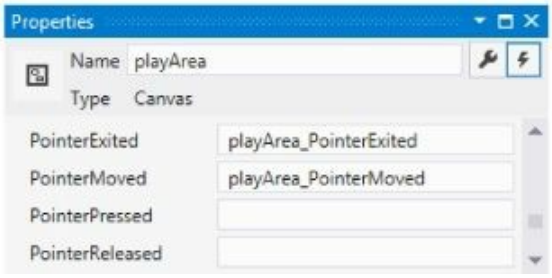
```
private void playArea_PointerMoved(object sender, PointerRoutedEventArgs e)
{
    if (humanCaptured)
    {
        Point pointerPosition = e.GetCurrentPoint(null).Position;
        Point relativePosition = grid.TransformToVisual(playArea).TransformPoint(pointerPosition);
        if ((Math.Abs(relativePosition.X - Canvas.GetLeft(human)) > human.ActualWidth * 3)
            || (Math.Abs(relativePosition.Y - Canvas.GetTop(human)) > human.ActualHeight * 3))
        {
            humanCaptured = false;
            human.IsHitTestVisible = true;
        }
        else
        {
            Canvas.SetLeft(human, relativePosition.X - human.ActualWidth / 2);
            Canvas.SetTop(human, relativePosition.Y - human.ActualHeight / 2);
        }
    }
}

private void playArea_PointerExited(object sender, PointerRoutedEventArgs e)
{
    if (humanCaptured)
        EndTheGame();
}
```

That's a lot of parentheses! Be really careful and get them right.

These two vertical bars are a logical operator. You'll learn about them in Chapter 2.

You can make the game more or less sensitive by changing these 3s to a lower or higher number.

Make sure you put the right code in the correct event handler! Don't accidentally swap them.



Dragging humans onto enemies ends the game

When the player drags the human into an enemy, the game should end. Let's add the code to do that. Go to your `AddEnemy()` method and add one more line of code to the end. Use the IntelliSense window to fill in `enemy.PointerEntered` from the list:

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
    enemy.Template = Resources["EnemyTemplate"] as ControlTemplate;
    AnimateEnemy(enemy, 0, playArea.ActualWidth - 100, "(Canvas.Left)");
    AnimateEnemy(enemy, random.Next((int)playArea.ActualHeight - 100),
        random.Next((int)(playArea.ActualHeight - 100)), "(Canvas.Top)");
    playArea.Children.Add(enemy);

    enemy.Pointer
}
```

Here's the last line of your AddEnemy() method. Put your cursor at the end of the line and hit Enter to add the new line of code.

Start typing this line of code. As soon as you enter the dot, an IntelliSense window will pop up. Keep typing "Pointer" to jump down to the entries in the list that start with "Pointer..."

PointerEventHandler UIElement.PointerEntered
Occurs when a pointer enters the hit test area of this element.

Choose `PointerEntered` from the list. (If you choose the wrong one, don't worry — just backspace over it to delete everything past the dot. Then enter the dot again to bring up the IntelliSense window.)

Next, add an event handler, just like you did before. Type += and then press Tab:

```
enemy.PointerEntered +=
```

enemy_PointerEntered; (Press TAB to insert)

You'll learn all about how event handlers like this work in Chapter 15.

Then press Tab again to generate the stub for your event handler:

```
enemy.PointerEntered += enemy_PointerEntered;
```

Press TAB to generate handler 'enemy_PointerEntered' in this class

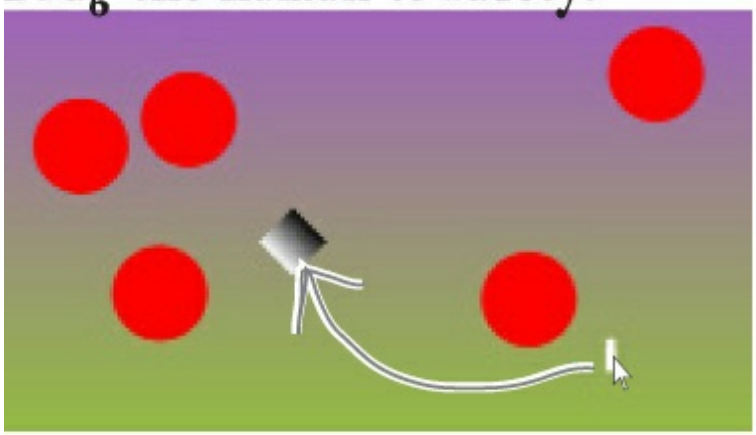
Now you can go to the new method that the IDE generated for you and fill in the code:

```
void enemy_PointerEntered(object sender, PointerRoutedEventArgs e)
{
    if (humanCaptured)
        EndTheGame();
}
```

Your game is now playable

Run your game — it's almost done! When you click the Start button, your play area is cleared of any enemies, and only the human and target remain. You have to get the human to the target before the progress bar fills up. Simple at first, but it gets harder as the screen fills with dangerous alien enemies!

Drag the human to safety!

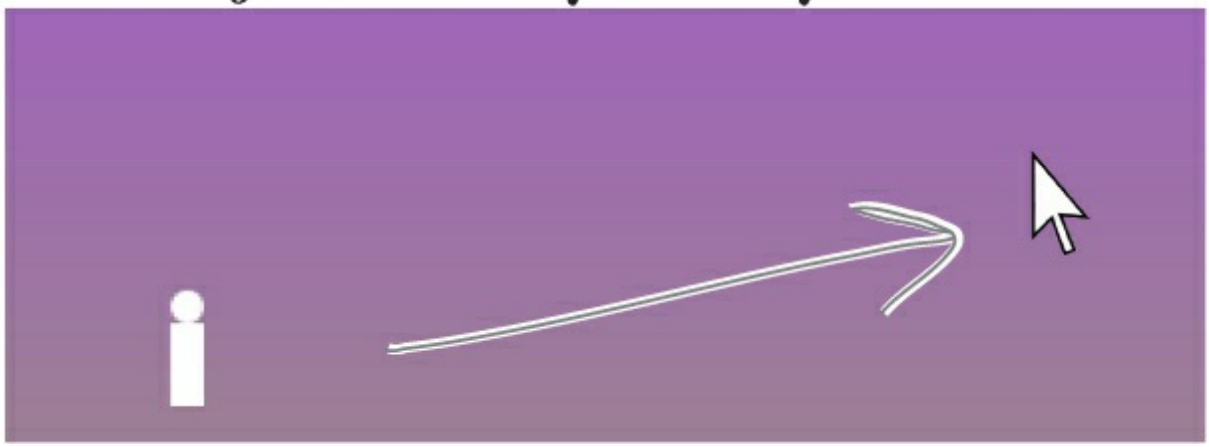


The aliens only spend their time patrolling for moving humans, so the game only ends if you drag a human onto an enemy. Once you release the human, he's temporarily safe from aliens.

Look through the code and find where you set the IsHitTestVisible property on the human. When it's on, the human intercepts the PointerEntered event because the human's StackPanel control is sitting between the enemy and the pointer.

Get him to the target before time's up...

...but drag too fast, and you'll lose your human!



Make your enemies look like aliens

WATCH IT!

Seeing events instead of properties?

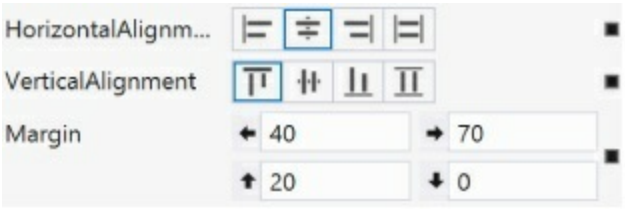
You can toggle the Properties window between displaying properties or events for the selected control by clicking the wrench or lightning bolt icons.

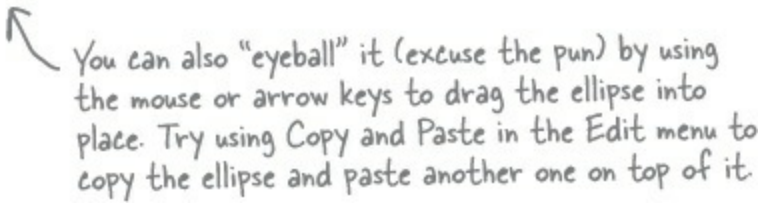
Red circles aren't exactly menacing. Luckily, you used a template. All you need to do is update it.

- ❶ Go to the Document Outline, right-click on the ContentControl, choose Edit Template, and then Edit Current to edit the template. You'll see the template in the XAML window. Edit the XAML code for the ellipse to set the width to `75` and the fill to `Gray`. Then add `Stroke="Black"` to add a black outline (if it's not already there), and reset its vertical and horizontal alignments. Here's what it should look like (you can delete any additional properties that may have inadvertently been added while you worked on it):

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

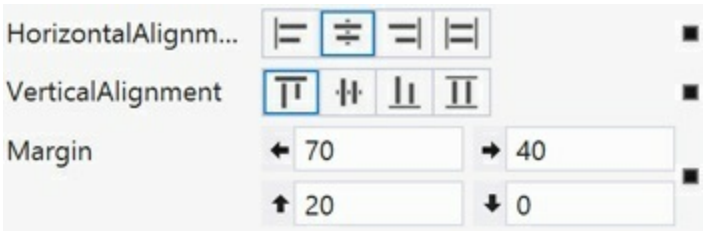
- ❷ Drag another Ellipse control out of the toolbox on top of the existing ellipse. Change its **Fill** to black, set its width to `25`, and its height to `35`. Set the alignment and margins like this:





- ❸ Use the button in the Transforms section of the Properties window to add a Skew transform:

- ❹ Drag one more Ellipse control out of the toolbox on top of the existing ellipse. Change its fill to `Black`, set its width to `25`, and set its height to `35`. Set the alignment and margins like this:



and add a skew like this:

Now your enemies look a lot more like human-eating aliens.

```
<ControlTemplate x:Key="EnemyTemplate" TargetType="ContentControl">
    <Grid>
        <Ellipse Fill="Gray" Stroke="Black" Height="100" Width="75"/>
        <Ellipse Fill="Black" Stroke="Black" Height="35" Width="25"
                 HorizontalAlignment="Center" VerticalAlignment="Top"
                 Margin="40,20,70,0" RenderTransformOrigin="0.5,0.5">
            <Ellipse.RenderTransform>
                <CompositeTransform SkewX="10"/>
            </Ellipse.RenderTransform>
        </Ellipse>
        <Ellipse Fill="Black" Stroke="Black" Height="35"  Width="25"
                 HorizontalAlignment="Center" VerticalAlignment="Top"
                 Margin="70,20,40,0" RenderTransformOrigin="0.5,0.5">
            <Ellipse.RenderTransform>
                <CompositeTransform SkewX="-10"/>
            </Ellipse.RenderTransform>
        </Ellipse>
    </Grid>
</ControlTemplate>
```

Here's the updated XAML for the new enemy template that you created.

THERE'S JUST ONE MORE THING YOU NEED TO DO...

PLAY YOUR GAME!

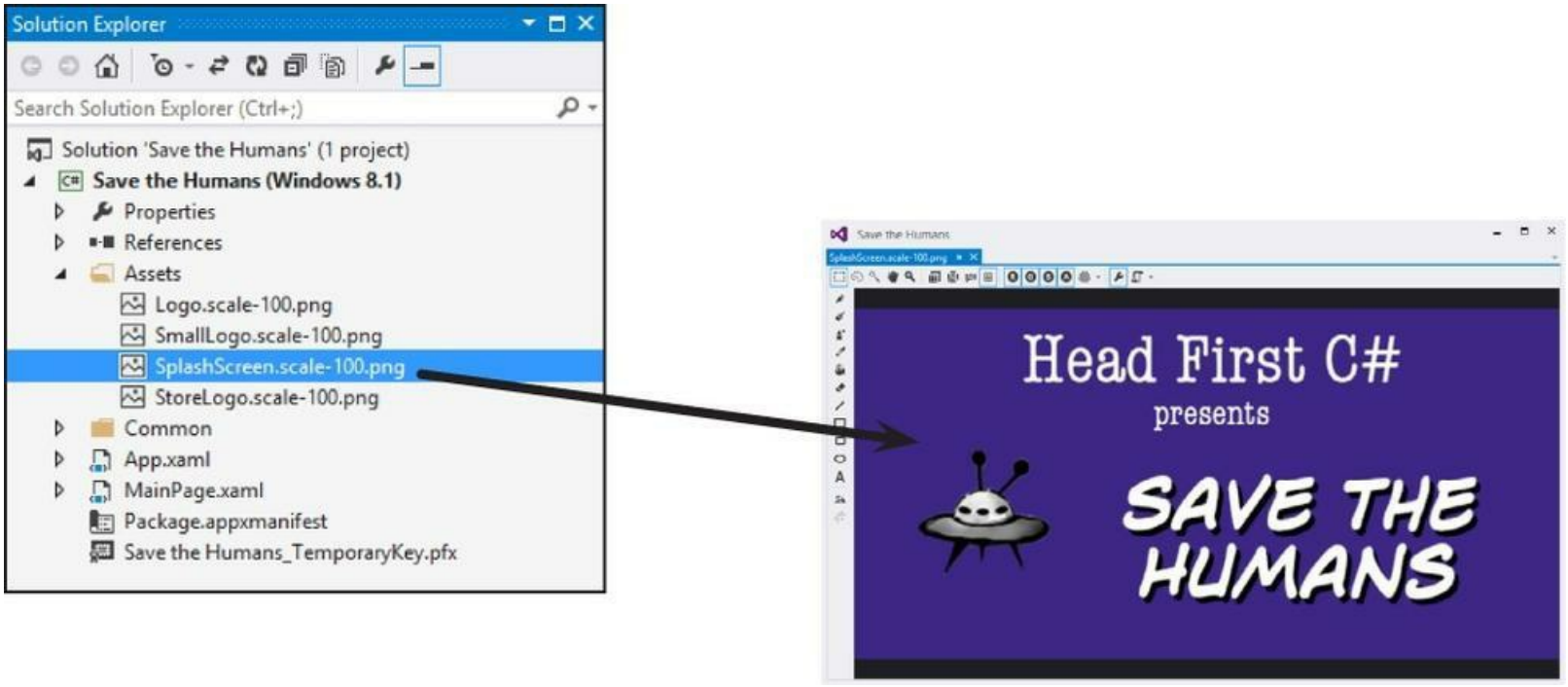
See if you can get creative and change the way the human, target, play area, and enemies look.

And don't forget to step back and really appreciate what you built. Good job!

Add a splash screen and a tile

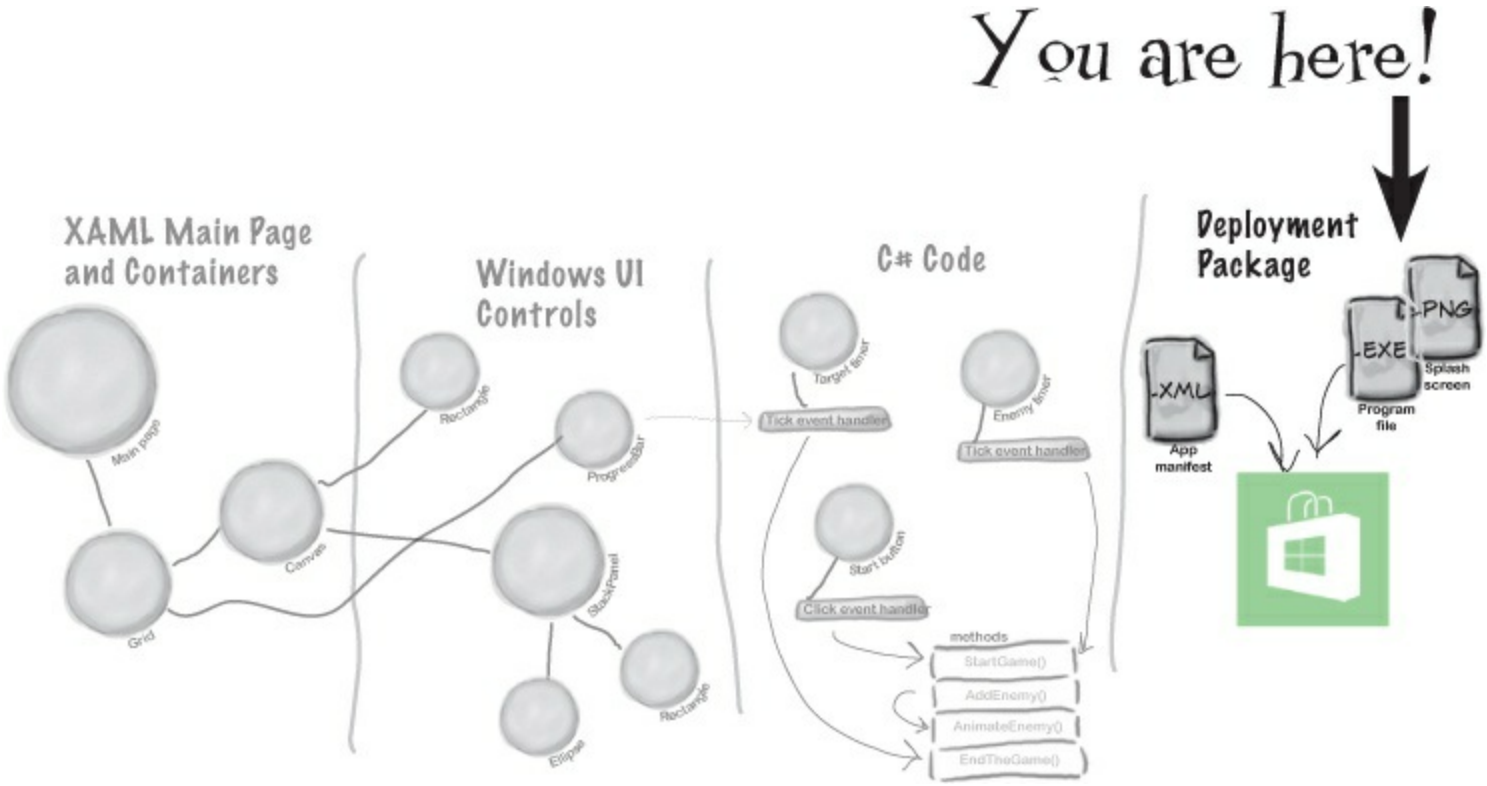
That big X that appears when you start your program is a splash screen. And when you go back to the Windows Start page, there it is again in the tile. Let's change these things.

Expand the Assets folder in the Solution Explorer window and you'll see four files. Double-click each of them to edit them in the Visual Studio graphics editor. Edit *SplashScreen.scale-100.png* to create a splash screen that's displayed when the game starts. *Logo.scale-100.png* and *SmallLogo.scale-100.png* are displayed in the Start screen. And when your app is displayed in the search results (or in the Windows Store!), it displays *StoreLogo.scale-100.png*.



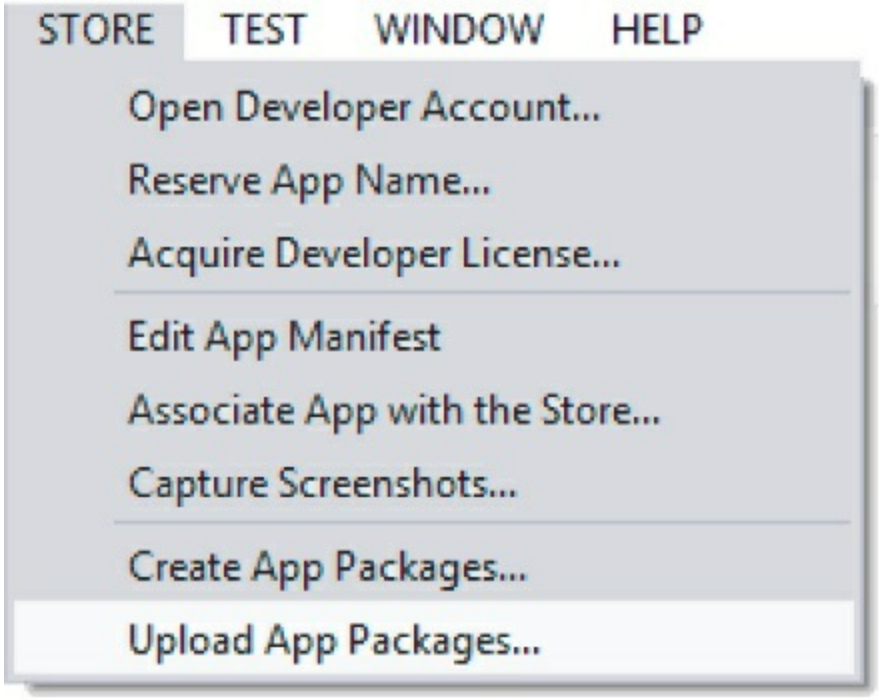
Publish your app

You should be pretty pleased with your app! Now it's time to deploy it. When you publish your app to the Windows Store, you make it available to millions of potential users. The IDE can help guide you through the steps to publish your app to the Windows Store.



Here's what it takes to get your app out there:

- ❶ Open a Windows Store developer account.
- ❷ Choose your app's name, set an age rating, write a description, and choose a business model to determine if your app is free, ad-supported, or has a price.
- ❸ Test your app using the Windows App Certification Kit to identify and fix any problems.
- ❹ **Submit your app to the Store!** Once it's accepted, millions of people around the world can find and download it.



The Store menu in the IDE has all of the tools you need to publish your app.

In some editions of Visual Studio, the Windows Store options appear under the Project menu instead of having their own top-level Store menu.

NOTE

Throughout the book we'll show you where to find more information from MSDN, the Microsoft Developer Network. This is a really valuable resource that helps you keep expanding your knowledge.

You can learn more about how to publish apps to the Windows Store here: http://msdn.microsoft.com/en-us/library/windows/apps/jj657972.aspx

Use the Remote Debugger to sideload your app

Sometimes you want to run your app on a remote machine without publishing it to the Windows Store. When you install your app on a machine without going through the Windows Store it's called **sideloading**, and one of the easiest ways to do it is to install the **Visual Studio Remote Debugger** on another computer.

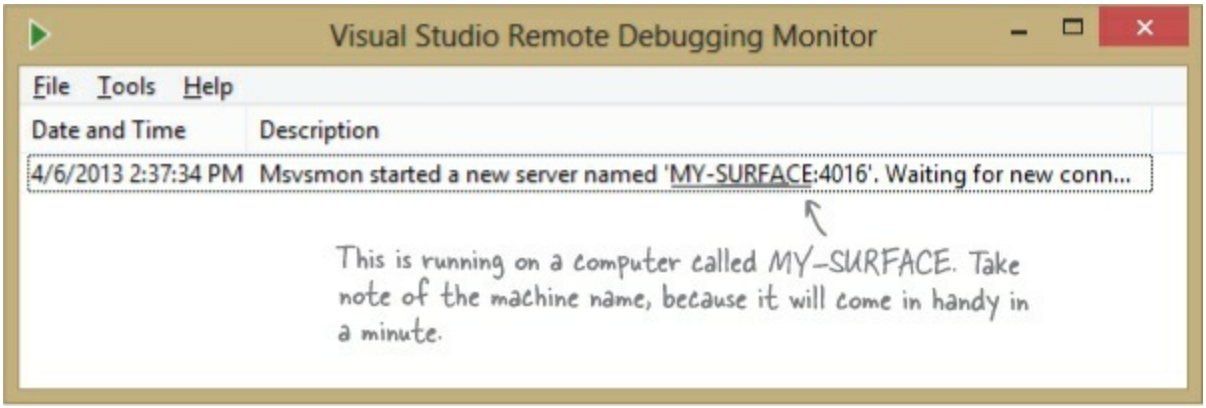
Here's how to get your app loaded using the Remote Debugger:

- Make sure the remote machine is running Windows 8.
- Go to the Microsoft Download Center (*http://www.microsoft.com/en-hk/download/default.aspx*) on the remote machine and search for "Remote Tools for Visual Studio" to find the download page.

> **NOTE**
>
> At the time this is being written, you'll find "Remote Tools for Visual Studio 2013," but you may find future updates.

- Download the installer for your machine's architecture (x86, x64, ARM) and run it to install the remote tools.
- Go to the Start page and launch the Remote Debugger. (You may need to search for the app if there's no icon.)

- If your computer's network configuration needs to change, it may pop up a wizard to help with that. Once it's running, you'll see the Visual Studio Remote Debugging Monitor window:



- Your remote computer is now running the Visual Studio Remote Debugging Monitor and waiting for incoming connections from Visual Studio on your development machine.

If you have an odd network setup, you may have trouble running the remote debugger. This MDSN page can help you get it set up: http://msdn.microsoft.com/en-us/library/vstudio/bt727f1t.aspx

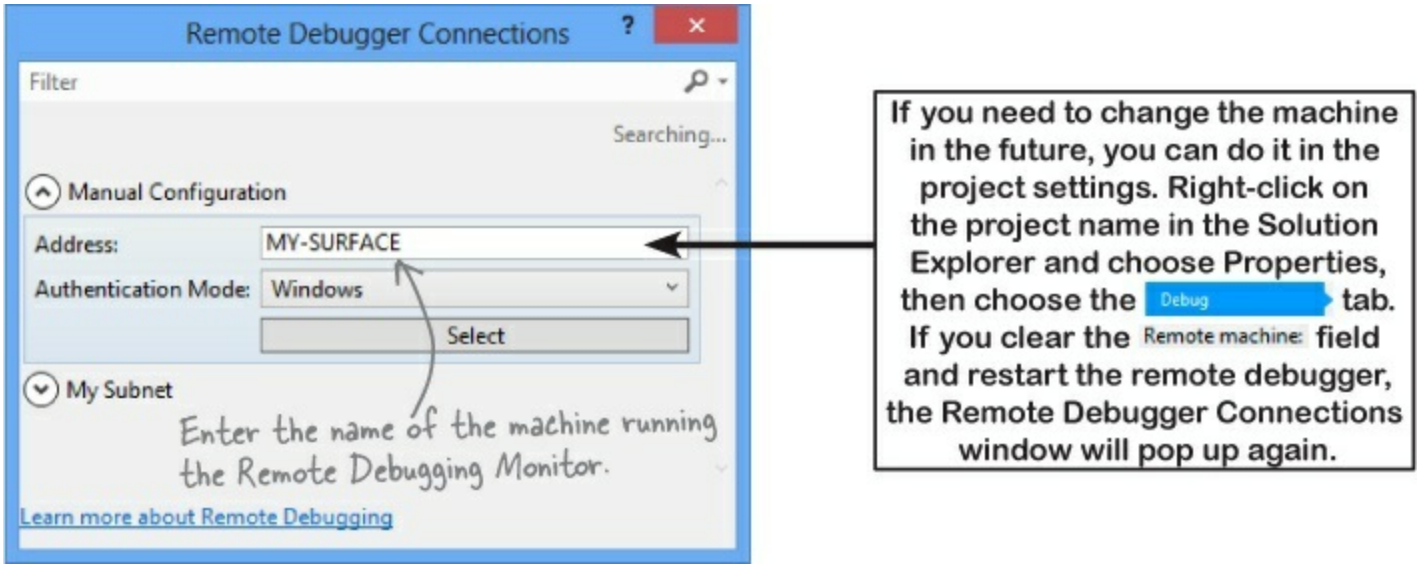
Flip to get your app up and running on the remote computer! →

Start remote debugging

Once you've got a remote computer running the remote debugging monitor, you can launch the app from Visual Studio to install and run it. This will automatically sideload your app on the computer, and you'll be able to run it again from the Start page any time you want.

- **❶ CHOOSE "REMOTE MACHINE" FROM THE DEBUG DROP-DOWN.**
  You can use the Debug drop-down to tell the IDE to run your program on a remote machine. Take a close look at the ▶ Local Machine button you've been using to run your program — you'll see a drop-down (▾). Click it to show the drop-down and choose Remote Machine:

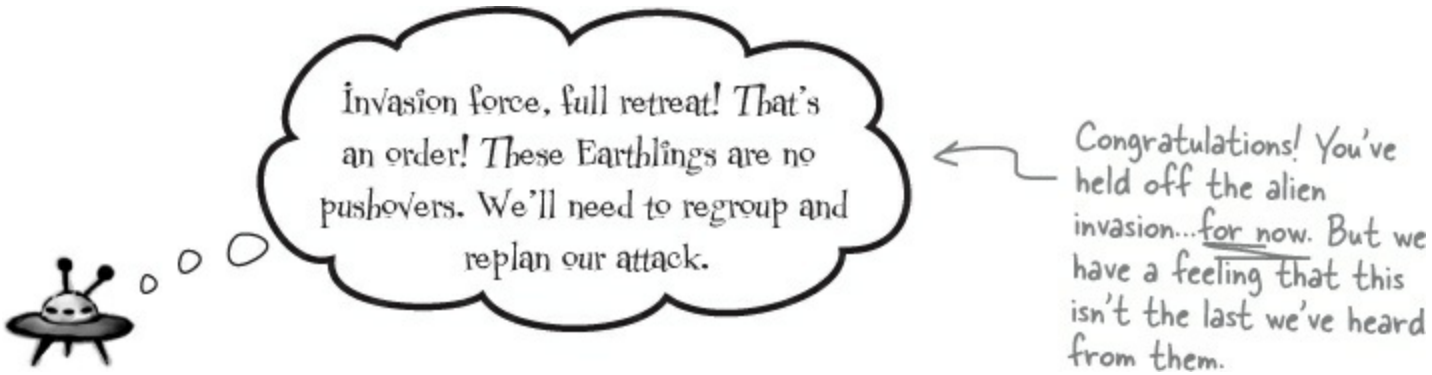
- **❷ RUN YOUR PROGRAM ON THE REMOTE MACHINE.**
  Now run your program by clicking the ▶ button. The IDE will pop up a window asking for the machine to run on. If it doesn't detect it in your subnet, you can enter the machine name manually:



- **❸ ENTER YOUR CREDENTIALS.**
  You'll be prompted to enter the username and password of the user on the remote machine. You can turn off authentication in the Remote Debugging Monitor if you want to avoid this (but that's not a great idea, because then anyone can run programs on your machine remotely!).

- **❹ GET YOUR DEVELOPER LICENSE.**
  You already obtained a free developer license from Microsoft when you installed Visual Studio. You need that license in order to sideload apps onto a machine. Luckily, the Remote Debugging Monitor will pop up a wizard to get it automatically.

- **❺ NOW...SAVE SOME HUMANS!**
  Once you get through that setup, your program will start running on the remote machine. Since it's sideloaded, if you want to run it again you can just run it from the Windows Start page. Congratulations, you've built your first Windows Store app and loaded it onto another computer!



Chapter 2. It's all Just Code: Under the hood

You're a programmer, not just an IDE user.

You can get a lot of work done using the IDE. But there's only so far it can take you. Sure, there are a lot of **repetitive tasks** that you do when you build an application. And the IDE is great at doing those things for you. But working with the IDE is *only the beginning*. You can get your programs to do so much more — and **writing C# code** is how you do it. Once you get the hang of coding, there's *nothing* your programs can't do.

When you're doing this...

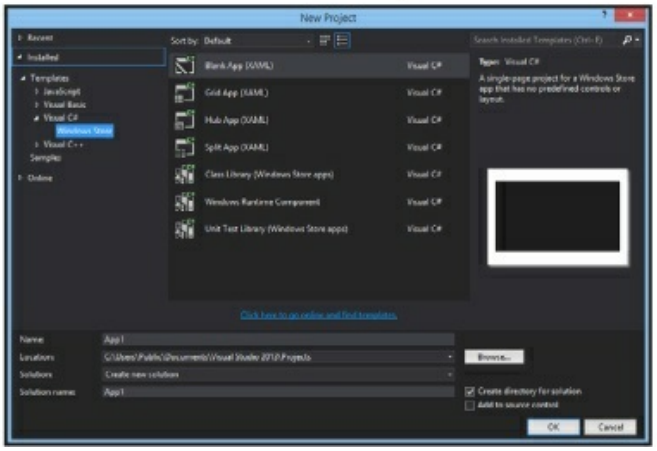
The IDE is a powerful tool — but that's all it is, a *tool* for you to use. Every time you change your project or drag and drop something in the IDE, it creates code automatically. It's really good at writing **boilerplate** code, or code that can be reused easily without requiring much customization.

Let's look at what the IDE does in a typical application development, when you're...

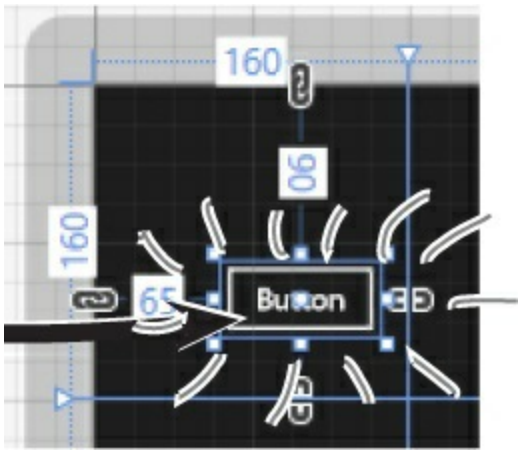
- **❶ CREATING A WINDOWS STORE PROJECT**
  There are several kinds of applications the IDE lets you build. We'll be concentrating on Windows Store applications for now — you'll learn about other kinds of applications in the next chapter.

All of these tasks have to do with standard actions and boilerplate code. Those are the things the IDE is great for helping with.

In Chapter 1, you created a blank **Windows Store** project— that told the IDE to create an empty page and add it to your new project.

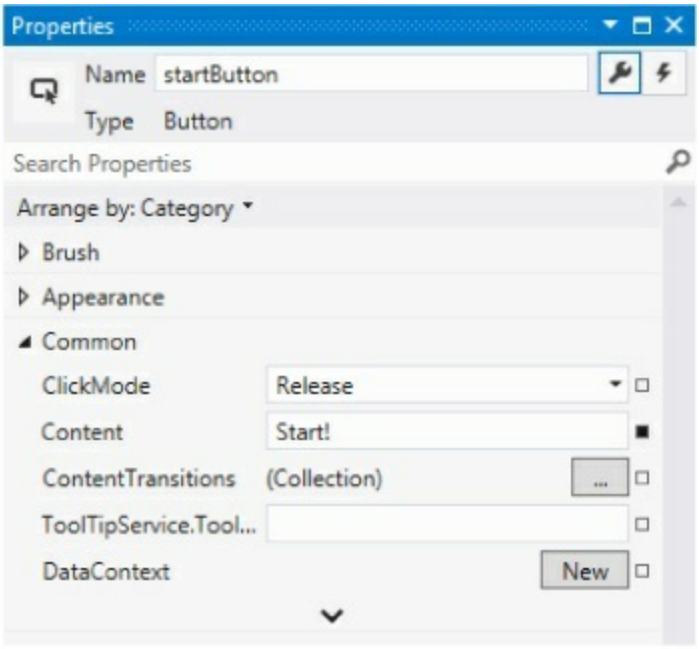


- **❷ DRAGGING A CONTROL OUT OF THE TOOLBOX AND ONTO YOUR PAGE, AND THEN DOUBLE-CLICKING IT**
  Controls are how you make things happen in your page. In this chapter, we'll use Button controls to explore various parts of the C# language.



- **❸ SETTING A PROPERTY ON YOUR PAGE**
  The **Properties window** in the IDE is a really powerful tool that you can use to change attributes

of just about everything in your program: all visual and functional properties for the controls on your page, and even options on your project itself.



The **Properties window** in the IDE is a really easy way to edit a specific chunk of XAML code in MainPage.xaml automatically, and it can save you time. Use the **Alt-Enter** shortcut to open the Properties window if it's closed.

...the IDE does this

Every time you make a change in the IDE, it makes a change to the code, which means it changes the files that contain that code. Sometimes it just modifies a few lines, but other times it adds entire files to your project.

- **❶ ...THE IDE CREATES THE FILES AND FOLDERS FOR THE PROJECT.**

These files are created from a predefined template that contains the basic code to create and display a page.

- **❷ ...THE IDE ADDS CODE TO MAINPAGE.XAML THAT ADDS A BUTTON, AND THEN ADDS A METHOD TO MAINPAGE.XAML.CS THAT GETS RUN ANY TIME THE BUTTON IS CLICKED.**

MainPage.xaml

```
private void startButton_Click(object sender, RoutedEventArgs e)
{

}
```

The IDE knows how to add an empty method to handle a button click. But it doesn't know what to put inside it—that's your job.

MainPage.xaml.cs

- **❸ ...THE IDE OPENS THE MAINPAGE.XAML FILE AND UPDATES A LINE OF XAML CODE.**

The IDE went into this file...

```
<Button x:Name="startButton"
                Content="Start!"
                HorizontalAlignment="Center"
                VerticalAlignment="Center" Click="startButton_Click"/>
```

MainPage.xaml

...and updated this XAML code.

Where programs come from

A C# program may start out as statements in a bunch of files, but it ends up as a program running in your computer. Here's how it gets there.

Every program starts out as source code files

You've already seen how to edit a program, and how the IDE saves your program to files in a folder. Those files **are** your program — you can copy them to a new folder and open them up, and everything will be there: pages, resources, code, and anything else you added to your project.

You can think of the IDE as a kind of fancy file editor. It automatically does the indenting for you, changes the colors of the keywords, matches up brackets for you, and even suggests what words might come next. But in the end, all the IDE does is edit the files that contain your program.

NOTE

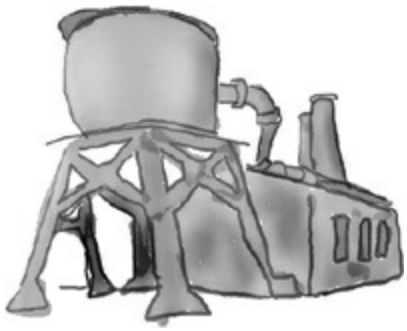
There's no reason you couldn't build your programs in Notepad, but it'd be a lot more time-consuming.

The IDE bundles all of the files for your program into a **solution** by creating a solution (*.sln*) file and a folder that contains all of the other files for the program. The solution file has a list of the project files (which end in *.csproj*) in the solution, and the project files contain lists of all the other files associated with the program. In this book, you'll be building solutions that only have one project in them, but you can easily add other projects to your solution using the IDE's Solution Explorer.

Build the program to create an executable

When you select Build Solution from the Build menu, the IDE **compiles** your program. It does this by running the **compiler**, which is a tool that reads your program's source code and turns it into an **executable**. The executable is a file on your disk that ends in *.exe* — that's the actual program that Windows runs. When you build the program, it creates the executable inside the *bin* folder, which is inside the project folder. When you publish your solution, it copies the executable (and any other files necessary) into a package that can be uploaded to the Windows Store or sideloaded.

When you select Start Debugging from the Debug menu, the IDE compiles your program and runs the executable. It's got some more advanced tools for **debugging** your program, which just means running it and being able to pause (or "break") it so you can figure out what's going on.



The .NET Framework gives you the right tools for the job

C# is just a language — by itself, it can't actually ***do*** anything. And that's where the **.NET Framework** comes in. Those controls you dragged out of the toolbox? Those are all part of a library of tools, classes, methods, and other useful things. It's got visual tools like the XAML toolbox controls you used, and other useful things like the DispatcherTimer that made your *Save the Humans* game work.

All of the controls you used are part of **.NET for Windows Store apps**, which contains an API with grids, buttons, pages, and other tools for building Windows Store apps. But for a few chapters starting with Chapter 3, you'll learn all about writing desktop applications, which are built using tools from the **.NET for Windows Desktop** (which some people call "WinForms"). It's got tools to build desktop applications from windows that hold forms with checkboxes, buttons, and lists. It can draw graphics, read and write files, manage collections of things...all sorts of tools for a lot of jobs that programmers have to do every day. The funny thing is that Windows Store apps need to do those things, too! One of the things you'll learn by the end of this book is how Windows Store and Windows Desktop apps do some of those things differently. That's the kind of insight and understanding that helps *good* programmers become *great* programmers.

NOTE

An API, or Application Programming Interface, is a collection of code tools that you use to access or control a system. Many systems have APIs, but they're especially important for operating systems like Windows.

The tools in both the Windows Runtime and the .NET Framework are divided up into **namespaces**. You've seen these namespaces before, at the top of your code in the "using" lines. One namespace is called `Windows.UI.Xaml.Conrols` — it's where your buttons, checkboxes, and other controls come from. Whenever you create a new Windows Store project, the IDE will add the necessary files so that your project contains a page, and those files have the line "`using Windows.UI.Xaml.Controls;`" at the top.

You can see an overview of .NET for Windows Store apps here: http://msdn.microsoft.com/en-us/library/windows/apps/br230302.aspx

Your program runs inside the Common Language Runtime

Every program in Windows 8 runs on an architecture called the Windows Runtime. But there's an extra "layer" between the Windows Runtime and your program called the **Common Language Runtime**, or CLR. Once upon a time, not so long ago (but before C# was around), writing programs was harder, because you had to deal with hardware and low-level machine stuff. You never knew exactly how someone was going to configure his computer. The CLR — often referred to as a **virtual machine** — takes care of all that for you by doing a sort of "translation" between your program and the computer running it.

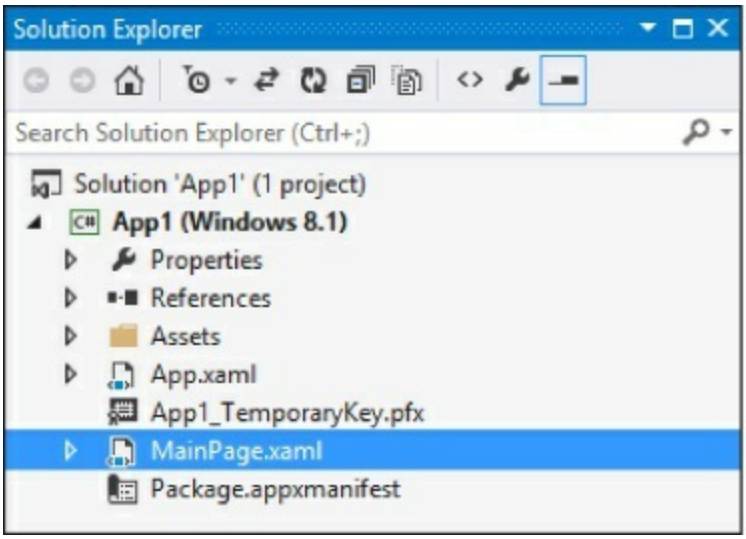
> **NOTE**
>
> You don't really have to worry about the CLR much right now. It's enough to know it's there, and takes care of running your program for you automatically. You'll learn more about it as you go.

You'll learn about all sorts of things the CLR does for you. For example, it tightly manages your computer's memory by figuring out when your program is finished with certain pieces of data and getting rid of them for you. That's something programmers used to have to do themselves, and it's something that you don't have to be bothered with. You won't know it at the time, but the CLR will make your job of learning C# a whole lot easier.

The IDE helps you code

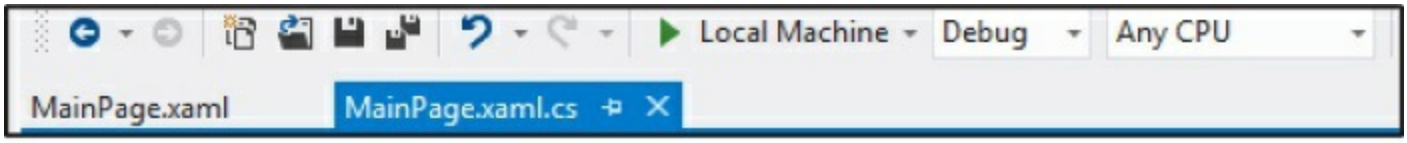
You've already seen many of the things that the IDE can do. Let's take a closer look at some of the tools it gives you, to make sure you're starting off with all the tools you need.

- **THE SOLUTION EXPLORER SHOWS YOU EVERYTHING IN YOUR PROJECT**
  You'll spend a lot of time going back and forth between classes, and the easiest way to do that is to use the Solution Explorer. Here's what the Solution Explorer looked like after creating a blank Windows Application called App1:



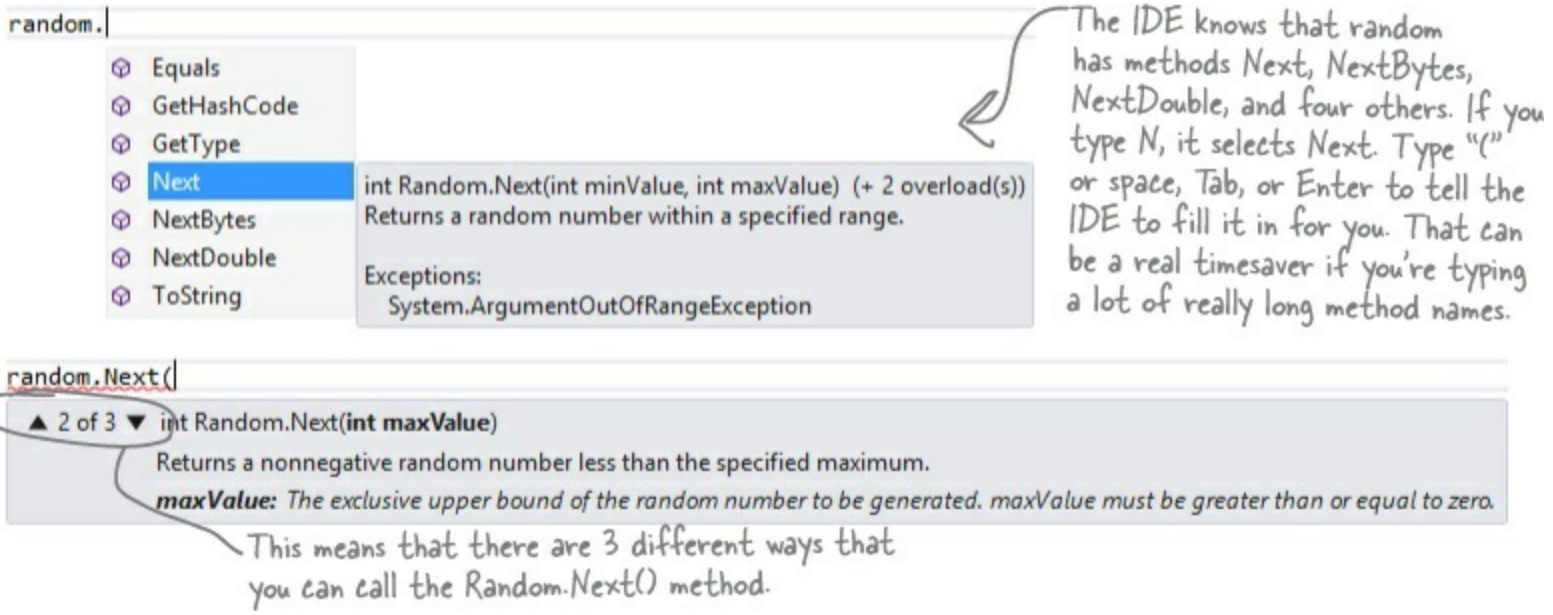
The Solution Explorer shows you the different files in the solution folder.

- **USE THE TABS TO SWITCH BETWEEN OPEN FILES**
  Since your program is split up into more than one file, you'll usually have several code files open at once. When you do, each one will be in its own tab in the code editor. The IDE displays an asterisk (*) next to a filename if it hasn't been saved yet.



When you're working on a program, you'll often have two tabs for it at the same time—one for the designer, and one to view the code. Use **Control-Tab** to switch between open windows quickly.

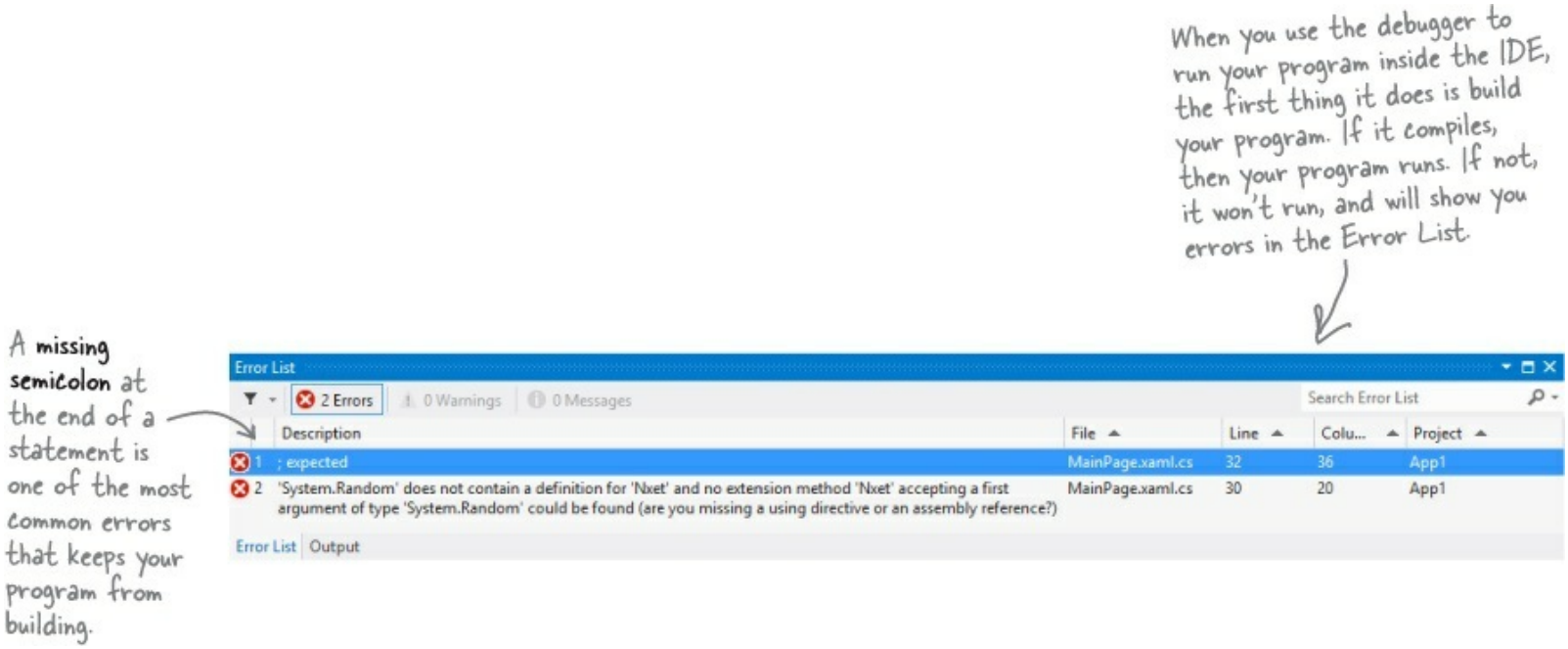
- **THE IDE HELPS YOU WRITE CODE**
  Did you notice little windows popping up as you typed code into the IDE? That's a feature called IntelliSense, and it's really useful. One thing it does is show you possible ways to complete your current line of code. If you type `random` and then a period, it knows that there are three valid ways to complete that line:



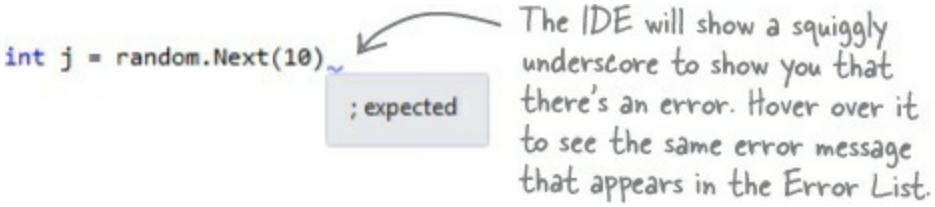
If you select Next and type (, the IDE's IntelliSense will show you information about how you can complete the line.

- **THE ERROR LIST HELPS YOU TROUBLESHOOT COMPILER ERRORS**

  If you haven't already discovered how easy it is to make typos in a C# program, you'll find out very soon! Luckily, the IDE gives you a great tool for troubleshooting them. When you build your solution, any problems that keep it from compiling will show up in the Error List window at the bottom of the IDE:

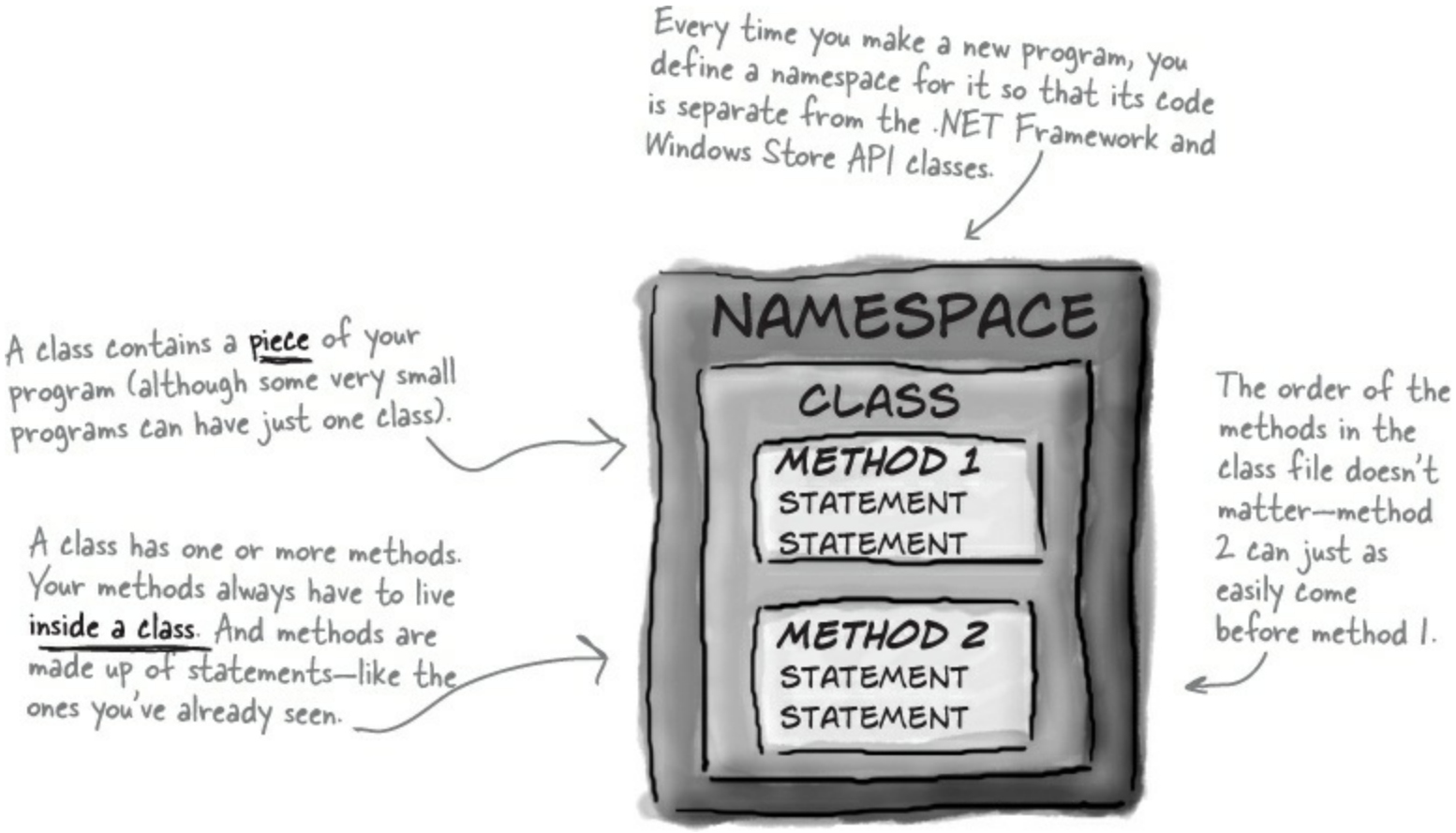


Double-click on an error, and the IDE will jump to the problem in the code:



Anatomy of a 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.



Let's take a closer look at your code

Open up the code from your Save the Humans project's *MainPage.xaml.cs* so we can have a closer look at it.

- **❶ THE CODE FILE STARTS BY USING THE .NET FRAMEWORK TOOLS**
  You'll find a set of `using` lines at the top of every program file. They tell C# which parts of the .NET Framework or Windows Store API to use. If you use other classes that are in other namespaces, then you'll add `using` lines for them, too. Since apps often use a lot of different tools from the .NET Framework and Windows Store API, the IDE automatically adds a bunch of `using` lines when it creates a page (which isn't quite as "blank" as it appeared) and adds it to your project.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
```

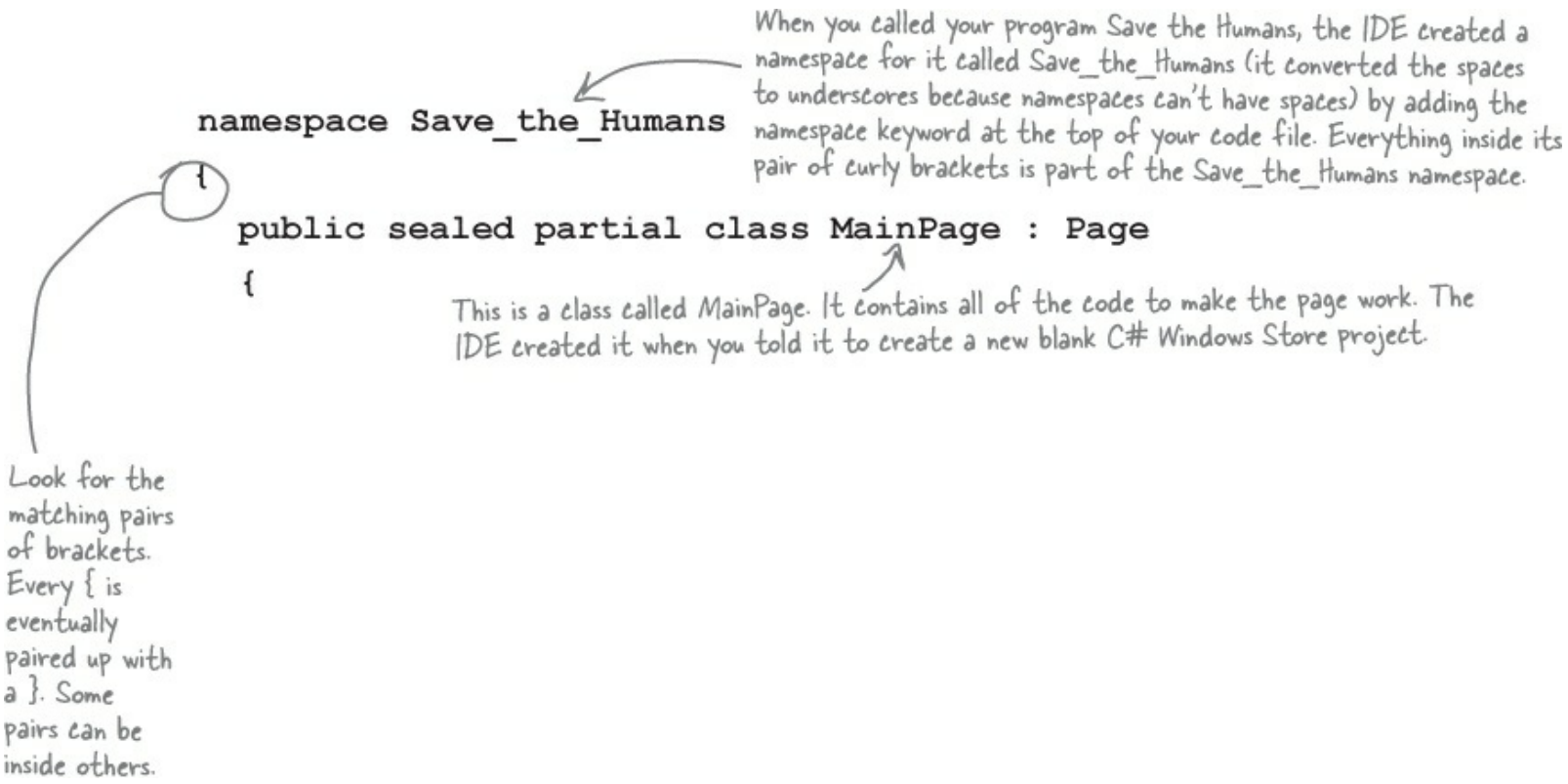
These **using** lines are at the top of every code file. They tell C# to use all of those .NET Framework classes. Each one tells your program that the classes in this particular .cs file will use all of the classes in one specific .NET Framework (System) or Windows Store API namespace.

  One thing to keep in mind: you don't actually *have* to use a `using` statement. You can always use the fully qualified name. Back in your Save the Humans app, you added this line:

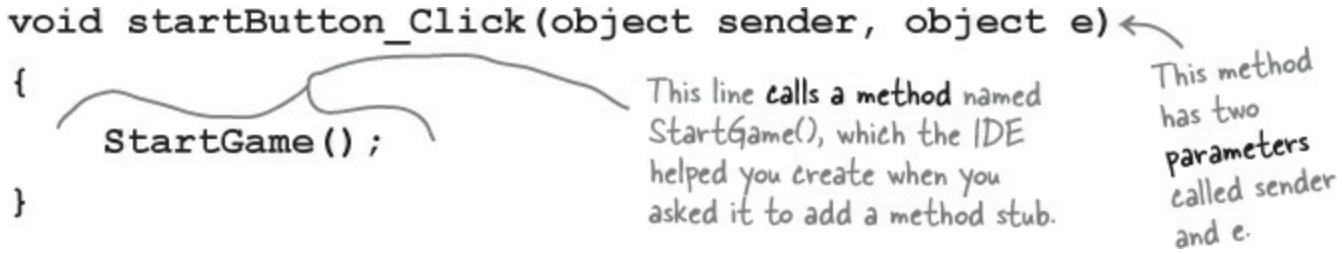
```
using Windows.UI.Xaml.Media.Animation;
```

  Try commenting out that line by adding `//` in front of it, then have a look at the errors that show up in the error list. You can make one of them go away. Find a `Storyboard` that the IDE now tells you has an error, and change it to `Windows.UI.Xaml.Media.Animation.Storyboard` (but you should undo the comment you added to make your program work again).

- **❷ C# PROGRAMS ARE ORGANIZED INTO CLASSES**
  Every C# program is organized into **classes**. A class can do anything, but most classes do one specific thing. When you created the new program, the IDE added a class called `MainPage` that displays the page.



- **❸ CLASSES CONTAIN METHODS THAT PERFORM ACTIONS**
  When a class needs to do something, it uses a **method**. A method takes input, performs some action, and sometimes produces an output. The way you pass input into a method is by using **parameters**. Methods can behave differently depending on what input they're given. Some methods produce output. When they do, it's called a **return value**. If you see the keyword `void` in front of a method, that means it doesn't return anything.



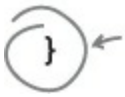
- **❹ A STATEMENT PERFORMS ONE SINGLE ACTION**
  When you filled in the `StartGame()` method, you added a bunch of **statements**. Every method is made up of statements. When your program calls a method, it executes the first statement in the method, then the next, then the next, etc. When the method runs out of statements or hits a `return` statement, it ends, and the program resumes after the statement that originally called the method.

```
private void StartGame()
{
    human.IsHitTestVisible = true;
    humanCaptured = false;
    progressBar.Value = 0;
    startButton.Visibility =
                Visibility.Collapsed;
    playArea.Children.Clear();
    playArea.Children.Add(target);
    playArea.Children.Add(human);
    enemyTimer.Start();
    targetTimer.Start();
}
}
```

This is the **method** called StartGame() that gets called when the user clicks the Start button.

The StartGame() method contains nine **statements**. Each statement ends with a semicolon.

It's OK to add extra line breaks to make your statements more readable. They're ignored when your program builds.



Here's the closing bracket at the very bottom of your MainPage.xaml.cs file.

THERE ARE NO DUMB QUESTIONS

Q: **Q: What's with all the curly brackets?**

A: **A:** C# uses curly brackets (or "braces") to group statements together into **blocks**. Curly 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 — just click on one, and you'll see it and its match get shaded darker.

Q: **Q: How come I get errors in the Error List window when I try to run my program? I thought that only happened when I did "Build Solution."**

A: **A:** Because the first thing that happens when you choose Start Debugging from the menu or press the toolbar button to start your program running is that it saves all the files in your solution and then tries to compile them. And when you compile your code — whether it's when you run it, or when you build the solution — if there are errors, the IDE will display them in the Error List instead of running your program.

NOTE

A lot of the errors that show up when you try to run your program also show up in the Error List window and as red squiggles under your code.

The IDE helps you build your code right.

A 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 a lot of helpful things like error highlighting, IntelliSense, WYSIWYG click-and-drag page 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.

WHAT'S MY PURPOSE?

Match each of these fragments of code generated by the IDE to what it does. (Some of these are new — take a guess and see if you got it right!)

	Set properties for a TextBlock control
	Nothing — it's a comment that the programmer added to explain the code to anyone who's reading it
	Disable the maximize icon ( ) in the title bar of the `Form1` window
	A special kind of comment that the IDE uses to explain what an entire block of code does
	Change the background color of a Grid control named `myGrid`
	A method that executes whenever a program displays its main page

WHAT'S MY PURPOSE?

Match each of these fragments of code generated by the IDE to what it does. (Some of these are new — take a guess and see if you got it right!)

```
myGrid.Background =
    new SolidColorBrush(Colors.Violet);
```

```
// This loop gets executed three times
```

```
public sealed partial class MainPage : Page
{
  private void InitializeComponent()
  {
  ⋮
  }
}
```

```
helloLabel.Text = "hi there";
helloLabel.FontSize = 24;
```

```
/// <summary>
/// Bring up the picture of Rover when
/// the button is clicked
/// </summary>
```

```
partial class Form1
{
  ⋮
  this.MaximizeBox = false;
  ⋮
}
```

Set properties for a TextBlock control

Nothing—it's a comment that the programmer added to explain the code to anyone who's reading it

Disable the maximize icon () in the title bar of the `Form1` window

Wait, a window? Not a page? You'll start learning about desktop apps with windows and forms later in this chapter.

A special kind of comment that the IDE uses to explain what an entire block of code does

Change the background color of a Grid control named `myGrid`

A method that executes whenever a program displays its main page

Two classes can be in the same namespace

Take a look at these two class files from a program called `PetFiler2`. They've got 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()`. It doesn't matter how the various namespaces and classes are divided up between files. They still act the same when they're run.

SomeClasses.cs

```
namespace PetFiler2 {

    class Dog {

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

    }

    partial class Cat {

        public void Meow() {
            // more statements
        }

    }
}
```

When a method is "public" it means every other class in the namespace can access its methods.

MoreClasses.cs

```
namespace PetFiler2 {

    class Fish {

        public void Swim() {
            // statements
        }

    }

    partial class Cat {

        public void Purr() {
            // statements
        }

    }
}
```

Since these classes are in the same namespace, they can all "see" each other—even though they're in different files. A class can span multiple files too, but you need to use the "partial" keyword when you declare it.

You can only split a class up into different files if you use the "partial" keyword. You probably won't do that in any of the code you write in this book, but the IDE used it to split your page up into two files so it could put the XAML code into MainPage.xaml and the C# code into MainPage.xaml.cs.

There's more to namespaces and class declarations, but you won't need them for the work you're doing right now. Flip to #3 in the "Leftovers" appendix to read more.

Your programs use variables to work with data

When you get right down to it, every program is basically a data cruncher. Sometimes the data is in the form of a document, or an image in a video game, or an instant message. But it's all just data. And that's where **variables** come in. A variable is what your program uses to store data.

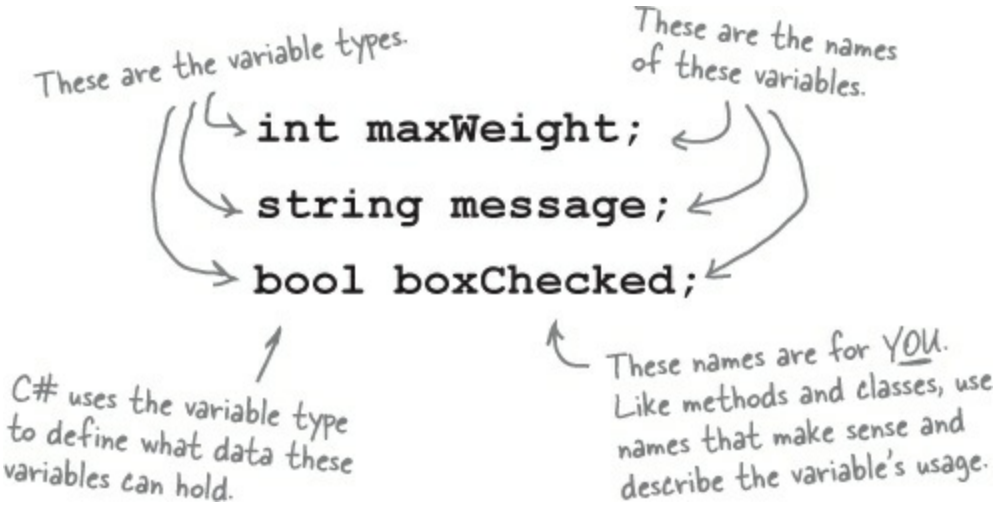
WATCH IT!

Are you already familiar with another language?

If so, you might find that a few things in this chapter seem really familiar. Still, it's worth taking the time to run through the exercises anyway, because there may be a few ways that C# is different from what you're used to.

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'll keep your program from compiling if you make a mistake and try to do something that doesn't make sense, like subtract "Fido" from 48353.



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 that you've written or will ever write. So if 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 — but the variable is still called `myHeight`.

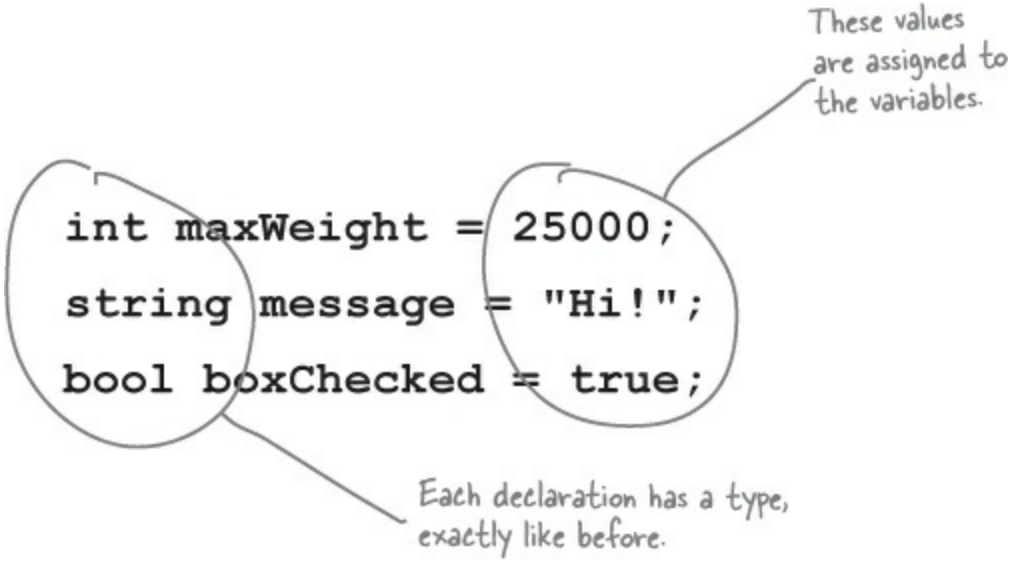
> **Whenever your program needs to work with numbers, text, true/false values, or any other kind of data, you'll use variables to keep track of them.**

You have to assign values to variables before you use them

Try putting these statements into a C# program:

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

Go ahead, give it a shot. You'll get an error, and the IDE will refuse to compile your code. That's because the compiler 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:



If you write code that uses a variable that hasn't been assigned a value, your code won't compile. 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.

NOTE

var-i-a-ble, noun.

an element or feature likely to change. *Predicting the weather would be a whole lot easier if meterologists didn't have to take so many **variables** into account.*

To programmers, the word "string" almost always means a string of text, and "int" is almost always short for integer.

C# uses familiar math symbols

Once you've got some data stored in a variable, what can you do with it? Well, if it's a number, you'll probably want to add, subtract, multiply, or divide it. And that's where **operators** come in. You already know the basic ones. Let's talk about a few more. Here's a block of code that uses operators to do some simple math:

```
int number = 15;
number = number + 10;
number = 36 * 15;
number = 12 - (42 / 7);
number += 10;
number *= 3;
number = 71 / 3;

int count = 0;
count ++;
count --;

string result = "hello";
result += " again " + result;
output.Text = result;
result = "the value is: " + count;
result = "";

bool yesNo = false;
bool anotherBool = true;
yesNo = !anotherBool;
```

We declared a new int variable called number and set it to 15. Then we added 10 to it. After the second statement, number is equal to 25.

The third statement changes the value of number, setting it equal to 36 times 15, which is 540. Then it resets it again, setting it equal to 12 – (42 / 7), which is 6.

This operator is a little different. += means take the value of number and add 10 to it. Since number is currently equal to 6, adding 10 to it sets its value to 16.

The *= operator is similar to +=, except it multiplies the current value of number by 3, so it ends up set to 48.

Normally, 71 divided by 3 is 23.666666.... But when you're dividing two ints, you'll always get an int result, so 23.666... gets truncated to 23.

You'll use int a lot for counting, and when you do, the ++ and -- operators come in handy. ++ increments count by adding one to the value, and -- decrements count by subtracting one from it, so it ends up equal to zero.

This sets the contents of a TextBlock control named output to "hello again hello".

When you use the + operator with a string, it just puts two strings together. It'll automatically convert numbers to strings for you.

The "" is an empty string. It has no characters. (It's kind of like a zero for adding strings.)

A bool stores true or false. The ! operator means NOT. It flips true to false, and vice versa.

RELAX

Don't worry about memorizing these operators now.

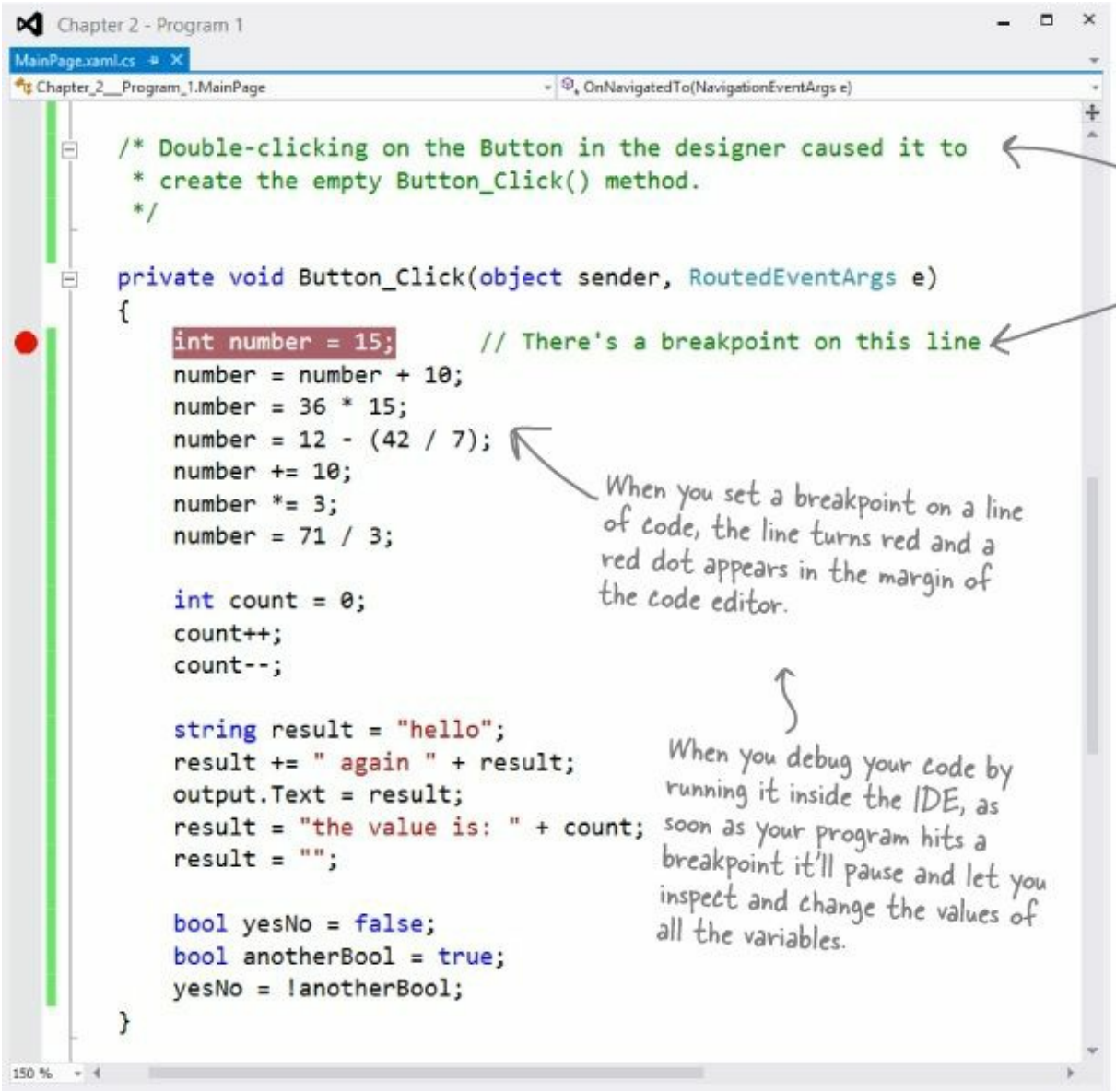
You'll get to know them because you'll see 'em over and over again.

Use the debugger to see your variables change

The debugger is a great tool for understanding how your programs work. You can use it to see the code on the previous page in action.

Debug this!

- **❶ CREATE A NEW VISUAL C# WINDOWS STORE BLANK APP (XAML) PROJECT.** Drag a TextBlock onto your page and give it the name `output`. Then add a Button and double-click it to add a method called `Button_Click()`. The IDE will automatically open that method in the code editor. Enter all of the code on the previous page into the method.
- **❷ INSERT A BREAKPOINT ON THE FIRST LINE OF CODE.** Right-click on the first line of code (`int number = 15;`) and choose Insert Breakpoint from the Breakpoint menu. (You can also click on it and choose Debug→Toggle Breakpoint or press F9.)



Comments (which either start with two or more slashes or are surrounded by /* and */ marks) show up in the IDE as green text. You don't have to worry about what you type in between those marks, because comments are always ignored by the compiler.

When you set a breakpoint on a line of code, the line turns red and a red dot appears in the margin of the code editor.

When you debug your code by running it inside the IDE, as soon as your program hits a breakpoint it'll pause and let you inspect and change the values of all the variables.

Creating a new Blank App project will tell the IDE to create a new project with a blank page. You might want to name it something like UseTheDebugger (to match the header of this page). You'll be building a whole lot of programs throughout the book, and you may want to go back to them later.

Flip the page and keep going!

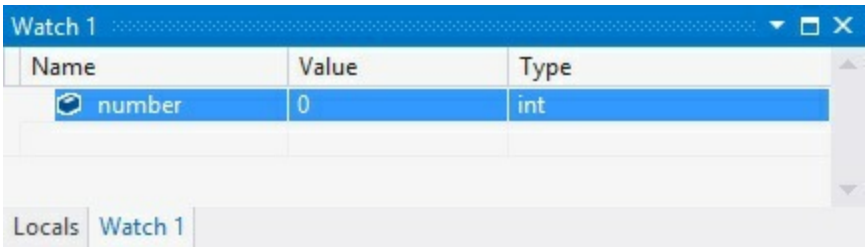
- **❸ START DEBUGGING YOUR PROGRAM.**
  Run your program in the debugger by clicking the Start Debugging button (or by pressing F5, or by choosing Debug→Start Debugging from the menu). Your program should start up as usual and display the page.
- **❹ CLICK ON THE BUTTON TO TRIGGER THE BREAKPOINT.**
  As soon as your program gets to the line of code that has the breakpoint, the IDE automatically brings up the code editor and highlights the current line of code in yellow.

```
int number = 15;
number = number + 10;
number = 36 * 15;
number = 12 - (42 / 7);
number += 10;
number *= 3;
number = 71 / 3;
```

IDE TIP:

When you're debugging a Windows Store app, you can return to the debugger by pressing the Windows logo key+D. If you're using a touch screen, swipe from the left edge of the screen to the right. Then you can pause or stop the debugger using the Debug toolbar or menu items.

- **❺ ADD A WATCH FOR THE `number` VARIABLE.**
  Right-click on the `number` variable (any occurrence of it will do!) and choose Add Watch from the menu. The Watch window should appear in the panel at the bottom of the IDE:



- **❻ STEP THROUGH THE CODE.**
  Press F10 to step through the code. (You can also choose Debug→Step Over from the menu, or click the Step Over button in the Debug toolbar.) The current line of code will be executed, setting the value of `number` to 15. The next line of code will then be highlighted in yellow, and the Watch window will be updated:

- **❼ CONTINUE RUNNING THE PROGRAM.**
  When you want to resume, just press F5 (or Debug→Continue), and the program will resume running as usual.

Adding a watch can help you keep track of the values of the variables in your program. This will really come in handy when your programs get more complex.

NOTE

You can also hover over a variable while you're debugging to see its value displayed in a tooltip...and you can pin it so it stays open!

Loops perform an action over and over

Here's a peculiar thing about most large programs: 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`!).

NOTE

That's a big part of why Booleans are so important. A loop uses a test to figure out if it should keep looping.

```
while (x > 5)
{
    x = x - 3;
}
```

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

Every for loop has three statements. The first 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
}
```

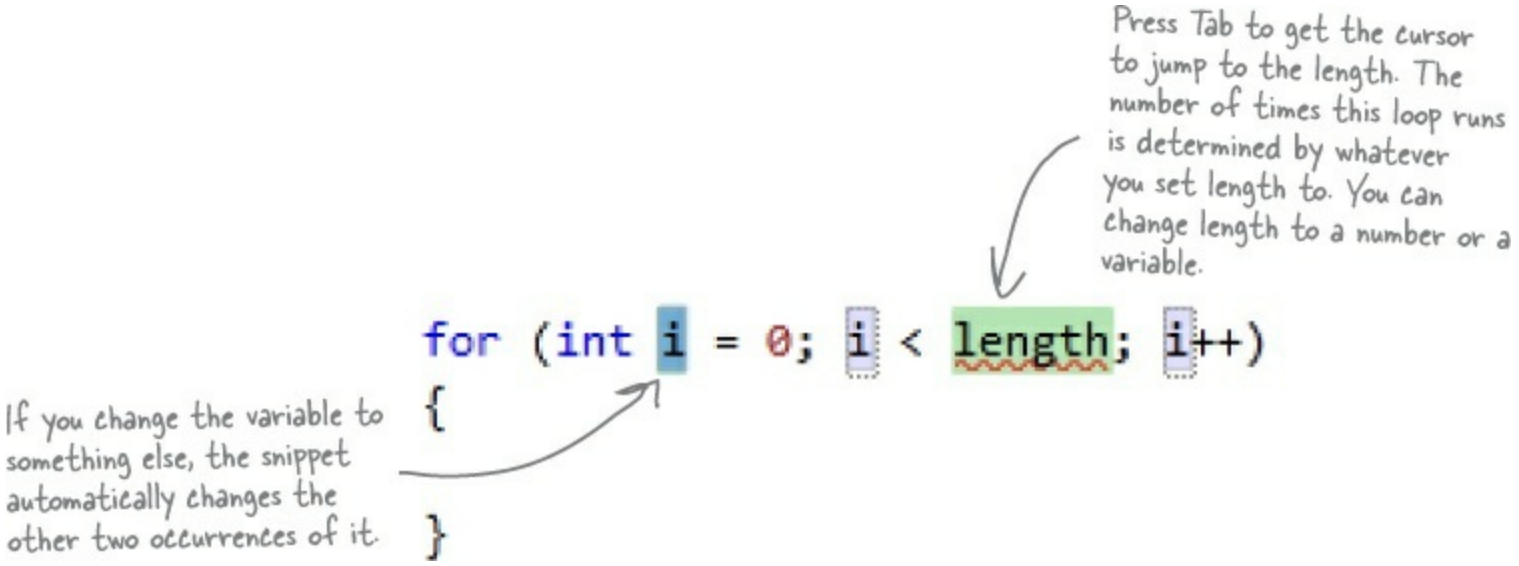
IDE TIP: BRACKETS

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:

```
bool test = true;
while (test == true)
{
    // Contents of the loop
}
```

Use a code snippet to write simple for loops

You'll be typing for loops in just a minute, and the IDE can help speed up your coding a little. Type `for` followed by two tabs, and the IDE will automatically insert code for you. If you type a new variable, it'll automatically update the rest of the snippet. Press Tab again, and the cursor will jump to the length.



if/else statements make decisions

Use **if/else statements** to tell your program to do certain things only when the **conditions** you set up are (or aren't) true. A lot of if/else 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.

```
string message = "";

if (someValue == 24)
{
    message = "The value was 24.";
}
```

Every if statement starts with a conditional test.

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

Always use two equals signs to check if two things are equal to each other.

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

if/else statements are pretty straightforward. If the conditional test is true, the program executes the statements between the first set of brackets. Otherwise, it executes the statements between the second set.

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.

NOTE

Make sure you choose a sensible name for this project, because you'll refer back to it later in the book.

Build an app from the ground up

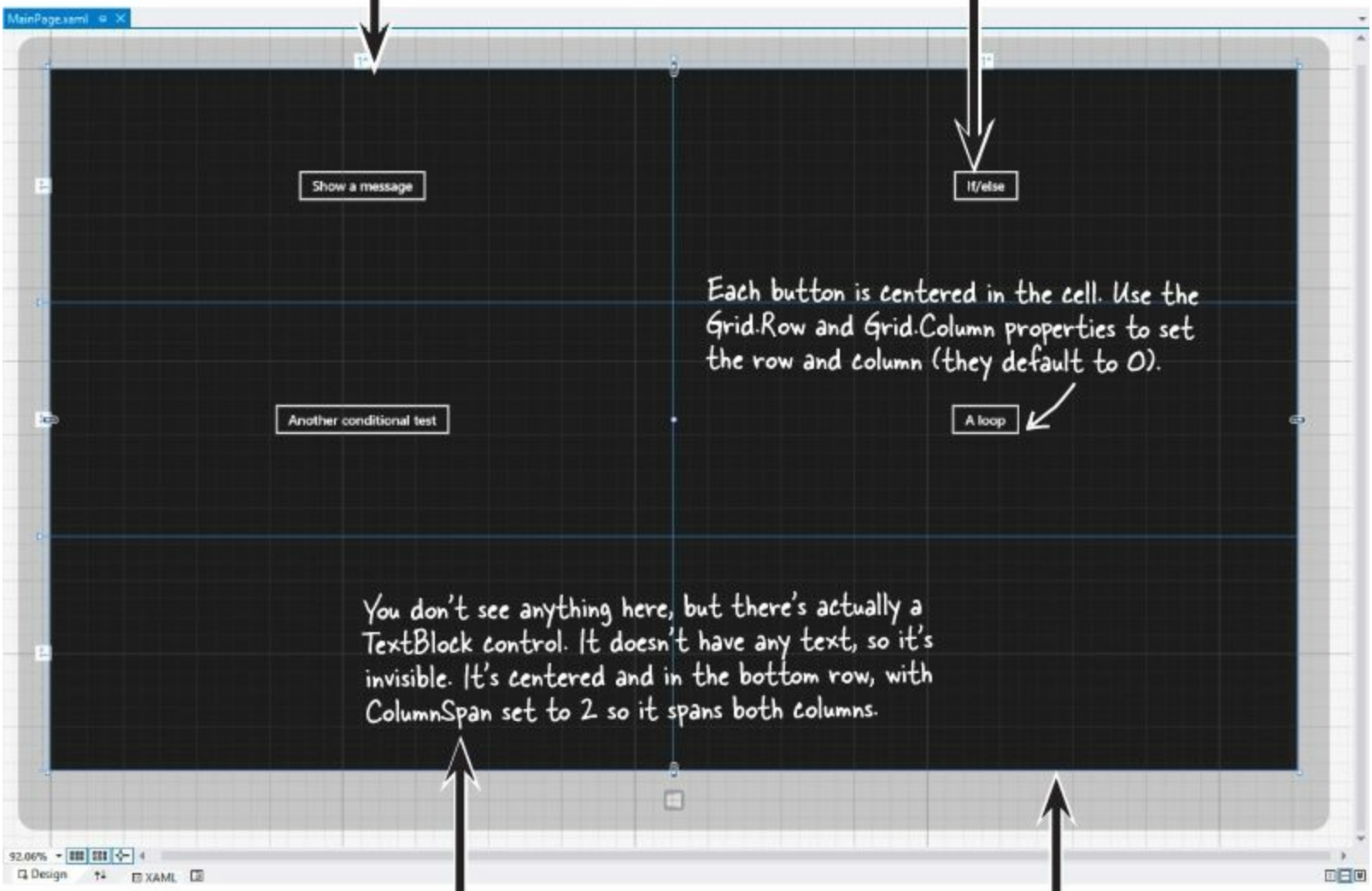
When you see these sneakers, it means that it's time for you to come up with code on your own.

The real work of any program is in its statements. You've already seen how statements fit into a page. Now let's really dig into a program so you can understand every line of code. Start by **creating a new Visual C# Windows Store Blank App project**. This time, don't delete the *MainPage.xaml* file created by the Blank App template. Instead, use the IDE to modify it by adding three rows and two columns to the grid, then adding four Button controls and a TextBlock to the cells.

Build this page

The page has a grid with three rows and two columns. Each row definition has its height set to `1*`, which gives it a `<RowDefinition/>` without any properties. The column heights work the same way.

The page has four Button controls, one in each row. Use the `Content` property to set their text to Show a message, If/else, Another conditional test, and A loop.



Each button is centered in the cell. Use the Grid.Row and Grid.Column properties to set the row and column (they default to 0).

You don't see anything here, but there's actually a TextBlock control. It doesn't have any text, so it's invisible. It's centered and in the bottom row, with ColumnSpan set to 2 so it spans both columns.

The bottom cell has a TextBlock control named `myLabel`. Use its `Style` property to set the style to `BodyTextBlockStyle`.

If you need to use the Edit Style right-mouse menu to set this but you're having trouble selecting the control, you can right-click on the TextBlock control in the Document Outline and choose Edit Style from there.

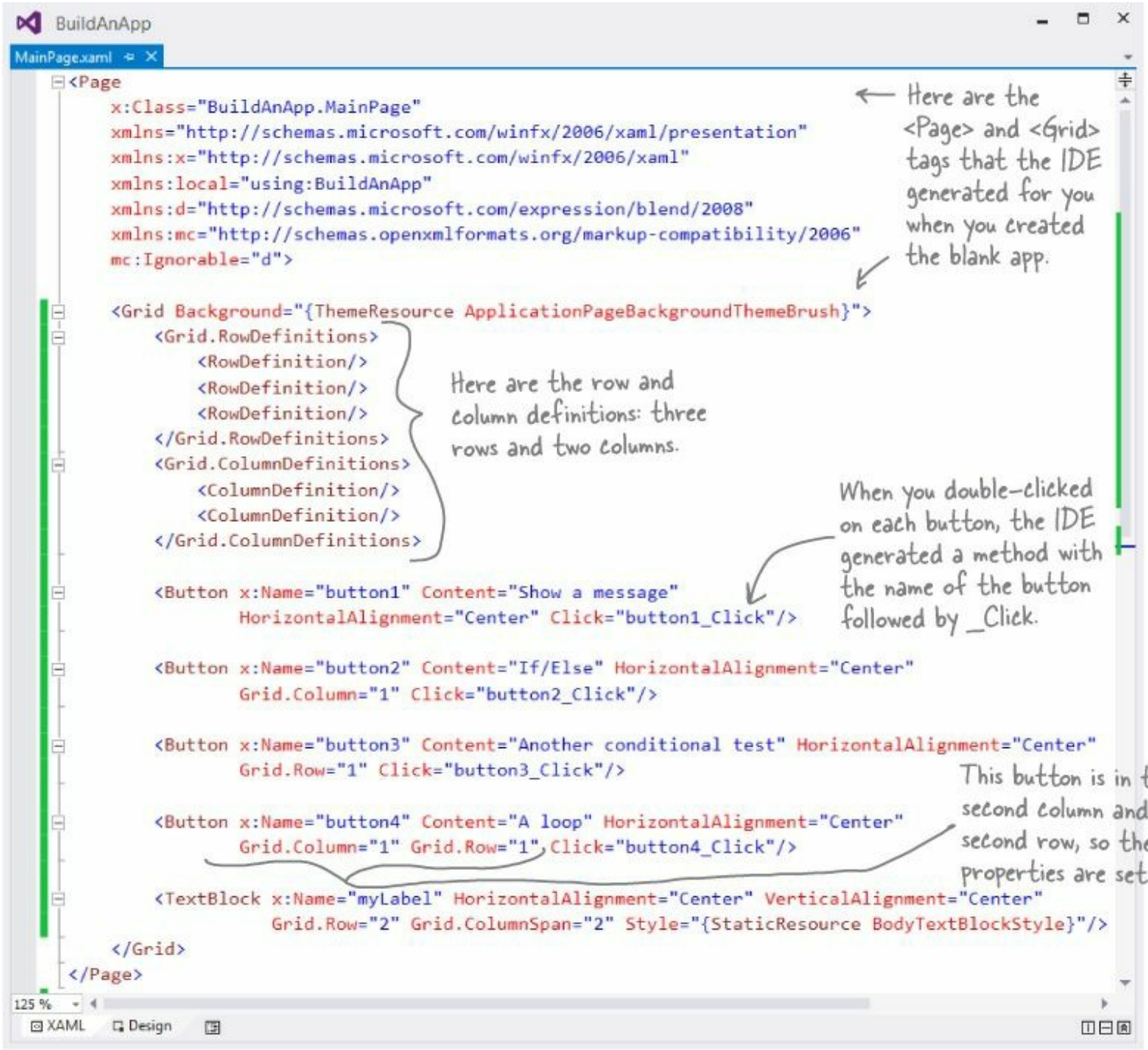
Use the `x:Name` property to name the buttons `button1`, `button2`, `button3`, and `button4`. Once they're named, double-click on each of them to add an event handler method.

EXERCISE SOLUTION

NOTE

Here's our solution to the exercise. Does your solution look similar? Are the line breaks different, or the properties in a different order? If so, that's OK!

A lot of programmers don't use the IDE to create their XAML — they build it by hand. If we asked you to type in the XAML by hand instead of using the IDE, would you be able to do it?




```
<Page
    x:Class="BuildAnApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BuildAnApp"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

        <Button x:Name="button1" Content="Show a message"
                HorizontalAlignment="Center" Click="button1_Click"/>

        <Button x:Name="button2" Content="If/Else" HorizontalAlignment="Center"
                Grid.Column="1" Click="button2_Click"/>

        <Button x:Name="button3" Content="Another conditional test" HorizontalAlignment="Center"
                Grid.Row="1" Click="button3_Click"/>

        <Button x:Name="button4" Content="A loop" HorizontalAlignment="Center"
                Grid.Column="1" Grid.Row="1" Click="button4_Click"/>

        <TextBlock x:Name="myLabel" HorizontalAlignment="Center" VerticalAlignment="Center"
                   Grid.Row="2" Grid.ColumnSpan="2" Style="{StaticResource BodyTextBlockStyle}"/>
    </Grid>
</Page>
```

BRAIN POWER

Why do you think the left column and top row are given the number 0, not 1? Why is it OK to leave out the `Grid.Row` and `Grid.Column` properties for the top-left cell?

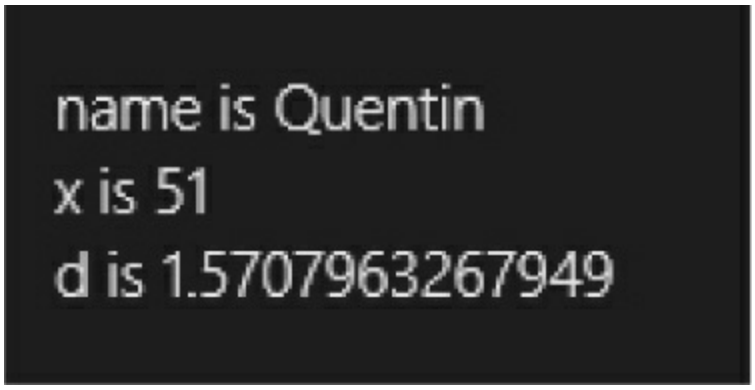
Make each button do something

Here's how your program is going to work. Each time you press one of the buttons, it will update the TextBlock at the bottom (which you named `myLabel`) with a different message. The way you'll do it is by adding code to each of the four event handler methods that you had the IDE generate for you. Let's get started!

DO THIS!

When you see a "Do this!", pop open the IDE 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.

- **❶ MAKE BUTTON1 UPDATE THE LABEL.**
  Go to the code for the `button1_Click()` method and fill in the code below. This is your chance to really understand what every statement does, and why the program will show this output:



Here's the code for the button:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    // this is a comment
    string name = "Quentin";
    int x = 3;
    x = x * 17;
    double d = Math.PI / 2;
    myLabel.Text = "name is " + name
        + "\nx is " + x
        + "\nd is " + d;
}
```

x is a variable. The "int" part tells C# that it's an integer, and the rest of the statement sets its value to 3.

This line creates the output of the program: the updated text in the TextBlock named myLabel.

There's a built-in class called Math, and it's got a member called PI. Math lives in the System namespace, so the file this code came from needs to have a using System; line at the top.

Luckily, the IDE generated the using line for you.

The \n is an escape sequence to add a line break to the TextBlock text.

A FEW HELPFUL TIPS

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

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

- You can add comments to your code by starting them with two slashes:

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

- Variables are declared with a **name** and a **type** (there are plenty of types that you'll learn about in Chapter 4):

```
int weight;
// weight is an integer
```

- The code for a class or a method goes between curly braces:

```
public void Go() {
   // your code here
}
```

- Most of the time, extra whitespace is fine:

```
int j    =     1234  ;
```

is the same as:

```
int j = 1234;
```

Run your program and make sure the output matches the screenshot on this page.

Flip the page to finish your program!

Set up conditions and see if they’re true

Use **if/else statements** to tell your program to do certain things only when the **conditions** you set up are (or aren’t) true.

Use logical operators to check conditions

You've just looked at the == operator, which you use to test whether two variables are equal. There are a few other operators, too. Don't worry about memorizing them right now — you'll get to know them over the next few chapters.

- The `!=` operator works a lot like ==, except it's true if the two things you're comparing are **not equal**.
- You can use > and < to compare numbers and see if one is bigger or smaller than the other.
- The ==, `!=`, >, and < operators are called **conditional operators**. When you use them to test two variables or values, it's called performing a **conditional test.**
- You can combine individual conditional tests into one long test using the `&&` operator for AND and the `||` operator for OR. So to check if `i` equals 3 or `j` is less than 5, do `(i == 3) || (j < 5)`.

When you use a conditional operator to compare two numbers, it's called a conditional test.

- **❷ SET A VARIABLE AND THEN CHECK ITS VALUE.**
  Here's the code for the second button. It's an if/else statement that checks an integer **variable** called `x` to see if it's equal to 10.

> **NOTE**
>
> Make sure you stop your program before you do this — the IDE won't let you edit the code while the program's running. You can stop it by closing the window, using the stop button on the toolbar, or selecting Stop Debugging from the Debug menu.

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    int x = 5;
    if (x == 10)
    {
        myLabel.Text = "x must be 10";
    }
    else
    {
        myLabel.Text = "x isn't 10";
    }
}
```

First we set up a variable called x and make it equal to 5. Then we check if it's equal to 10.

Here's the output. See if you can tweak one line of code and get it to say "x must be 10" instead.

- **❸ ADD ANOTHER CONDITIONAL TEST.**
  The third button makes this output. Then change it so `someValue` is set to 3 instead of 4. The code inside the if block doesn't get run — can ypou figure out why? Put a breakpoint on the first statement and step through the method, using Alt-Tab to switch to the app and back to make sure the TextBlock gets updated.

this line runs no matter what

This line checks someValue to see if it's equal to 3, and then it checks to make sure name is "Joe".

```
private void button3_Click(object sender, RoutedEventArgs e)
{
    int someValue = 4;
    string name = "Bobbo Jr.";
    if ((someValue == 3) && (name == "Joe"))
    {
        myLabel.Text = "x is 3 and the name is Joe";
    }
    myLabel.Text = "this line runs no matter what";
}
```

- **❹ ADD LOOPS TO YOUR PROGRAM.**
  Here's the code for the last button. It's got two loops. The first is a **`while`** loop, which repeats the statements inside the brackets as long as the condition is true — do something *while* this is true. The second one is a **`for`** loop. Take a look and see how it works.

```
private void button4_Click(object sender, RoutedEventArgs e)
{
    int count = 0;

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

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

    myLabel.Text = "The answer is " + count;
}
```

This loop keeps repeating as long as the count variable is less than 10.

The second part of the for statement is the test. It says "for as long as i is less than five, the loop should keep on going." The test is run before the code block, and the block is executed only if the test is true.

This sets up the loop. It just assigns a value to the integer that'll be used in it.

This statement gets executed at the end of each loop. In this case, it adds one to i every time the loop executes. This is called the iterator, and it's run **immediately after** all the statements in the code block.

Before you click on the button, read through the code and try to figure out what the TextBlock will show. Then click the button and see if you were right!

SHARPEN YOUR PENCIL

Let's get a little more practice with conditional tests and loops. Take a look at the code below. Circle the conditional tests, and fill in the blanks so that the comments correctly describe the code that's being run.

```
int result = 0; // this variable will hold the final result
int x = 6; // declare a variable x and set it to 6
while (x > 3) {
  // execute these statements as long as .....
  result = result + x; // add x .....
  x = x - 1; // subtract .....
}
for (int z = 1; z < 3; z = z + 1) {
  // start the loop by .....
  // keep looping as long as .....
  // after each loop, .....
  result = result + z; // .....
}
// The next statement will update a TextBlock with text that says
// .....
myLabel.Text = "The result is " + result;
```

We filled in the first one for you.

MORE ABOUT CONDITIONAL TESTS

You can do simple conditional 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.

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 actually times when you'll want to use one in your program.

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?

LOOP #1

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

For Loop #3, how many times will this statement be executed?

LOOP #2

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

LOOP #3

```
int j = 2;
for (int i = 1; i < 100;
     i = i * 2)
{
    j = j - 1;
    while (j < 25)
    {
        j = j + 5;
    }
}
```

For Loop #5, how many times will this statement be executed?

LOOP #4

```
while (true) { int i = 1;}
```

LOOP #5

```
int p = 2;
for (int q = 2; q < 32;
     q = q * 2)
{
    while (p < q)
    {
        p = p * 2;
    }
    q = p - q;
}
```

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

NOTE

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

BRAIN POWER

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

SHARPEN YOUR PENCIL SOLUTION

Let's get a little more practice with conditional tests and loops. Take a look at the code below. Circle the conditional tests, and fill in the blanks so that the comments correctly describe the code that's being run.

```
int result = 0; // this variable will hold the final result

int x = 6; // declare a variable x and set it to 6

while (x > 3) {

// execute these statements as long as x is greater than 3

result = result + x; // add x to the result variable


x = x - 1; // subtract 1 from the value of x

}

for (int z = 1; z < 3; z = z + 1) {

// start the loop by declaring a variable z and setting it to 1

// keep looping as long as z is less than 3

// after each loop, add 1 to z

result = result + z; // add the value of z to result

}

// The next statement will update a TextBlock with text that says

// The result is 18

myLabel.Text = "The result is " + result;
```

This loop runs twice—first with z set to 1, and then a second time with z set to 2. Once it hits 3, it's no longer less than 3, so the loop stops.

SHARPEN YOUR PENCIL SOLUTION

Here are a few loops. Write down if each loop will repeat forever or eventually end. If it's going to end, how many times will it loop?

LOOP #1
This loop executes once

LOOP #3
This loop executes 7 times

LOOP #5
This loop executes 8 times

LOOP #2
This loop runs forever

LOOP #4
Another infinite loop

Take the time to really figure this one out. Here's a perfect opportunity to try out the debugger on your own! Set a breakpoint on the statement `q = p - q;`. Add watches for the variables `p` and `q` and step through the loop.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Is every statement always in a class?**

A: **A:** Yes. Any time a C# program does something, it's because statements were executed. Those statements are a part of classes, and those classes are a part of namespaces. Even when it looks like something is not a statement in a class — like when you use the designer to set a property on a control on your page — if you search through your code you'll find that the IDE added or changed statements inside a class somewhere.

Q: **Q: Are there any namespaces I'm not allowed to use? Are there any I *have* to use?**

A: **A:** Yes, there are a few namespaces that will technically work, but which you should avoid. Notice how all of the `using` lines at the top of your C# class files always said `System`? That's because there's a `System` namespace that's used by the Windows Store API and the .NET Framework. It's where you find all of your important tools to add power to your programs, like `System.Linq`, which lets you manipulate sequences of data, and `System.IO`, which lets you work with files and data streams. But for the most part, you can choose any name you want for a namespace (as long as it only has letters, numbers, and underscores). When you create a new program, the IDE will automatically choose a namespace for you based on the program's name.

Q: **Q: I still don't get why I need this partial class stuff.**

A: **A:** Partial classes are how you can spread the code for one class between more than one file. The IDE does that when it creates a page — it keeps the code you edit in one file (like *MainPage. xaml*), and the code it modifies automatically for you in another file (*MainPage.xaml.cs*). You don't need to do that with a namespace, though. One namespace can span two, three, or a dozen or more files. Just put the namespace declaration at the top of the file, and everything within the curly brackets after the declaration is inside the same namespace. One more thing: you can have more than one class in a file. And you can have more than one namespace in a file. You'll learn a lot more about classes in the next few chapters.

Q: **Q: Let's say I drag something onto my page, so the IDE generates a bunch of code automatically. What happens to that code if I click Undo?**

A: **A:** The best way to answer this question is to try it! Give it a shot — do something where the IDE generates some code for you. Drag a button on a page, change properties. Then try to undo it. What happens? For most simple things, you'll see that the IDE is smart enough to undo it itself. (For some more complex things, like working with databases, you might be given a warning message that you're about to make a change that the IDE can't undo. You won't see any of those in this book.)

Q: **Q: So exactly how careful do I have to be with the code that's automatically generated by the IDE?**

A: **A:** You should generally be pretty careful. It's really useful to know what the IDE is doing to your code, and once in a while you'll need to know what's in there in order to solve a serious problem. But in almost all cases, you'll be able to do everything you need to do through the IDE.

BULLET POINTS

- You tell your program to perform actions using statements. Statements are always part of classes, and every class is in a namespace.
- Every statement ends with a semicolon (`;`).
- When you use the visual tools in the Visual Studio IDE, it automatically adds or changes code in your program.
- Code blocks are surrounded by curly braces `{ }`. Classes, `while` loops, if/else statements, and lots of other kinds of statements use those blocks.
- A conditional test is either `true` or `false`. You use conditional tests to determine when a loop ends, and which block of code to execute in an if/else statement.
- Any time your program needs to store some data, you use a variable. Use `=` to assign a variable, and `==` to test if two variables are equal.
- A `while` loop runs everything within its block (defined by curly braces) as long as the *conditional test* is `true`.
- If the conditional test is `false`, the `while` loop code block won't run, and execution will move down to the code immediately after the loop block.

CODE MAGNETS

Part of a C# program is all scrambled up on the fridge. Can you rearrange the code snippets to make a working C# program that produces the output? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need! (Hint: you'll definitely need to add a couple. Just write them in!)

The "" is an empty string—it means the variable result has no characters in it yet.

This magnet didn't fall off the fridge...

```
string result = "";
```

```
output.Text = result;
```

```
if (x == 1) {
  result = result + "d";
  x = x - 1;
}
```

```
if (x == 2) {
  result = result + "b c";
}
```

```
if (x > 2) {
  result = result + "a";
}
```

```
int x = 3;
```

```
x = x - 1;
result = result + "-";
```

```
while (x > 0)
```

Output:

This is a TextBlock named "output" that the program updates by setting its Text property.

Answers in Code Magnets Solution.

NOTE

We'll give you a lot of exercises like this throughout the book. We'll give you the answer in a couple of pages. If you get stuck, don't be afraid to peek at the answer — it's not cheating!

You'll be creating a lot of applications throughout this book, and you'll need to give each one a different name. We recommend naming this one "PracticeUsingIfElse". It helps to put programs from a chapter in the same folder.

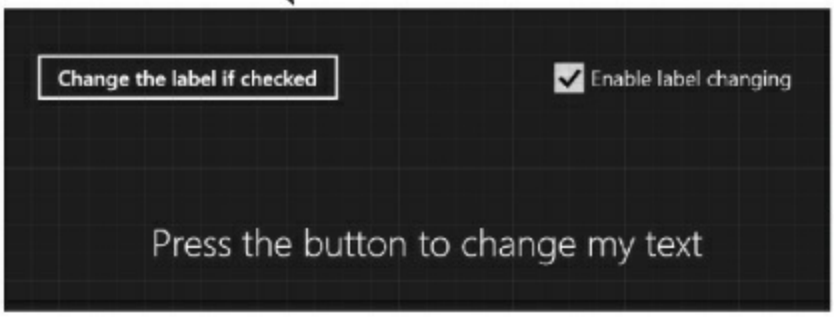
EXERCISE

Time to get some practice using if/else statements. Can you build this program?

Build this page.
It's got a grid with two rows and two columns. Just use the default page (don't delete *MainPage.xaml* and add a basic page).

If you create two rows and set one row's height to `1*` in the IDE, it seems to disappear because it's collapsed to a tiny size. Just set the other row to `1*` and it'll show up again.

Add a Button and a CheckBox.
You can find the CheckBox control in the toolbox, just below the Button control. Set the Button's name to `changeText` and the CheckBox's name to `enableCheckbox`. Use the Edit Text right-click menu option to set the text for both controls (hit Escape to finish editing the text). Right-click on each control and chose Reset Layout→All, then make sure both of them have their VerticalAlignment and HorizontalAlignment set to `Center`.



Add a TextBlock.
It's almost identical to the one you added to the bottom of the page in the last project. This time, name it `labelToChange` and set its `Grid.Row` property to `"1"`.

Set the TextBlock to this message if the user clicks the button but the box IS NOT checked.

Here's the conditional test to see if the checkbox is checked:

```
enableCheckbox.IsChecked == true
```

If that test is **NOT** true, then your program should execute two statements:

Hint: you'll put this code in the else block.

```
labelToChange.Text = "Text changing is disabled";
labelToChange.HorizontalAlignment = HorizontalAlignment.Center;
```

If the user clicks the button and the box IS checked, change the TextBlock so it either shows Left on the lefthand side or Right on the righthand side.

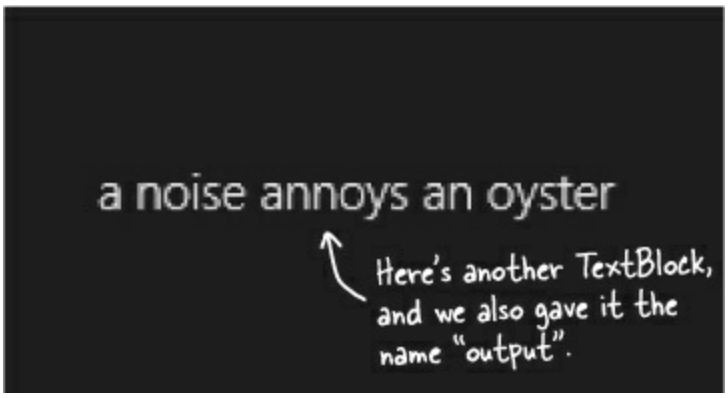
If the label's `Text` property is currently equal to `"Right"` then the program should change the text to `"Left"` and set its `HorizontalAlignment` property to `HorizontalAlignment.Left`. Otherwise, set its text to `"Right"` and its `HorizontalAlignment` property to `HorizontalAlignment.Right`. This should cause the program to flip the label back and forth when the user presses the button — but only if the checkbox is checked.

POOL PUZZLE

Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make a class that will compile and run. Don't be fooled — this one's harder than it looks.

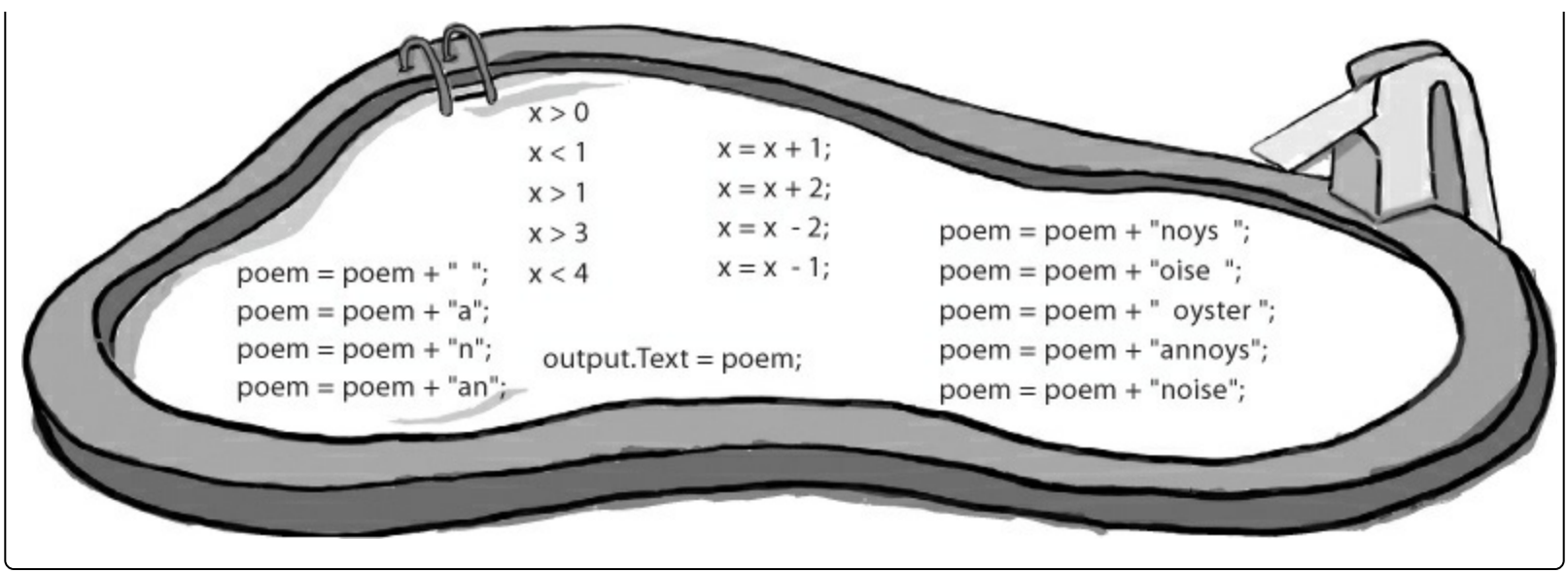
> **NOTE**
>
> We included these Pool Puzzle exercises throughout the book to give your brain an extra-tough workout. If you're the kind of person who loves twisty little logic puzzles, then you'll love this one. If you're not, give it a shot anyway — but don't be afraid to look at the answer to figure out what's going on. And if you're stumped by a pool puzzle, definitely move on.

Output




```
int x = 0;
string poem = "";

while ( __________ ) {

  _____________________________
  if ( x < 1 ) {
    _____________________________
  }
  _____________________________
  if ( __________ ) {
    _____________________________
    ____________
  }
  if ( x == 1 ) {

    _____________________________
  }
  if ( ____________ ) {
    _____________________________
  }

  _____________
}
____________________
```

Note: each snippet from the pool can only be used once!


x > 0
x < 1
x > 1
x > 3
x < 4
x = x + 1;
x = x + 2;
x = x - 2;
x = x - 1;
poem = poem + " ";
poem = poem + "a";
poem = poem + "n";
poem = poem + "an";
output.Text = poem;
poem = poem + "noys ";
poem = poem + "oise ";
poem = poem + " oyster ";
poem = poem + "annoys";
poem = poem + "noise";

EXERCISE SOLUTION

Time to get some practice using if/else statements. Can you build this program?

Here's the XAML code for the grid:

> **NOTE**
>
> We added line breaks as usual to make it easier to read on the page.

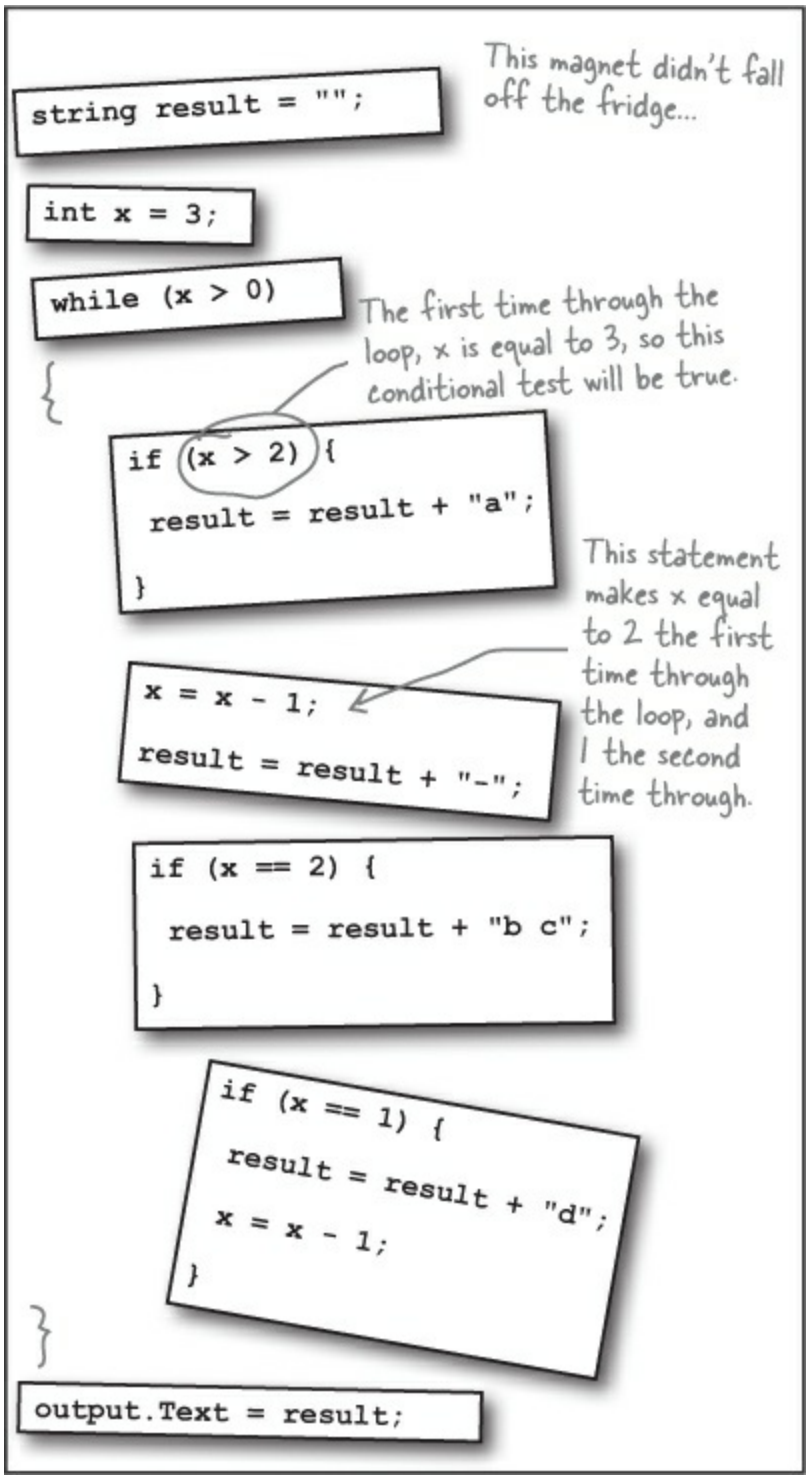
```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Button x:Name="changeText" Content="Change the label if checked"
            HorizontalAlignment="Center" Click="changeText_Click"/>

    <CheckBox x:Name="enableCheckbox" Content="Enable label changing"
              HorizontalAlignment="Center" IsChecked="true" Grid.Column="1"/>

    <TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
               Text="Press the button to set my text"
               HorizontalAlignment="Center" VerticalAlignment="Center"
               Grid.ColumnSpan="2"/>
</Grid>
```

If you double-clicked the button in the designer before you set its name, it may have created a Click event handler method called Button_Click_1() instead of changeText_Click().

And here's the C# code for the button's event handler method:

```
private void changeText_Click(object sender, RoutedEventArgs e)
{
    if (enableCheckbox.IsChecked == true)
    {
        if (labelToChange.Text == "Right")
        {
            labelToChange.Text = "Left";
            labelToChange.HorizontalAlignment = HorizontalAlignment.Left;
        }
        else
        {
            labelToChange.Text = "Right";
            labelToChange.HorizontalAlignment = HorizontalAlignment.Right;
        }
    }
    else
    {
    labelToChange.Text = "Text changing is disabled";
    labelToChange.HorizontalAlignment = HorizontalAlignment.Center;
    }
}
```

CODE MAGNETS SOLUTION



POOL PUZZLE SOLUTION

```
int x = 0;
string poem = "";

while ( x < 4 ) {

  poem = poem + "a";
  if ( x < 1 ) {
    poem = poem + " ";
  }
  poem = poem + "n";

  if ( x > 1 ) {

    poem = poem + " oyster";

    x = x + 2;
  }
  if ( x == 1 ) {

    poem = poem + "noys ";
  }
  if ( x < 1 ) {

    poem = poem + "oise ";
  }

  x = x + 1;
}
output.Text = poem;
```

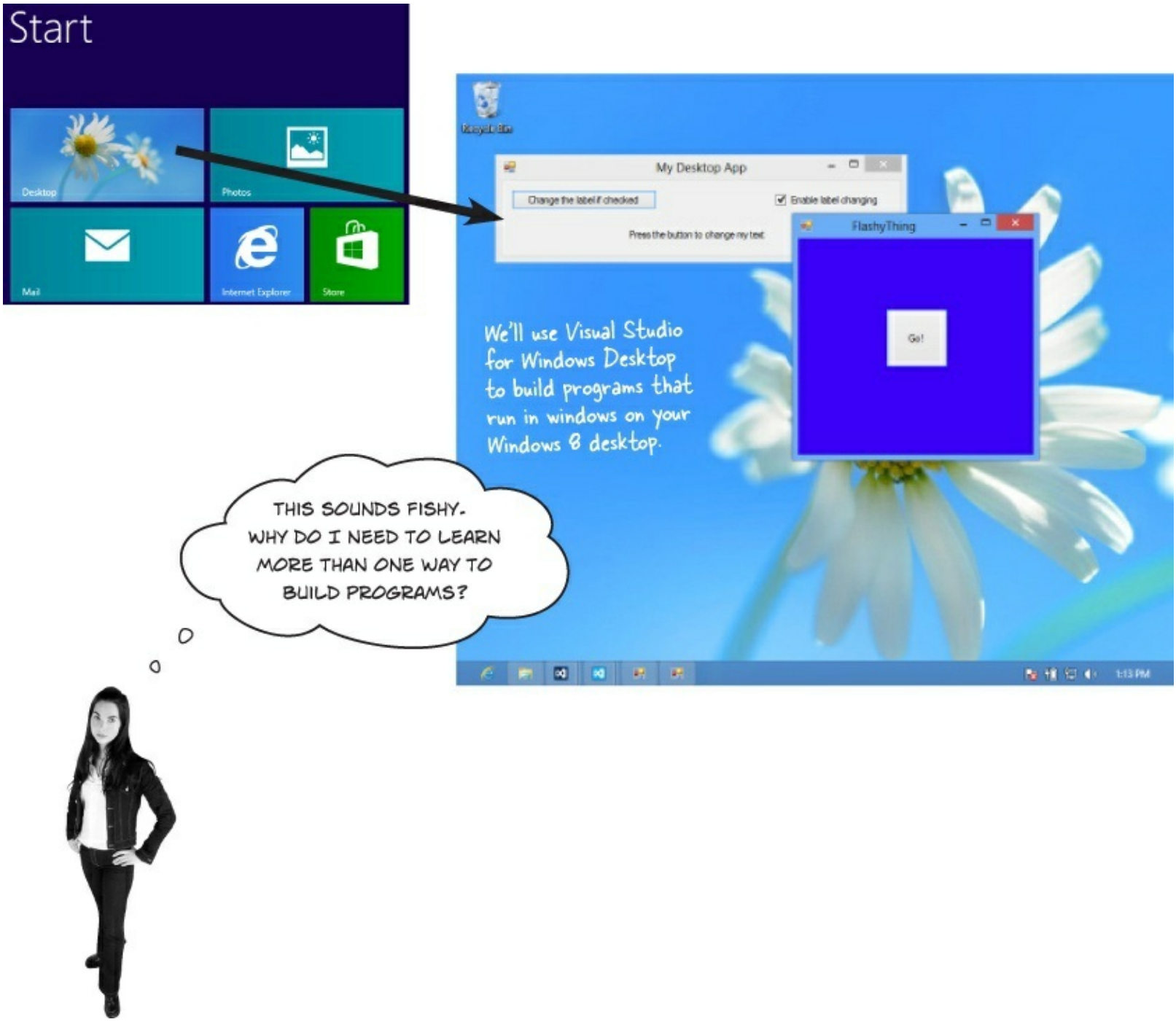
Did you get a different solution? Type it into the IDE and see if it works! There's more than one correct solution to the pool puzzle.

NOTE

If you want a real challenge, see if you can figure out what that other solution is! Here's a hint: there's another solution that keeps the word fragments in order. If you came up with that solution instead of the one on this page, see if you can figure out why this one works too.

Windows Desktop apps are easy to build

Windows 8 brought Windows Store apps, and that gave everyone a totally new way to use software on Windows. But that's not the only kind of program that you can create with Visual Studio. You can use Visual Studio for Windows Desktop to build **Windows Desktop applications** that run in windows on your Windows 8 desktop.



Windows Desktop apps are an effective learning tool

We'll spend the next several chapters building programs using Visual Studio for Windows Desktop before coming back to Windows Store apps. The reason is that in many ways, Windows Desktop apps are simpler. They may not look as slick, and more importantly, they don't integrate with Windows 8 or provide the great, consistent user interface that you get with Windows Store apps. But there are a lot of important, fundamental concepts that you need to understand in order to build Windows Store apps effectively. Windows Desktop programming is a **great tool for exploring those fundamental concepts**. We'll return to programming Windows Store apps once we've laid down that foundation.

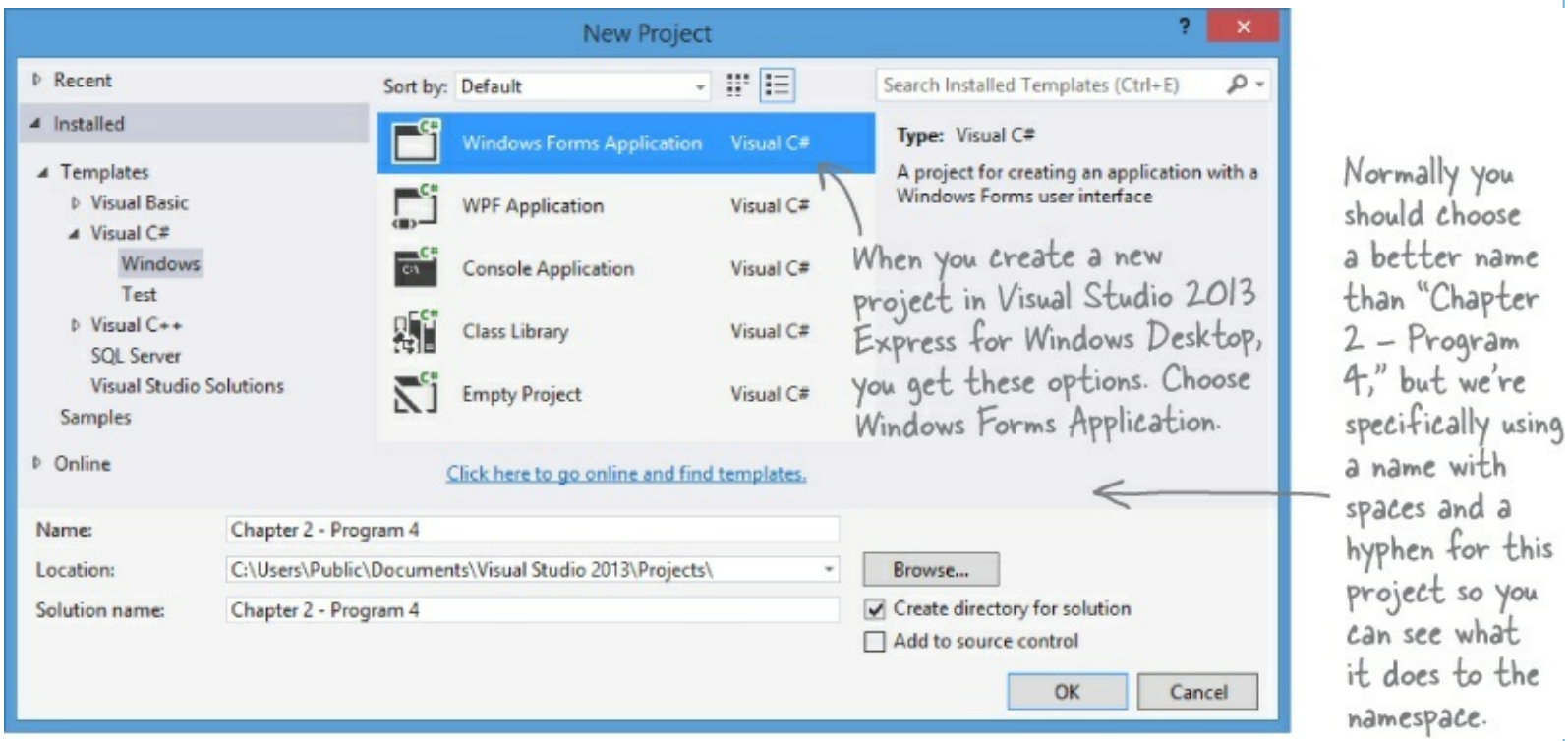
NOTE

Another great reason to learn Windows Desktop programming is that you get to see the same thing done more than one way. That's a really quick way to get concepts into your brain. Flip the page to see what we mean...

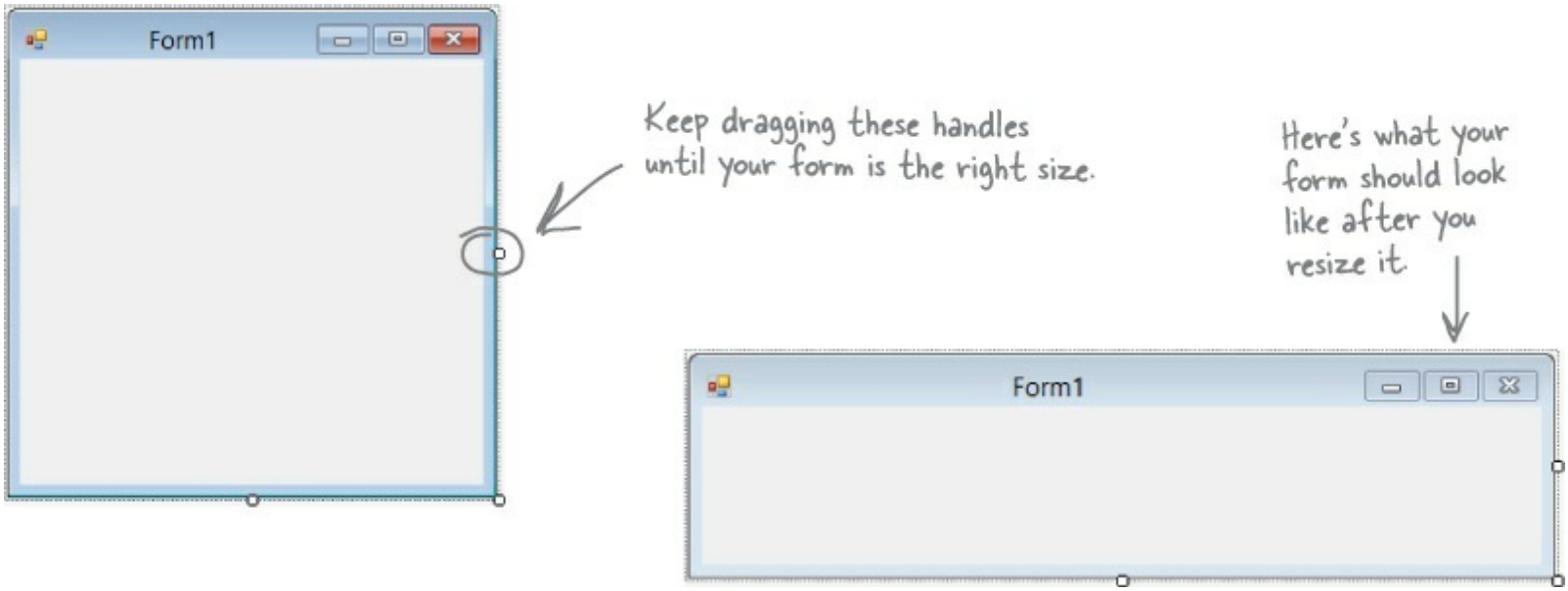
Rebuild your app for Windows Desktop

Start up Visual Studio 2013 for Windows Desktop and create a new project. This time, you'll see different options than before. Click on Visual C# and Windows, and **create a new Windows Forms Application project**.

DO THIS!

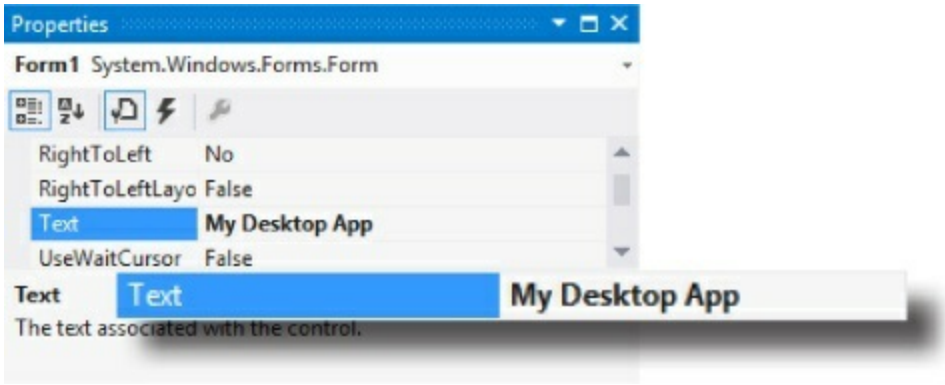


- **❶ WINDOWS FORMS APPS START WITH A FORM THAT YOU CAN RESIZE.**
  Your Windows Forms Application has a main window that you design using the designer in the IDE. Start by resizing it to 500x130. Find the handle on the form in the Designer window and drag to resize it. As you drag it, keep an eye on the changing numbers in the status bar in the IDE that show you the new size. Keep dragging until you see 500 x 130 in the status bar.

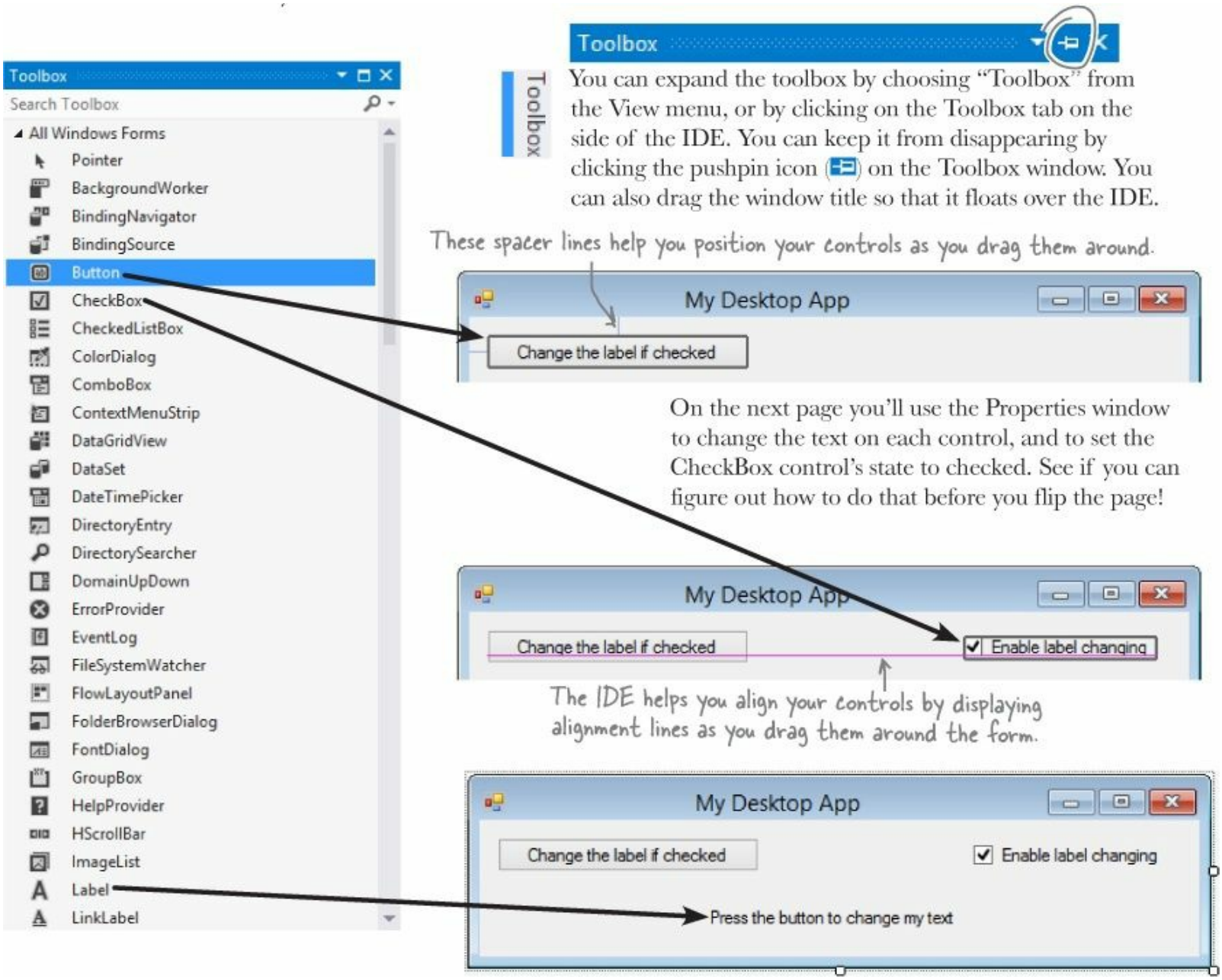


- **❷ CHANGE THE TITLE OF YOUR FORM.**

Right now the form has the default title ("Form1"). You can change that by clicking on the form to select it, and then changing the `Text` property in the Properties window.



- **❸ ADD A BUTTON, CHECKBOX, AND LABEL.**
  Open up the toolbox and drag a Button, CheckBox, and Label control onto your form.



Hint: you'll need to use the AutoSize property to get the Label control to look right.

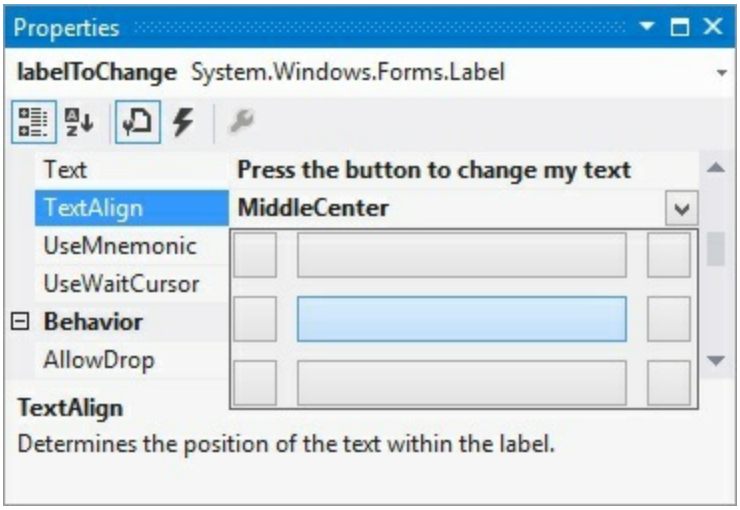
WATCH IT!

Make sure you're using the right Visual Studio

*If you're using the Express edition of Visual Studio 2013, you'll need to install two editions. You've been using Visual Studio 2013 for Windows to build Windows Store apps. Now you'll need to use **Visual Studio 2013 for Windows Desktop**. Luckily, both Express editions are available for free from Microsoft.*

- **❹ USE THE PROPERTIES WINDOW TO SET UP THE CONTROLS.**
Click on the Button control to select it. Then go to the Properties window and set its `Text` property:

Change the `Text` property for the CheckBox control and the Label control so they match the screenshot on the next page, and set the CheckBox's `Checked` property to `True`. Then select the Label control and set the TextAlign control to `MiddleCenter`. Use the Properties window to **set the names of your controls**. Name the Button `changeText`, set the CheckBox control's name to `enableCheckbox`, and name the Label control `labelToChange`. Look at the code below carefully and see if you can see how those names are used in the code.

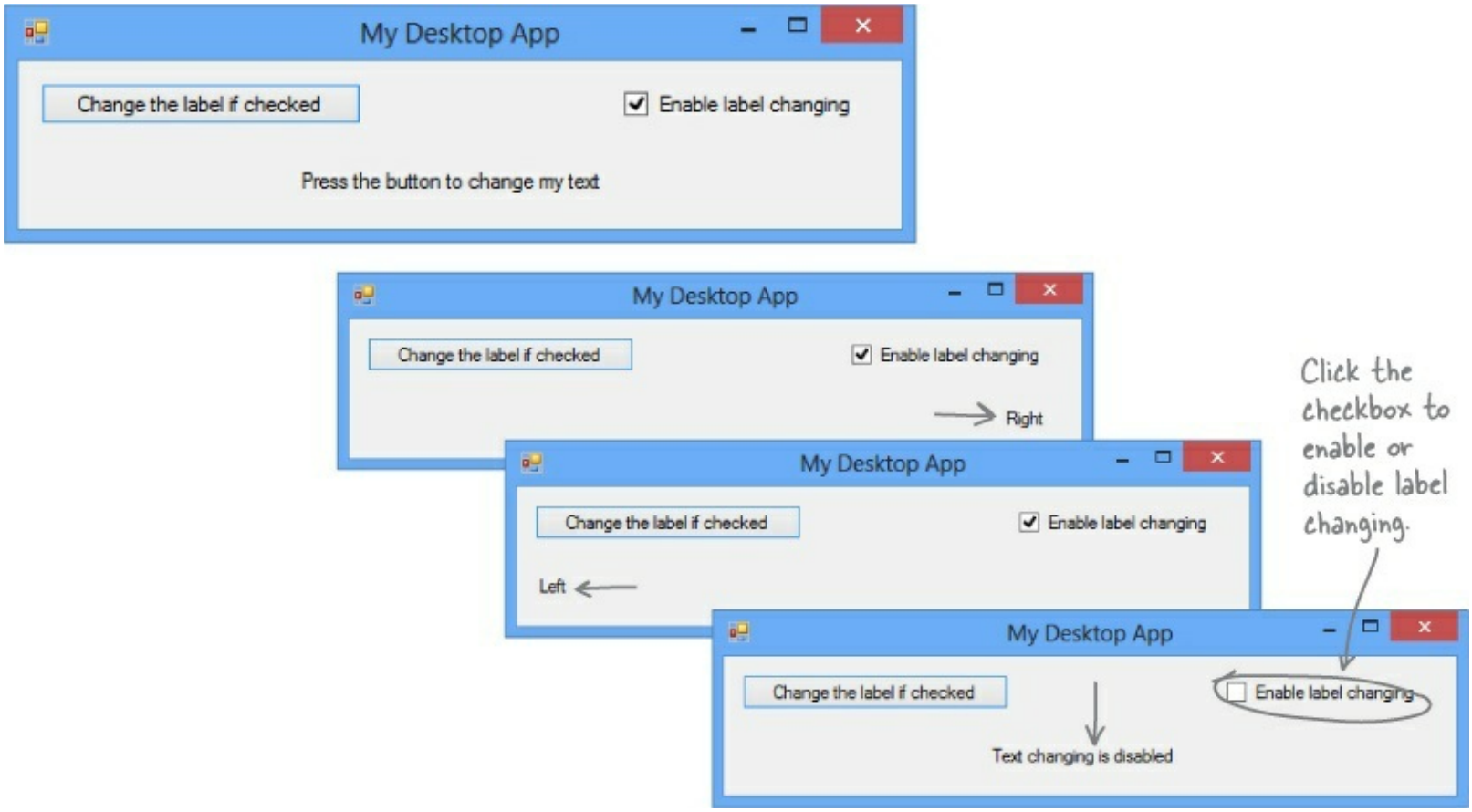


Change the `AutoSize` property on the Label control to `False`. Labels normally resize themselves based on their contents. Disabling `AutoSize` to `true` causes the drag handles to show up. Drag it so it's the **entire width of the window**.

- **❺ ADD THE EVENT HANDLER METHOD FOR YOUR BUTTON.**
Double-click on the button to make the IDE add an event handler method. Here's the code:

Debug your program in the IDE.

When you do, the IDE will build your program and run it, which pops up the main window that you built. Try clicking the button and checkbox.

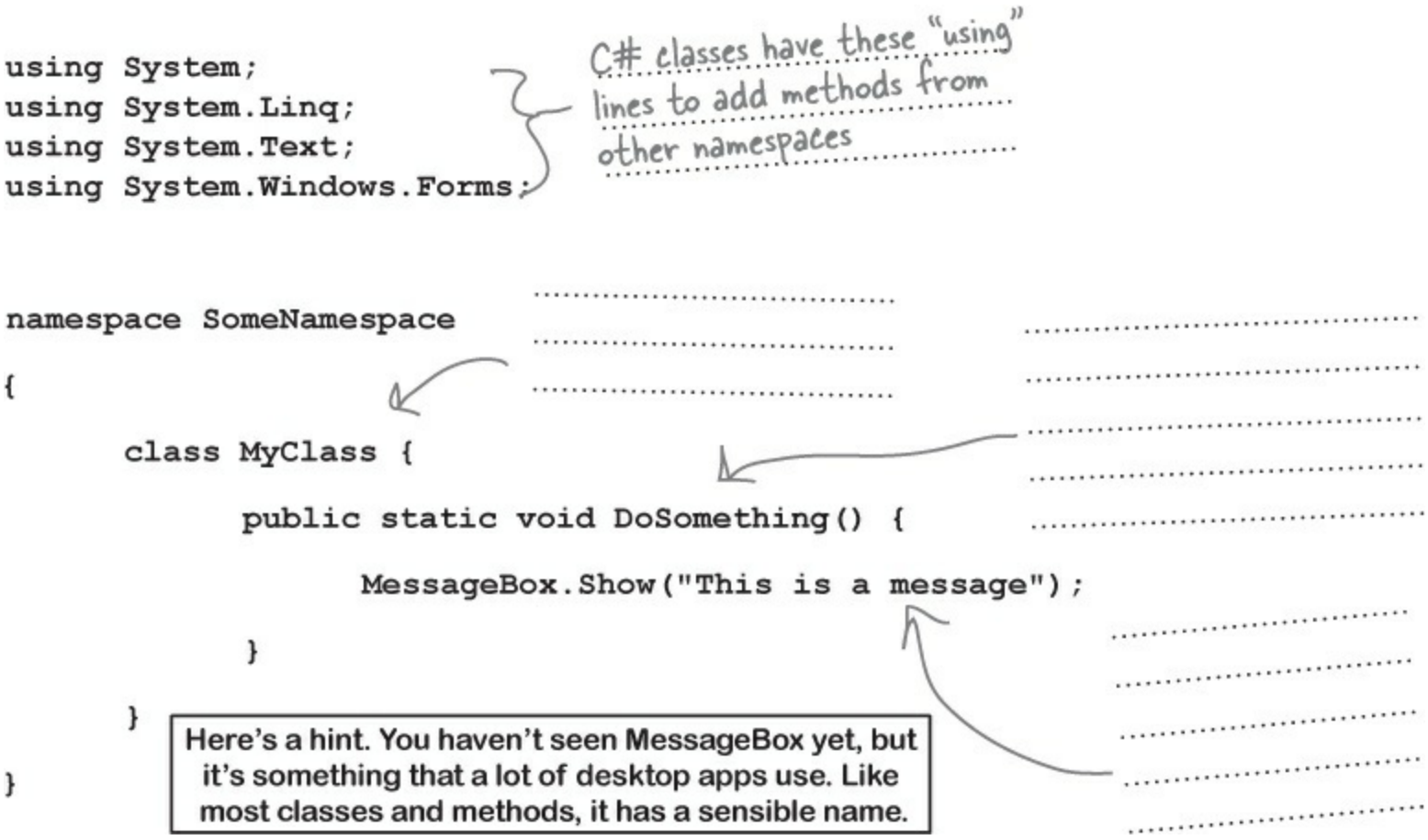


NOTE

When label changing is enabled, the label shows either Left or Right with matching alignment. If it's disabled, it shows a message that's centered.

SHARPEN YOUR PENCIL

Fill in the annotations so they describe the lines in this C# file that they're pointing to. We've filled in the first one for you. Can you **guess** what the last annotation should say?



Solution in So what happened? →

RELAX

Desktop apps are different, and that's good for learning.

Windows Desktop applications are a lot less slick than Windows Store apps because it's much harder (but not impossible) to build the kinds of advanced user interfaces that Windows Store apps give you. And that's a good thing for now! Because they're simple and straightforward, desktop apps are a great tool for learning the core C# concepts, and that will make it much easier for you to understand Windows Store apps when we return to them later.

Your desktop app knows where to start

When you created the new Windows Forms Application project, one of the files the IDE added was called *Program.cs*. Go to the Solution Explorer and double-click on it. It's got a class called `Program`, and inside that class is a method called `Main()`. That method is the **entry point**, which means that it's the very first thing that's run in your program.

NOTE

Here's some code the IDE built for you automatically in the last chapter. You'll find it in Program.cs.

YOUR CODE UP CLOSE

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Chapter_2___Program_4
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

The IDE generated this namespace based on the project name. We named ours "Chapter 2 – Program 4," so this is the namespace the IDE generated for us. We chose a name with spaces and a hyphen to show you how the IDE converts them to underscores in the namespace.

Lines that begin with two or more slashes are comments, which you can add anywhere you want. The slashes tell C# to ignore them.

Every time you run your program, it starts here, at the entry point.

This statement creates and displays the form, and ends the program when the form's closed.

I do declare!

The first part of every class or method is called a declaration.

Remember, this is just a starting point for you to dig into the code. But before you do, you'll need to know what you're looking at.

These are some of the "nuts and bolts" of desktop apps. You'll play with them on the next few pages so you can see what's going on behind the scenes. But most of the work you do on desktop apps will be done by dragging controls out of the toolbox and onto a form — and, obviously, editing C# code.

- **❶ C# AND .NET HAVE LOTS OF BUILT-IN FEATURES.**
  You'll find lines like this at the top of almost every C# class file. `System.Windows.Forms` is a **namespace**. The `using System.Windows.Forms` line makes everything in that namespace available to your program. In this case, that namespace has lots of visual elements in it, like buttons and forms.

> **NOTE**
>
> Your programs will use more and more namespaces like this one as you learn about C# and .NET's other built-in features throughout the book.
>
> If you didn't specify the "using" line, you'd have to explicitly type out System. Windows.Forms every time you use anything in that namespace.

- **❷ THE IDE CHOSE A NAMESPACE FOR YOUR CODE.**
  Here's the namespace the IDE created for you — it chose a namespace based on your project's name. All of the code in your program lives in this namespace.

> **NOTE**
>
> Namespaces let you use the same name in different programs, as long as those programs aren't also in the same namespace.

- **❸ YOUR CODE IS STORED IN A CLASS.**
  This particular class is called `Program`. The IDE created it and added the code that starts the program and brings up the form called Form1.

> **NOTE**
>
> You can have multiple classes in a single namespace.

- **❹ THIS CODE HAS ONE METHOD, AND IT CONTAINS SEVERAL STATEMENTS.**
  A namespace has classes in it, and classes have methods. Inside each method is a set of statements. In this program, the statements handle starting up the form. You already know that methods are where the action happens — every method **does** something.
- **❺ EACH DESKTOP APP HAS A SPECIAL KIND OF METHOD CALLED THE ENTRY POINT.**
  Every desktop app **must** have exactly one method called `Main`. Even though your program has a lot of methods, only one can be the first one that gets executed, and that's your `Main` method. C# checks every class in your code for a method that reads `static void Main()`. Then, when the program is run, the first statement in this method gets executed, and everything else follows from that first statement.

> **NOTE**
>
> Technically, a program can have more than one Main() method, and you can tell C# which one is the entry point... but you won't need to do that now.

Every desktop app must have exactly one method called Main. That method is the entry point for your code. When you run your code, the code in your Main() method is executed FIRST.

You can change your program's entry point

As long as your program has an entry point, it doesn't matter which class your entry point method is in, or what that method does. There's nothing magical or mysterious about how it works, or how your desktop app runs. You can prove it to yourself by changing your program's entry point.

DO THIS!

- ❶ Go back to the program you just wrote. Edit *Program.cs* and change the name of the `Main()` method to `NotMain()`. Now **try to build and run your program**. What happens? Can you guess why it happened?
- ❷ Now let's create a new entry point. **Add a new class** called *AnotherClass.cs*. You add a class to your program by right-clicking on the project name in the Solution Explorer and selecting "Add→Class...". Name your class file *AnotherClass.cs*. The IDE will add a class to your program called `AnotherClass`. Here's the file the IDE added:

> **NOTE**
>
> Right-click on the project in Properties and select "Add" and "Class..."

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Chapter_2___Program_4
{
    class AnotherClass
    {
    }
}
```

These four standard using lines were added to the file.

This class is in the same namespace that the IDE added when you first created the project.

The IDE automatically named the class based on the filename.

- ❸ Add a new `using` line to the top of the file: **`using System.Windows.Forms;`** Don't forget to end the line with a semicolon!
- ❹ Add this method to the **`AnotherClass`** class by typing it in between the curly brackets:

```
class AnotherClass
{
    public static void Main()
    {
        MessageBox.Show("Pow!");
    }
}
```

MessageBox is a class that lives in the System.Windows.Forms namespace, which is why you had to add the using line in step #3. Show() is a method that's part of the MessageBox class.

C# is case-sensitive! Make sure your upper- and lowercase letters match the example code.

Desktop apps use MessageBox.Show() to pop up windows with messages and alerts.

Now run it!

- ❺ Figure out how to fix your program so it pops up the app again.

> **NOTE**
>
> Hint: you only have to change two lines in two files to do it.

So what happened?

Instead of popping up the app you wrote, your program now shows this message box. When you made the new `Main()` method, you gave your program a new entry point. Now the first thing the program does is run the statements in that method — which means running that `MessageBox.Show()` statement. There's nothing else in that method, so once you click the OK button, the program runs out of statements to execute and then it ends.

SHARPEN YOUR PENCIL SOLUTION

Fill in the annotations so they describe the lines in this C# file that they're pointing to. We've filled in the first one for you.

```
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;


namespace SomeNamespace
{
      class MyClass {
            public static void DoSomething() {
                  MessageBox.Show("This is a message");
            }
      }
}
```

C# classes have these "using" lines to add methods from other namespaces.

All of the code lives in classes, so the program needs a class here.

This class has one method. Its name is "DoSomething," and when it's called it pops up a MessageBox.

This is a statement. When it's executed, it pops up a little window with a message inside of it.

When you change things in the IDE, you're also changing your code

The IDE is great at writing visual code for you. But don't take our word for it. Open up Visual Studio, **create a new Windows Forms Application project**, and see for yourself.

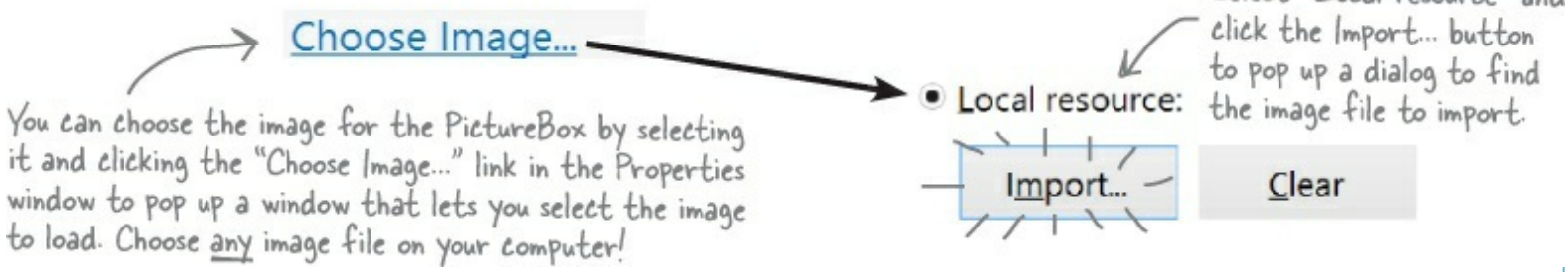
DO THIS!

- **❶ OPEN UP THE DESIGNER CODE.**
  Open the *Form1.Designer.cs* file in the IDE. But this time, instead of opening it in the Form Designer, open up its code by right-clicking on it in the Solution Explorer and selecting View Code. Look for the `Form1` class declaration:

```
partial class Form1
```

Notice how it's a partial class? We'll talk about that in a minute.

- **❷ OPEN UP THE FORM DESIGNER AND ADD A PICTUREBOX TO YOUR FORM.**
  Get used to working with more than one tab. Go to the Solution Explorer and open up the Form designer by double-clicking on *Form1.cs*. **Drag a new PictureBox control** out of the toolbox and onto the form. A PictureBox control displays a picture, which you can import from an image file.



- **❸ FIND AND EXPAND THE DESIGNER-GENERATED CODE FOR THE PICTUREBOX.**
  Then go back to the Form1.Designer.cs tab in the IDE. Scroll down and look for this line in the code:

Click on the + on the lefthand side of the line to expand the code. Scroll down and find these lines:

```
//
// pictureBox1
//
this.pictureBox1.Image = ((System.Drawing.Image)(resources.GetObject("pictureBox1.Image")));
this.pictureBox1.Location = new System.Drawing.Point(416, 160);
this.pictureBox1.Name = "pictureBox1";
this.pictureBox1.Size = new System.Drawing.Size(141, 147);
this.pictureBox1.TabIndex = 0;
this.pictureBox1.TabStop = false;
```

If you double-click on Form1.resx in the Solution Explorer, you'll see the image that you imported. The IDE imported our image and named it "pictureBox1.Image"—and here's the code that it generated to load that image into the PictureBox control so it's displayed.

Don't worry if the numbers in your code for the Location and Size lines are a little different than these. They'll vary depending on where you dragged your PictureBox control.

Wait, wait! What did that say?

Scroll back up for a minute. There it is, at the top of the Windows Form Designer–generated code section:

```
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
```

Most comments only start with two slashes (//). But the IDE sometimes adds these three-slash comments.

These are XML comments, and you can use them to document your code. Flip to "Leftovers" section #2 in the Appendix of this book to learn more about them.

There's nothing more attractive to a kid than a big sign that says, "Don't touch this!" Come on, you know you're tempted...let's go modify the contents of that method with the code editor! **Add a button to your form** called `button1` (you'll need to switch back to the designer)**, and then go ahead and do this:**

- **❶ CHANGE THE CODE THAT SETS THE BUTTON1.TEXT PROPERTY. WHAT DO YOU THINK IT WILL DO TO THE PROPERTIES WINDOW IN THE IDE?**
  Give it a shot — see what happens! Now go back to the form designer and check the `Text` property. Did it change?
- **❷ STAY IN THE DESIGNER, AND USE THE PROPERTIES WINDOW TO CHANGE THE NAME PROPERTY TO SOMETHING ELSE.**
  See if you can find a way to get the IDE to change the `Name` property. It's in the Properties window at the very top, under "(Name)". What happened to the code? What about the comment in the code?
- **❸ CHANGE THE CODE THAT SETS THE LOCATION PROPERTY TO (0,0) AND THE SIZE PROPERTY TO MAKE THE BUTTON REALLY BIG.**
  Did it work?
- **❹ GO BACK TO THE DESIGNER, AND CHANGE THE BUTTON'S BACKCOLOR PROPERTY TO SOMETHING ELSE.**
  Look closely at the *Form1.Designer.cs* code. Were any lines added?

> **NOTE**
>
> You don't have to save the form or run the program to see the changes. Just make the change in the code editor, and then click on the tab labeled "Form1.cs [Design]" to flip over to the form designer — the changes should show up immediately.

It's always easier to use the IDE to change your form's designer-generated code. But when you do, any change you make in the IDE ends up as a change to your project's code.

THERE ARE NO DUMB QUESTIONS

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

A: **A:** Your program has a whole lot of statements in it, but they're not all run at once. 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. The compiler will not build your code unless there is **exactly one method called** `Main()`, which we call the entry point. The program starts running with the first statement in `Main()`.

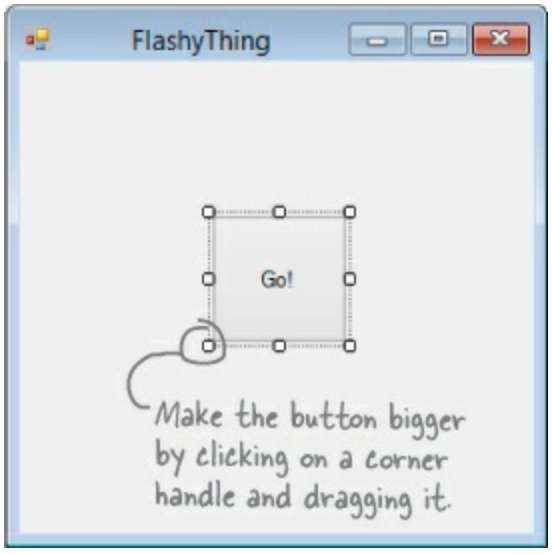
EXERCISE

Desktop apps aren't nearly as easy to animate as Windows Store apps, but it's definitely possible! Let's build something **flashy** to prove it. Start by creating a new **Windows Forms Application**.

NOTE

Remember, to create a Windows Forms Application you need to be using Visual Studio for Windows Desktop.

- **❶ HERE'S THE FORM TO BUILD.**
  Here's a hint for this exercise: if you declare a variable inside a for loop — for (int c = 0; ...) — then that variable's only valid inside the loop's curly brackets. So if you have two for loops that both use the variable, you'll either declare it in each loop or have one declaration outside the loop. And if the variable c is already declared outside of the loops, you can't use it in either one.



- **❷ MAKE THE FORM BACKGROUND GO ALL PSYCHEDELIC!**
  When the button's clicked, make the form's background color cycle through a whole lot of colors! Create a loop that has a variable `c` go from 0 to 253. Here's the block of code that goes inside the curly brackets:

```
this.BackColor = Color.FromArgb(c, 255 - c, c);

Application.DoEvents();
```

This line tells the program to stop your loop momentarily and do the other things it needs to do, like refresh the form, check for mouse clicks, etc. Try taking out this line and see what happens. The form doesn't redraw itself, because it's waiting until the loop is done before it deals with those events.

For now, you'll use Application.DoEvents() to make sure your form stays responsive while it's in a loop, but it's kind of a hack. You shouldn't use this code outside of a toy program like this. Later on in the book, you'll learn about a much better way to let your programs do more than one thing at a time!

I'M TICKLED PINK!

The .NET Framework has a bunch of predefined colors like Blue and Red, but it also lets you make your own colors using the Color. FromArgb() method, by specifying three numbers: a red value, a green value, and a blue value.

- **❸ MAKE IT SLOWER.**
  Slow down the flashing by adding this line after the `Application.DoEvents()` line:

```
System.Threading.Thread.Sleep(3);
```

This statement inserts a 3 millisecond delay in the loop. It's a part of the .NET Framework, and it's in the System.Threading namespace.

- **❹ MAKE IT SMOOTHER.**
  Let's make the colors cycle back to where they started. Add another loop that has `c` go from 254 down to 0. Use the same block of code inside the curly brackets.
- **❺ KEEP IT GOING.**
  Surround your two loops with another loop that continuously executes and doesn't stop, so that when the button is pressed, the background starts changing colors and then keeps doing it. (Hint: the `while (true)` loop will run forever!)

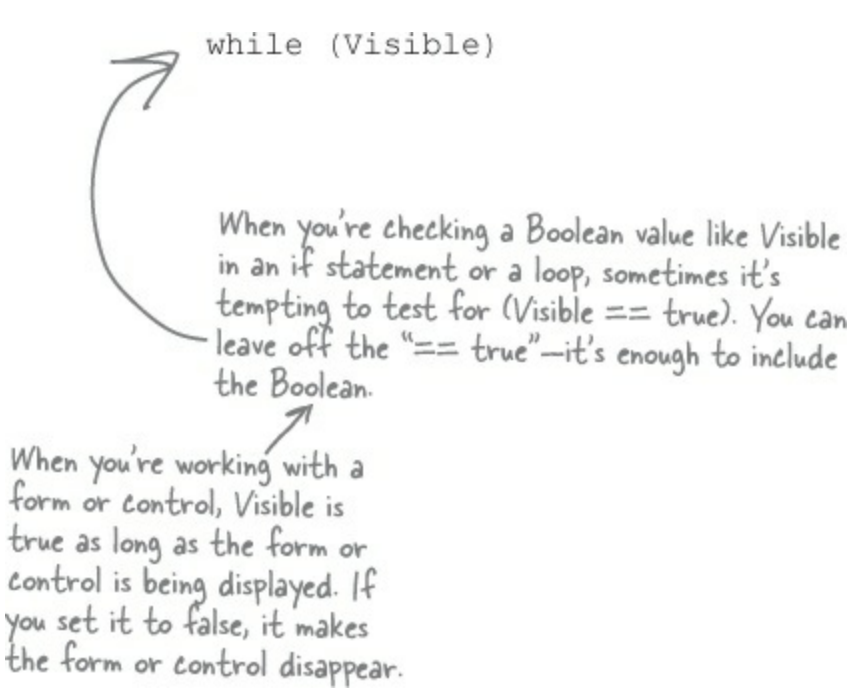
NOTE

When one loop is inside another one, we call it a "nested" loop.

UH OH! THE PROGRAM DOESN'T STOP!

Run your program in the IDE. Start it looping. Now close the window. Wait a minute — the IDE didn't go back into edit mode! It's acting like the program is still running. You need to actually stop the program using the square stop button in the IDE (or select Stop Debugging from the Debug menu).

- **❻ MAKE IT STOP.**
  Make the loop you added in step #5 stop when the program is closed. Change your outer loop to this:



Now run the program and click the X box in the corner. The window closes, and then the program stops! Except...there's a delay of a few seconds before the IDE goes back to edit mode.

Can you figure out what's causing that delay? Can you fix it so the program ends immediately when you close the window?

NOTE

Hint: the `&&` operator means "AND." It's how you string a bunch of conditional tests together into one big test that's true only if the first test is true AND the second is true AND the third, etc. And it'll come in handy to solve this problem.

EXERCISE SOLUTION

When the IDE added this method, it added an extra return before the curly bracket. Sometimes we'll put the bracket on the same line like this to save space—but C# doesn't care about extra space, so this is perfectly valid.

Consistency is generally really important to make it easy for people to read code. But we're purposefully showing you different ways, because you'll need to get used to reading code from different people using different styles.

Sometimes we won't show you the entire code in the solution, just the bits that changed. All of the logic in the FlashyThing project is in this button1_Click() method that the IDE added when you double-clicked the button in the form designer.

```
private void button1_Click(object sender, EventArgs e) {
    while (Visible) {
        for (int c = 0; c < 254 && Visible; c++) {
            this.BackColor = Color.FromArgb(c, 255 - c, c);
            Application.DoEvents();
            System.Threading.Thread.Sleep(3);
        }
        for (int c = 254; c >= 0 && Visible; c--) {
            this.BackColor = Color.FromArgb(c, 255 - c, c);
            Application.DoEvents();
            System.Threading.Thread.Sleep(3);
        }
    }
}
```

The outer loop keeps running as long as the form is visible. As soon as it's closed, Visible is false, and the while will stop looping.

The first for loop makes the colors cycle one way, and the second for loop reverses them so they look smooth.

We used && Visible instead of && Visible == true. It's just like saying "if it's visible" instead of "if it's true that it's visible"—they mean the same thing.

We fixed the extra delay by using the && operator to make each of the for loops also check Visible. That way the loop ends as soon as Visible turns false.

Can you figure out what's causing that delay? Can you fix it so the program ends immediately when you close the window?

The delay happens because the `for` loops need to finish before the `while` loop can check if `Visible` is still `true`. You can fix it by adding **`&& Visible`** to the conditional test in each for loop.

Was your code a little different than ours? There's more than one way to solve any programming problem (e.g., you could have used while loops instead of for loops). If your program works, then you got the exercise right!

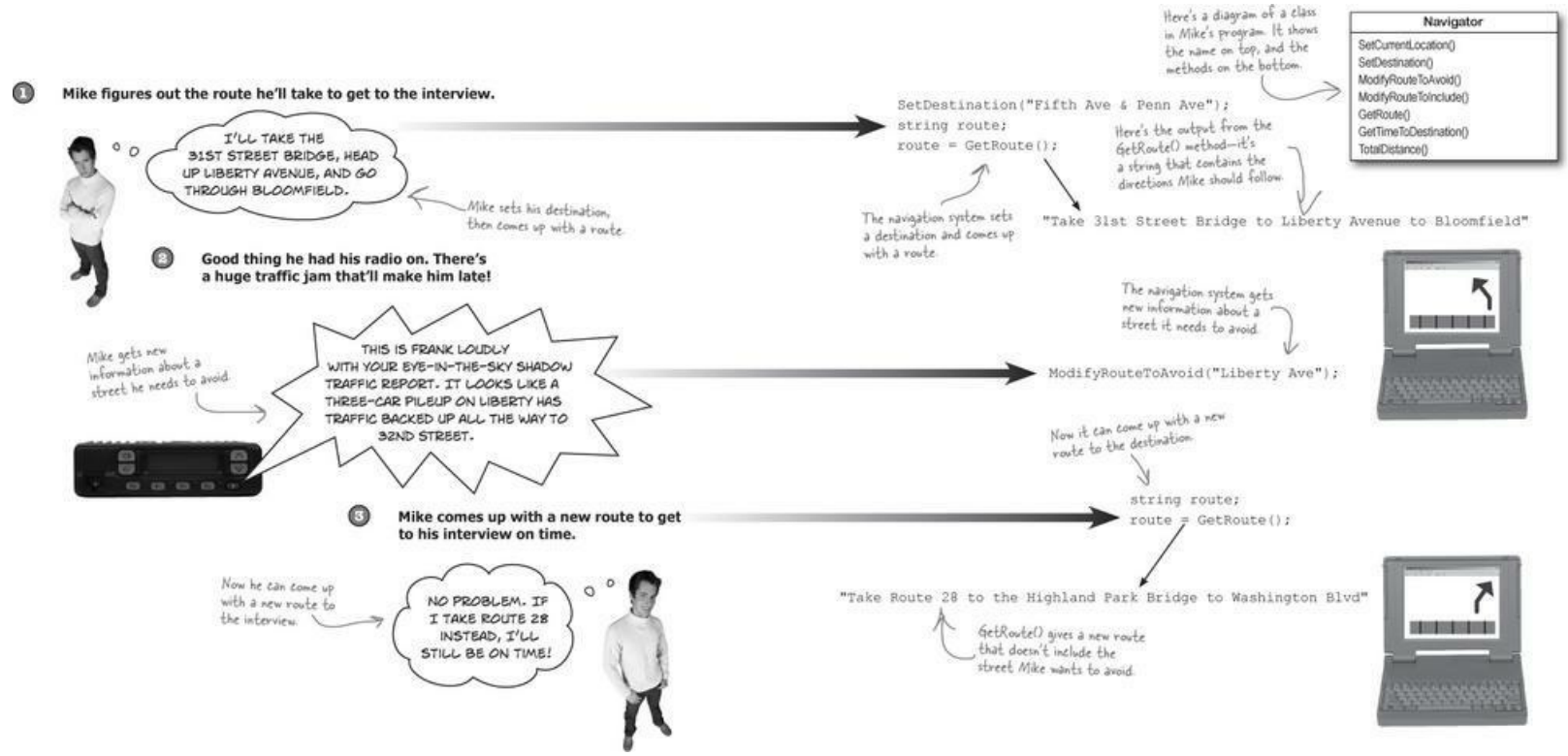
Chapter 3. Objects: Get Oriented!: Making code make sense

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, you end up with code that's *intuitive* to write, and easy to read and change.

How Mike thinks about his problems

Mike's a programmer about to head out to a job interview. He can't wait to show off his C# skills, but first he has to get there — and he's running late!



How Mike’s car navigation system thinks about his problems

Mike built his own GPS navigation system, which he uses to help him get around town.

Mike’s navigation system solves the street navigation problem the same way he does.

Mike's Navigator class has methods to set and modify routes

Mike's `Navigator` class has methods, which are where the action happens. But unlike the `button_Click()` methods in the forms you've built, they're all focused around a single problem: navigating a route through a city. That's why Mike stuck them together into one class, and called that class `Navigator`.

Mike designed his `Navigator` class so that it's easy to create and modify routes. To get a route, Mike's program calls the `SetDestination()` method to set the destination, and then uses the `GetRoute()` method to put the route into a string. If he needs to change the route, his program calls the `ModifyRouteToAvoid()` method to change the route so that it avoids a certain street, and then calls the `GetRoute()` method to get the new directions.

Mike chose method names that would make sense to someone who was thinking about how to navigate a route through a city.

```
class Navigator {

   public void SetCurrentLocation(string locationName) { ... }

   public void SetDestination(string destinationName) { ... }

   public void ModifyRouteToAvoid(string streetName) { ... }

   public string GetRoute() { ... }

}
```

This is the **return type** of the method. It means that the statement calling the GetRoute() method can use it to set a string variable that will contain the directions. When it's **void**, that means the method doesn't return anything.

```
string route =
       GetRoute();
```

Some methods have a return value

Every method is made up of statements that do things. Some methods just execute their statements and then exit. But other methods 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**.

The **`return`** statement tells the method to immediately exit. If your method doesn't have a return value — which means it's declared with a return type of `void` — then the `return` statement doesn't need any values or variables ("`return;`"), and you don't always have to have one in your method. But if the 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. The method uses the two **parameters** to calculate the result.

```
public int MultiplyTwoNumbers(int firstNumber, int secondNumber) {
    int result = firstNumber * secondNumber;
    return result;
}
```

This **return** statement passes the value back to the statement that called the method.

Here's a statement that calls a method to multiply two numbers. It returns an `int`:

```
int myResult = MultiplyTwoNumbers(3, 5);
```

Methods can take values like 3 and 5. But you can also use variables to pass values to a method.

BULLET POINTS

- Classes have methods that contain statements that perform actions. You can design a class that is easy to use by choosing methods that make sense.
- Some methods have a **return type**. You set a method's return type in its declaration. A method with a declaration that starts "`public int`" 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 you've got a method that's declared "`public string`" 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; }`

Use what you've learned to build a program that uses a class

Let's hook up a form to a class, and make its button call a method inside that class.

DO THIS!

- ❶ Create a **new Windows Forms Application project** in the IDE. Then add a class file to it called *Talker.cs* by right-clicking on the project in the Solution Explorer and selecting "Class..." from the Add menu. When you name your new class file "Talker.cs," the IDE will automatically name the class in the new file `Talker`. Then it'll pop up the new class in a new tab inside the IDE.
- ❷ Add `using System.Windows.Forms;` to the top of the class file. Then add code to the class:

```
class Talker {
    public static int BlahBlahBlah(string thingToSay, int numberOfTimes)
    {
        string finalString = "";
        for (int count = 0; count < numberOfTimes; count++)
        {
            finalString = finalString + thingToSay + "\n";
        }
        MessageBox.Show(finalString);
        return finalString.Length;
    }
}
```

This statement declares a finalString variable and sets it equal to an empty string.

This line of code adds the contents of thingToSay and a line break ("\n") onto the end of it to the finalString variable.

The BlahBlahBlah() method's return value is an integer that has the total length of the message it displayed. You can add ".Length" to any string to figure out how long it is.

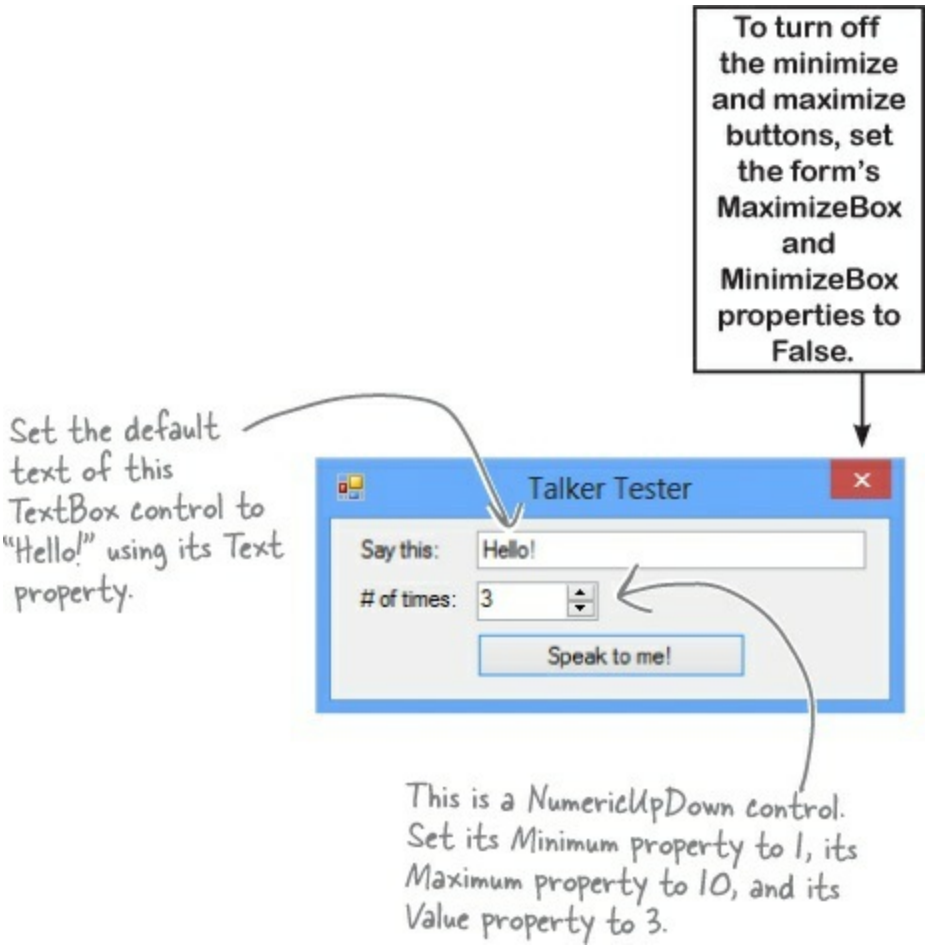
This is called a **property**. Every string has a property called Length. When it calculates the length of a string, a line break ("\n") counts as one character.

Flip the page to keep going!

So what did you just build?

The new class has one method called `BlahBlahBlah()` that takes two parameters. The first parameter is a string that tells it something to say, and the second is the number of times to say it. When it's called, it pops up a message box with the message repeated a number of times. Its return value is the length of the string. The method needs a string for its `thingToSay` parameter and a number for its `numberOfTimes` parameter. It'll get those parameters from a form that lets the user enter text using a **`TextBox`** control and a number using a **`NumericUpDown`** control.

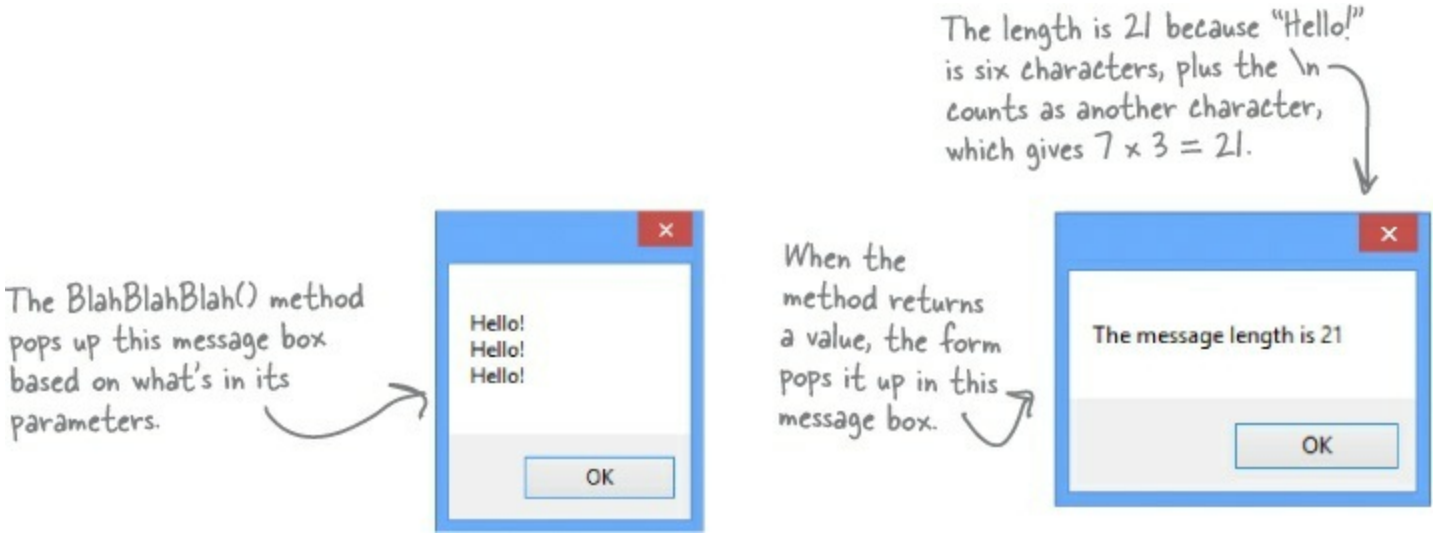
Now add a form that uses your new class!



- ❸ Make your project's form look like this.
  Then double-click on the button and have it run this code that calls `BlahBlahBlah()` and assigns its return value to an integer called `len`:

```
private void button1_Click(object sender, EventArgs e)
{
    int len = Talker.BlahBlahBlah(textBox1.Text, (int)numericUpDown1.Value);
    MessageBox.Show("The message length is " + len);
}
```

- ❹ Now run your program! Click the button and watch it pop up two message boxes. The class pops up the first message box, and the form pops up the second one.



You can add a class to your project and share its methods with the other classes in the project.

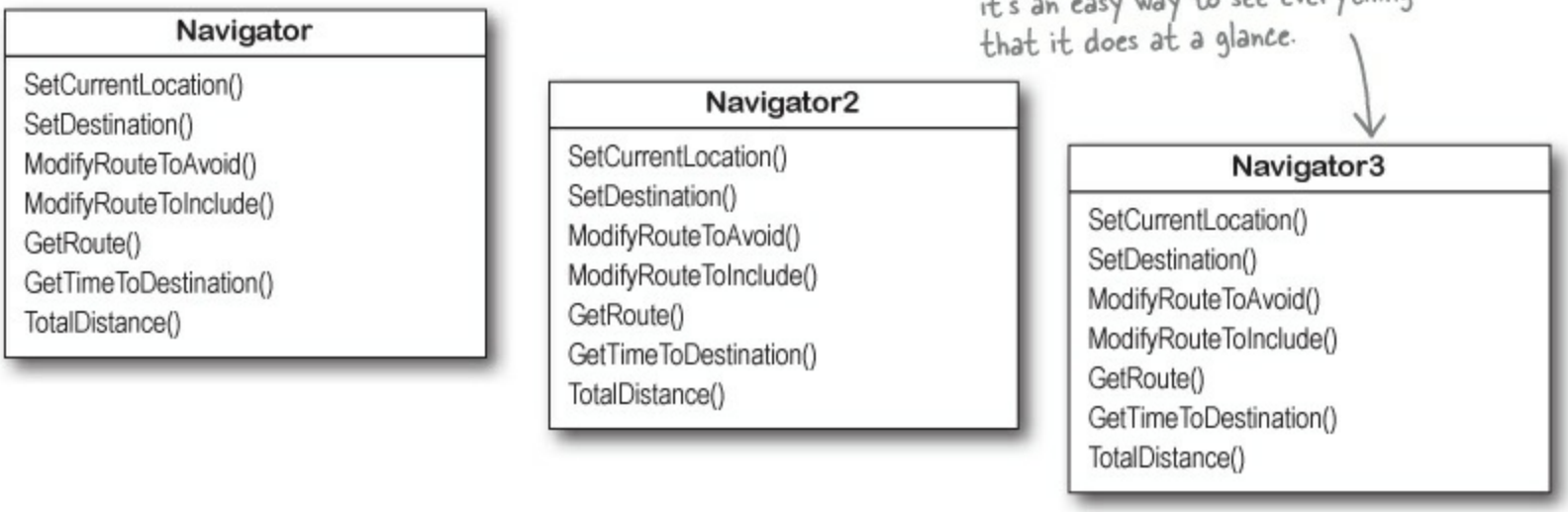
Mike gets an idea

The interview went great! But the traffic jam this morning got Mike thinking about how he could improve his navigator.

He could create three different Navigator classes...

Mike *could* copy the `Navigator` class code and paste it into two more classes. Then his program could store three routes at once.

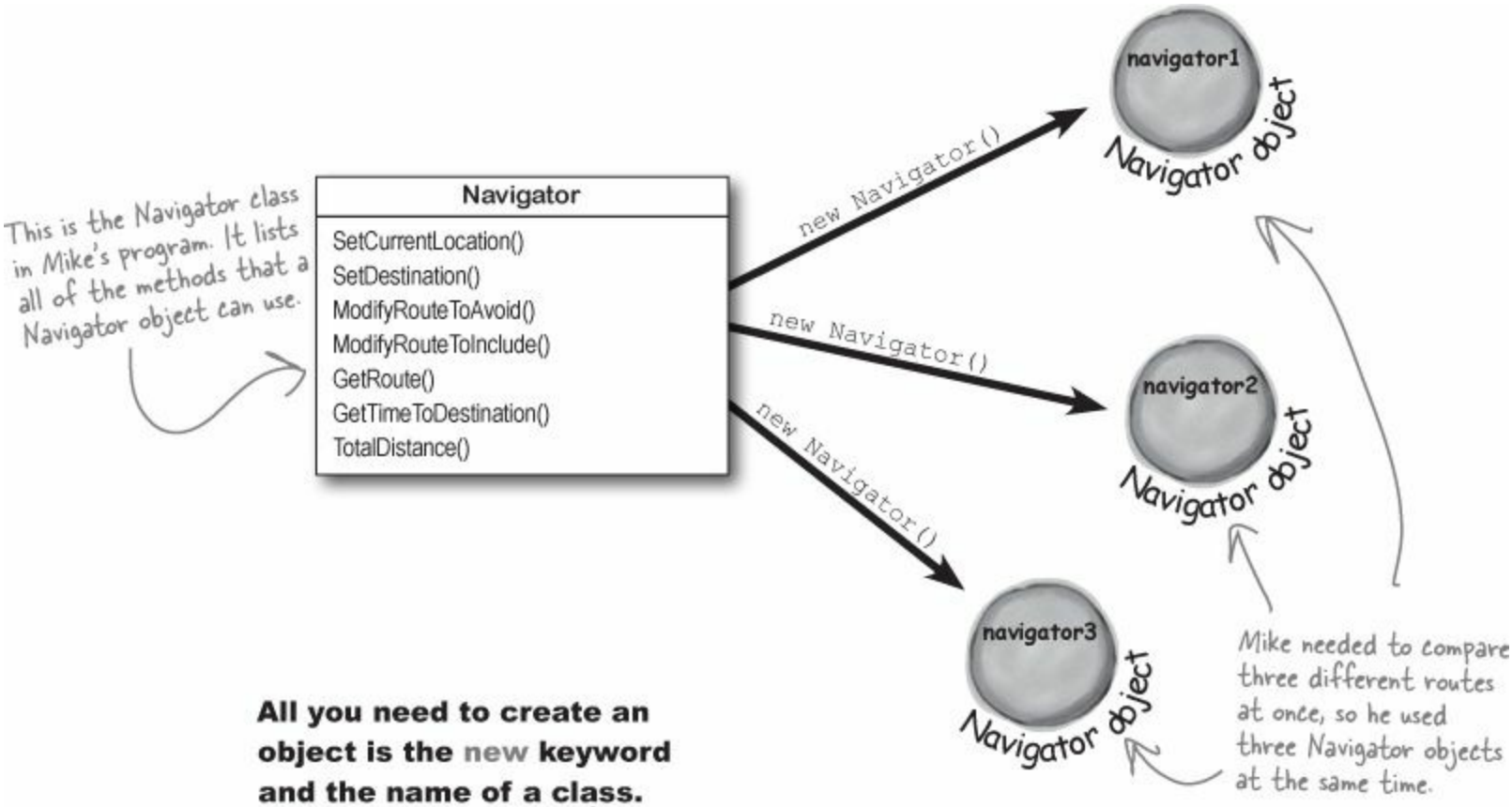
This box is a **class diagram**. It lists all of the methods in a class, and it's an easy way to see everything that it does at a glance.



Right! 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 a bunch of routes. But it could be a bunch of people, or aliens, or music files, or anything. All of those programs have one thing in common: they always need to treat the same kind of thing in the same way, no matter how many of the thing they're dealing with.

Mike can use objects to solve his problem

Objects are C#'s tool that you use to work with a bunch of similar things. Mike can use objects to program his `Navigator` class just once, but use it *as many times as he wants* in a program.



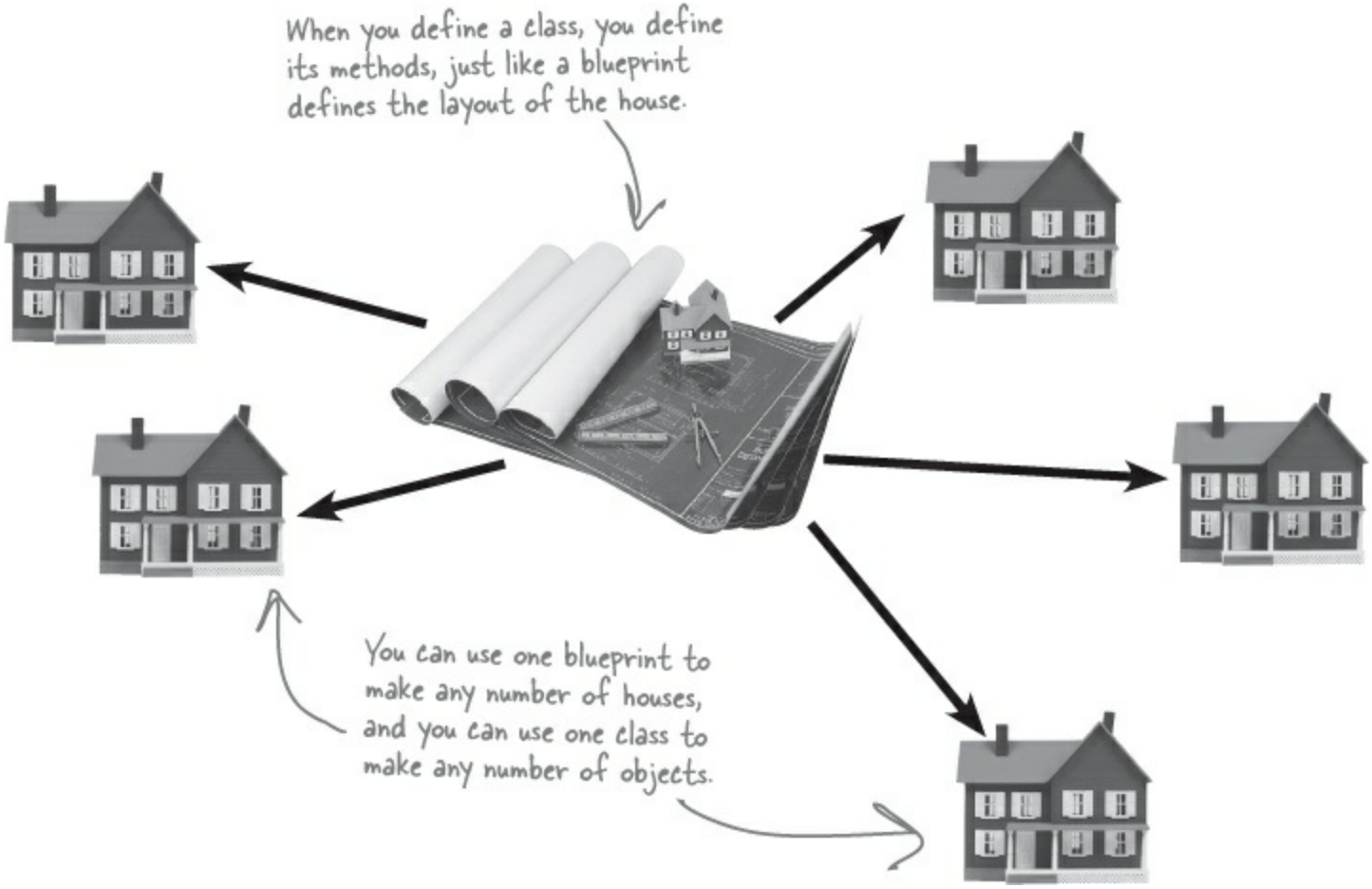
All you need to create an object is the new keyword and the name of a class.

```
Navigator navigator1 = new Navigator();
navigator1.SetDestination("Fifth Ave & Penn Ave");
string route;
route = navigator1.GetRoute();
```

Now you can use the object! When you create an object from a class, that object has all of the methods from that class.

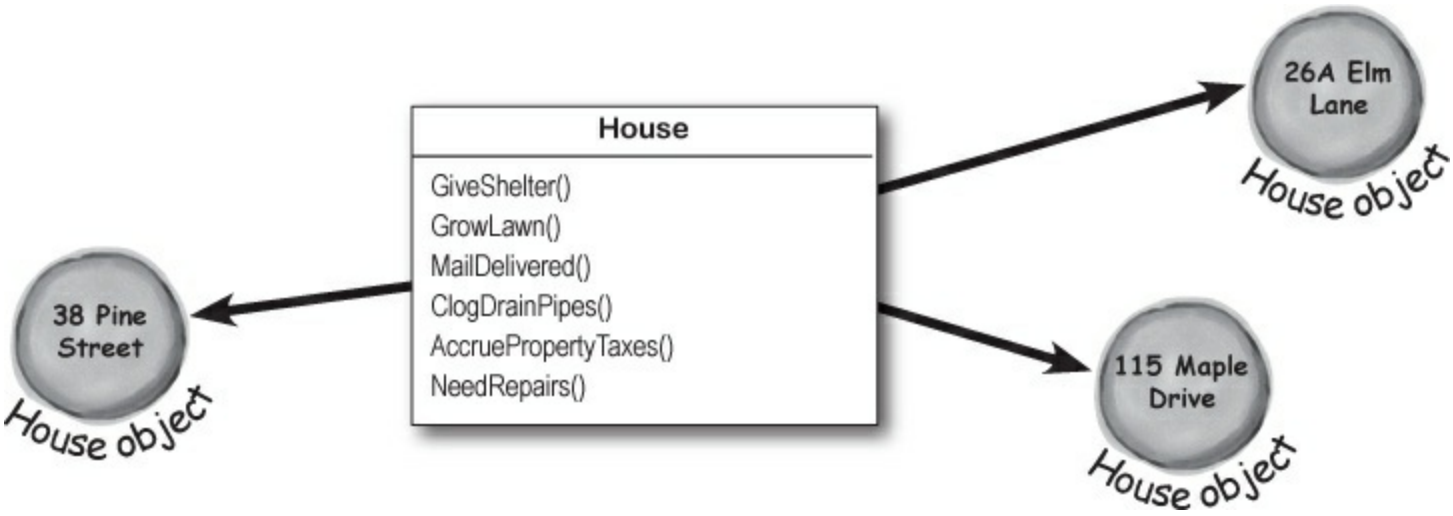
You use a class to build an object

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



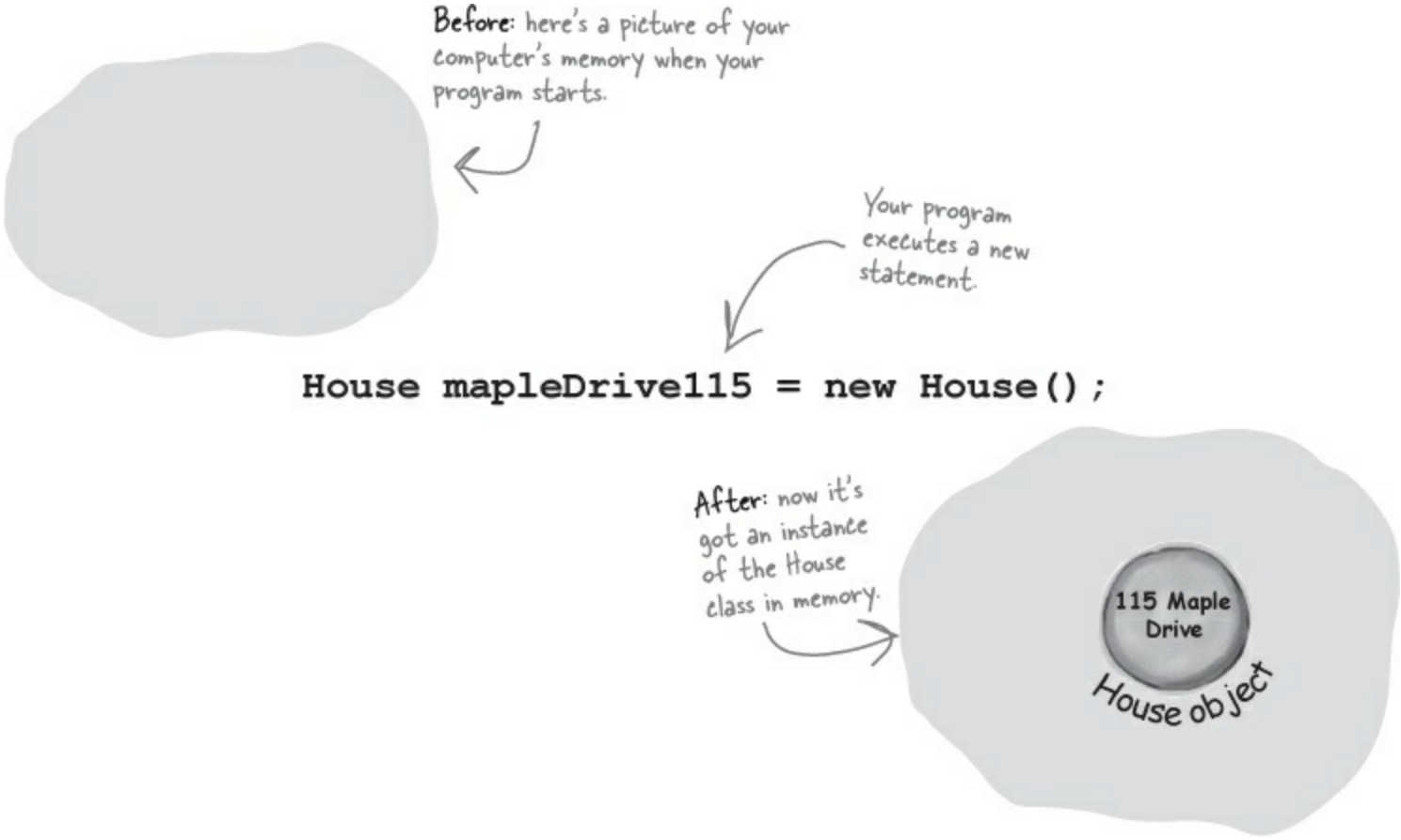
An object gets its methods from its class

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

Guess what...you already know this stuff ! Everything in the toolbox is a class: there's a `Button` class, a `TextBox` class, a `Label` class, etc. When you drag a button out of the toolbox, the IDE automatically creates an instance of the `Button` class and calls it `button1`. When you drag another button out of the toolbox, it creates another instance called `button2`. Each instance of `Button` has its own properties and methods. But every button acts exactly the same way, because they're all instances of the same class.



Check it out for yourself!

DO THIS!

Open any project that uses a button called `button1`, and use the IDE to search the entire project for the text "**`button1 = new`**". You'll find the code that the IDE added to the form designer to create the instance of the `Button` class.

NOTE

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.*

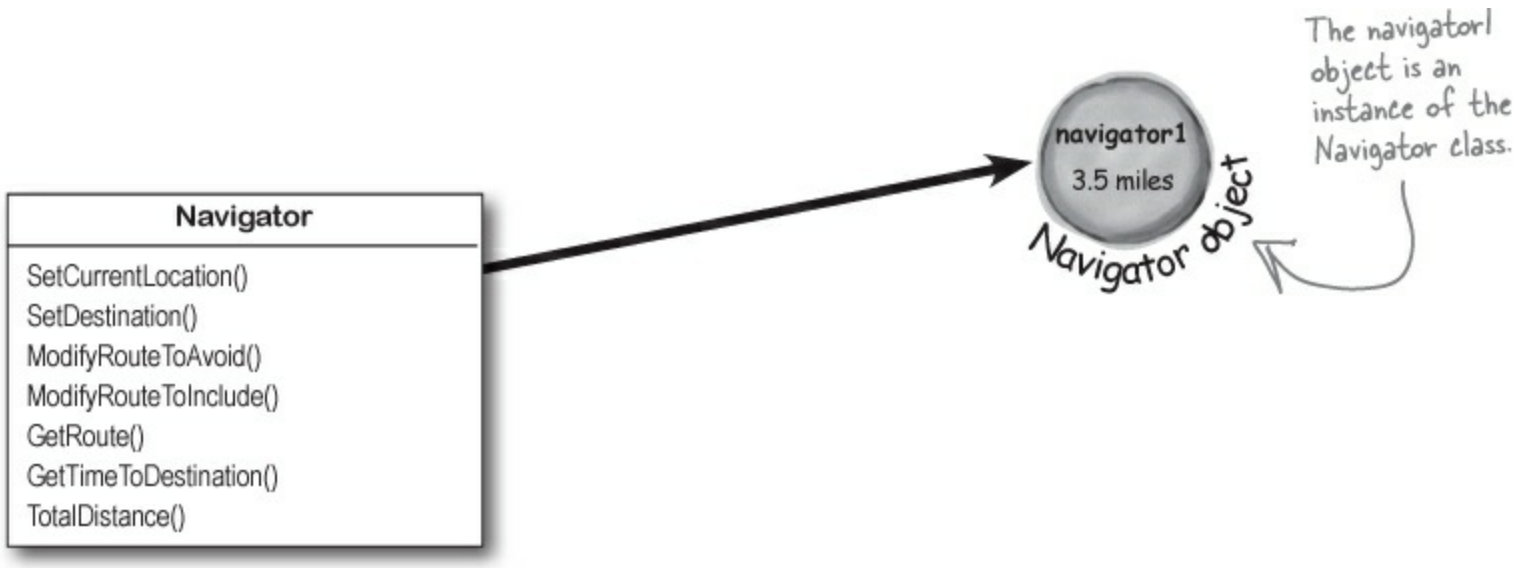
A better solution...brought to you by objects!

Mike came up with a new route comparison program that uses objects to find the shortest of three different routes to the same destination. Here's how he built his program.

- ❶ Mike set up a GUI with a textbox — `textBox1` contains the **destination** for the three routes. Then he added `textBox2`, which has a street that one of the routes should **avoid**; and `textBox3`, which contains a different street that the third route has to **include**.

> **NOTE**
>
> GUI stands for Graphical User Interface, which is what you're building when you make a form in the form designer.

- ❷ He created a `Navigator` object and set its destination.




```
string destination = textBox1.Text;
Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
route = navigator1.GetRoute();
```

- ❸ Then he added a second `Navigator` object called `navigator2`. He called its `SetDestination()` method to set the destination, and then he called its `ModifyRouteToAvoid()` method.

> **NOTE**
>
> The SetDestination(), ModifyRouteToAvoid(), and ModifyRouteToInclude() methods all take a string as a parameter.

- ❹ The third `Navigator` object is called `navigator3`. Mike set its destination, and then called its `ModifyRouteToInclude()` method.

- ❺ Now Mike can call each object's `TotalDistance()` method to figure out which route is the

shortest. And he only had to write the code once, not three times!

Any time you create a new object from a class, it's called creating an instance of that class.

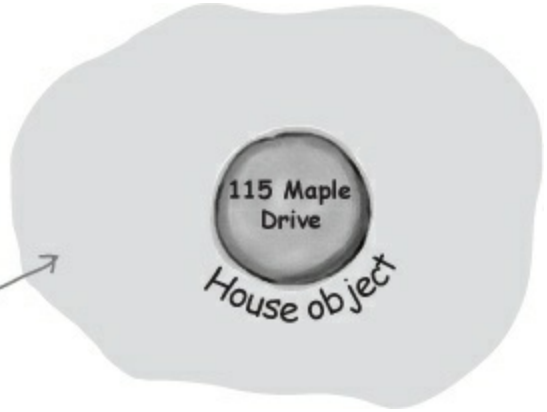
That's right, we didn't. A geographic navigation program is a really complicated thing to build. But complicated programs follow the same patterns as simple ones. Mike's navigation program is an example of how someone would use objects in real life.

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.



After we've introduced a concept, we'll give you a chance to get it into your brain. Sometimes we'll follow up the theory with a writing exercise — like the *Sharpen your pencil* exercise on the next page. Other times, we'll jump straight into code. This combination of theory and practice is an effective way to get these concepts off of the page and stuck in your brain.

A little advice for the code exercises

If you keep a few simple things in mind, it'll make the code exercises go 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 2012! But it's easy to accidentally type things wrong (like typing a one instead of a lowercase L).
-

SHARPEN YOUR PENCIL

Follow the same steps that Mike followed earlier in the chapter to write the code to create `Navigator` objects and call their methods.

```
string destination = textBox1.Text;
string route2StreetToAvoid = textBox2.Text;
string route3StreetToInclude = textBox3.Text;
```

We gave you a head start. Here's the code Mike wrote to get the destination and street names from the text boxes.

```
Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
int distance1 = navigator1.TotalDistance();
```

And here's the code to create the navigator object, set its destination, and get the distance.

1. Create the `navigator2` object, set its destination, call its `ModifyRouteToAvoid()` method, and use its `TotalDistance()` method to set an integer variable called `distance2`.

```
Navigator navigator2 = _____________________________
navigator2. _____________________________
navigator2. _____________________________
int distance2 = _____________________________
```

2. Create the `navigator3` object, set its destination, call its `ModifyRouteToInclude()` method, and use its `TotalDistance()` method to set an integer variable called `distance3`.

_____

_____

_____

_____

The Math.Min() method built into the .NET Framework compares two numbers and returns the smallest one. Mike used it to find the shortest distance to the destination.

```
int shortestDistance = Math.Min(distance1, Math.Min(distance2, distance3));
```

SHARPEN YOUR PENCIL SOLUTION

Follow the same steps that Mike followed earlier in the chapter to write the code to create `Navigator` objects and call their methods.

```
string destination = textBox1.Text;
string route2StreetToAvoid = textBox2.Text;
string route3StreetToInclude = textBox3.Text;

Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
int distance1 = navigator1.TotalDistance();
```

We gave you a head start. Here's the code Mike wrote to get the destination and street names from the text boxes.

And here's the code to create the navigator object, set its destination, and get the distance.

1. Create the `navigator2` object, set its destination, call its `ModifyRouteToAvoid()` method, and use its `TotalDistance()` method to set an integer variable called `distance2`.

```
Navigator navigator2 = __new Navigator()_____________________
navigator2. ___SetDestination(destination);_____________________
navigator2. __ModifyRouteToAvoid(route2StreetToAvoid);__________
int distance2 = __navigator2.TotalDistance();_____________________
```

2. Create the `navigator3` object, set its destination, call its `ModifyRouteToInclude()` method, and use its `TotalDistance()` method to set an integer variable called `distance3`.

NOTE

Navigator navigator3 = new Navigator()

navigator3.SetDestination(destination);

navigator3.ModifyRouteToInclude(route3StreetToInclude);

int distance3 = navigator3.TotalDistance();

The Math.Min() method built into the .NET Framework compares two numbers and returns the smallest one. Mike used it to find the shortest distance to the destination.

```
int shortestDistance = Math.Min(distance1, Math.Min(distance2, distance3));
```

Yes! That's why you used the `static` keyword in your methods.

Take another look at the declaration for the `Talker` class you built a few pages ago:

```
class Talker
{
  public static int BlahBlahBlah(string thingToSay, int numberOfTimes)
  {
    string finalString = "";
```

When you called the method, you didn't create a `new` instance of `Talker`. You just did this:

```
Talker.BlahBlahBlah("Hello hello hello", 5);
```

That's how you call `static` methods, and you've been doing that all along. If you take away the `static` keyword from the `BlahBlahBlah()` method declaration, then you'll have to create an instance of `Talker` in order to call the method. Other than that distinction, static methods are just like object methods. You can pass parameters, they can return values, and they live in classes.

There's one more thing you can do with the `static` keyword. You can mark your **whole class** as static, and then all of its methods **must** be `static` too. If you try to add a nonstatic method to a static class, it won't compile.

THERE ARE NO DUMB QUESTIONS

Q: **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: **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. A lot of people have trouble remembering that, because the word "static" isn't really all that intuitive.

Q: **Q: So I can't use my class until I create an instance of an object?**

A: **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: **Q: Then why would I want a method that needs an instance? Why wouldn't I make all my methods static?**

A: **A:** Because if you have an object that's keeping track of certain data — like Mike's instances of his `Navigator` class that each kept track of a different route — then you can use each instance's methods to work with that data. So when Mike called his `ModifyRouteToAvoid()` method in the `navigator2` instance, it only affected the route that was stored in that particular instance. It didn't affect the `navigator1` or `navigator3` objects. That's how he was able to work with three different routes at the same time — and his program could keep track of all of it.

Q: **Q: So how does an instance keep track of data?**

A: **A:** Turn the page and find out!

An instance uses fields to keep track of things

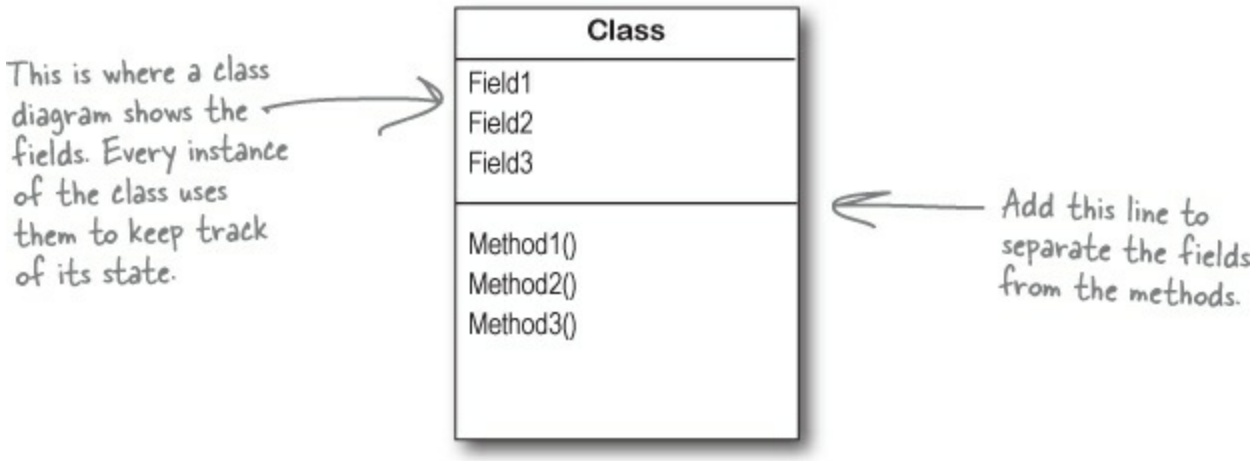
You change the text on a button by setting its `Text` property in the IDE. When you do, the IDE adds code like this to the designer:

```
button1.Text = "Text for the button";
```

Now you know that `button1` is an instance of the `Button` class. What that code does is modify a **field** for the `button1` instance. You can add fields to a class diagram — just draw a horizontal line in the middle of it. Fields go above the line, methods go underneath it.

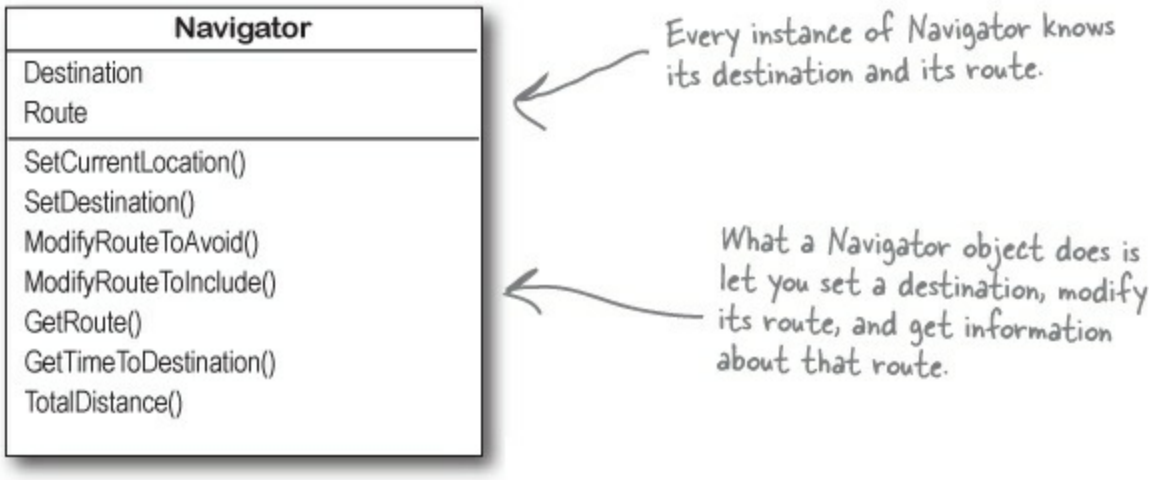
NOTE

Technically, it's setting a property. A property is very similar to a field — but we'll get into all that a little later on.



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

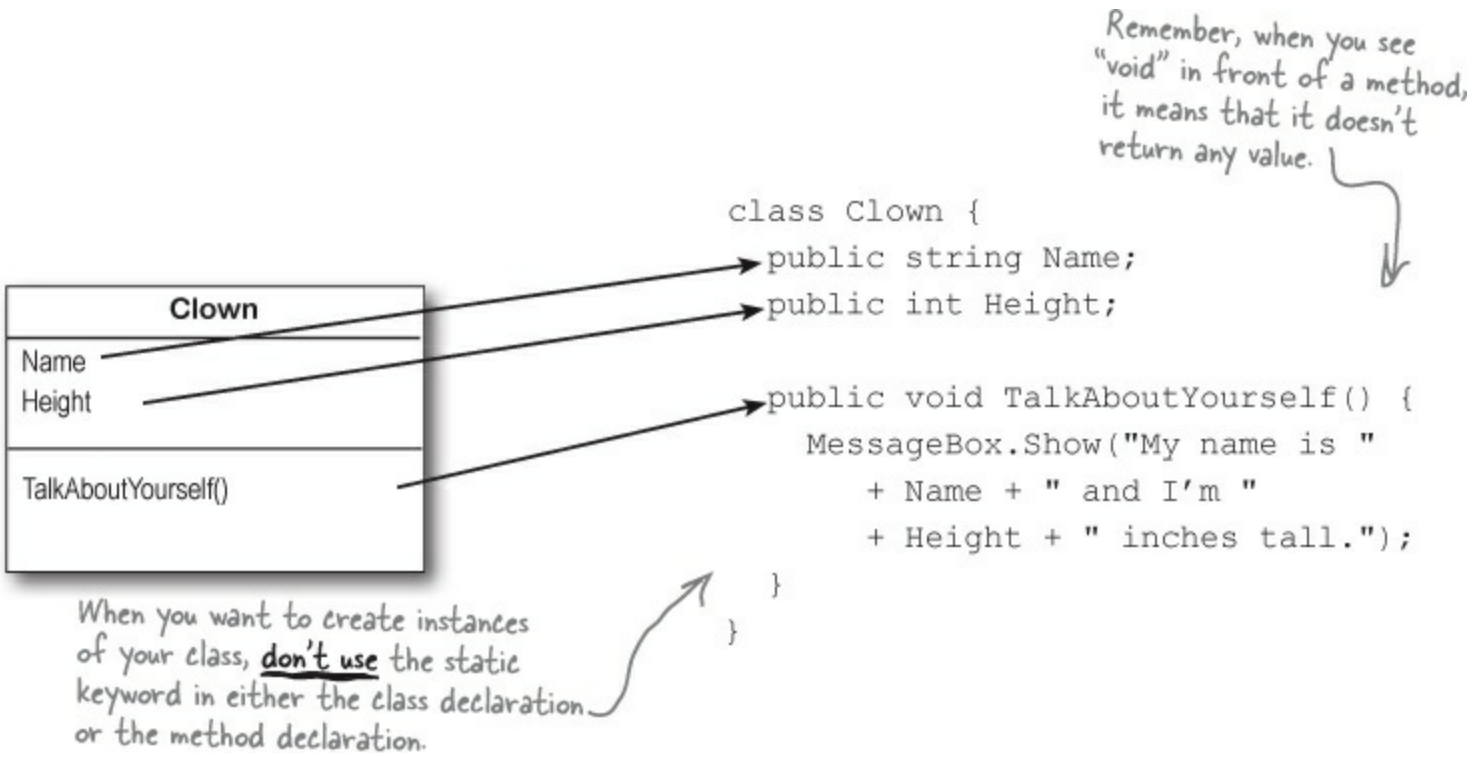
When Mike created three instances of `Navigator` classes, his program created three objects. Each of those objects was used to keep track of a different route. When the program created the `navigator2` instance and called its `SetDestination()` method, it set the destination for that one instance. But it didn't affect the `navigator1` instance or the `navigator3` instance.



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

Let's create some instances!

It's easy to add fields to your class. Just declare variables outside of any methods. Now every instance gets its own copy of those variables.



SHARPEN YOUR PENCIL

Write down the contents of each message box that will be displayed after the statement next to it is executed.

NOTE

Remember, the *= operator tells C# to take whatever's on the left of the operator and multiply it by whatever's on the right.

`Clown oneClown = new Clown();`
`oneClown.Name = "Boffo";`
`oneClown.Height = 14;`
`oneClown.TalkAboutYourself();`
	"My name is _____ and I'm _____ inches tall."

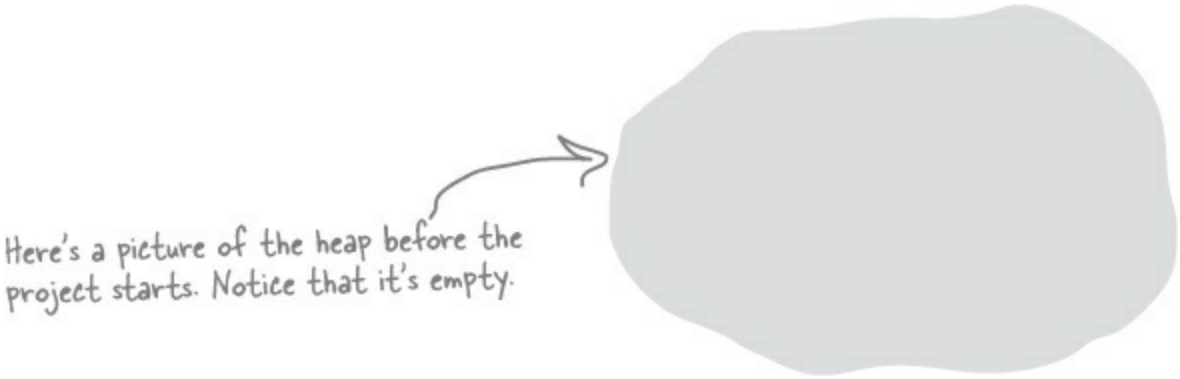
`Clown anotherClown = new Clown();`
`anotherClown.Name = "Biff";`
`anotherClown.Height = 16;`
`anotherClown.TalkAboutYourself();`
	"My name is _____ and I'm _____ inches tall."

`Clown clown3 = new Clown();`
`clown3.Name = anotherClown.Name;`
`clown3.Height = oneClown.Height - 3;`
`clown3.TalkAboutYourself();`
	"My name is _____ and I'm _____ inches tall."

`anotherClown.Height *= 2;` `anotherClown.TalkAboutYourself();`	"My name is _____ and I'm _____ inches tall."

Thanks for the memory

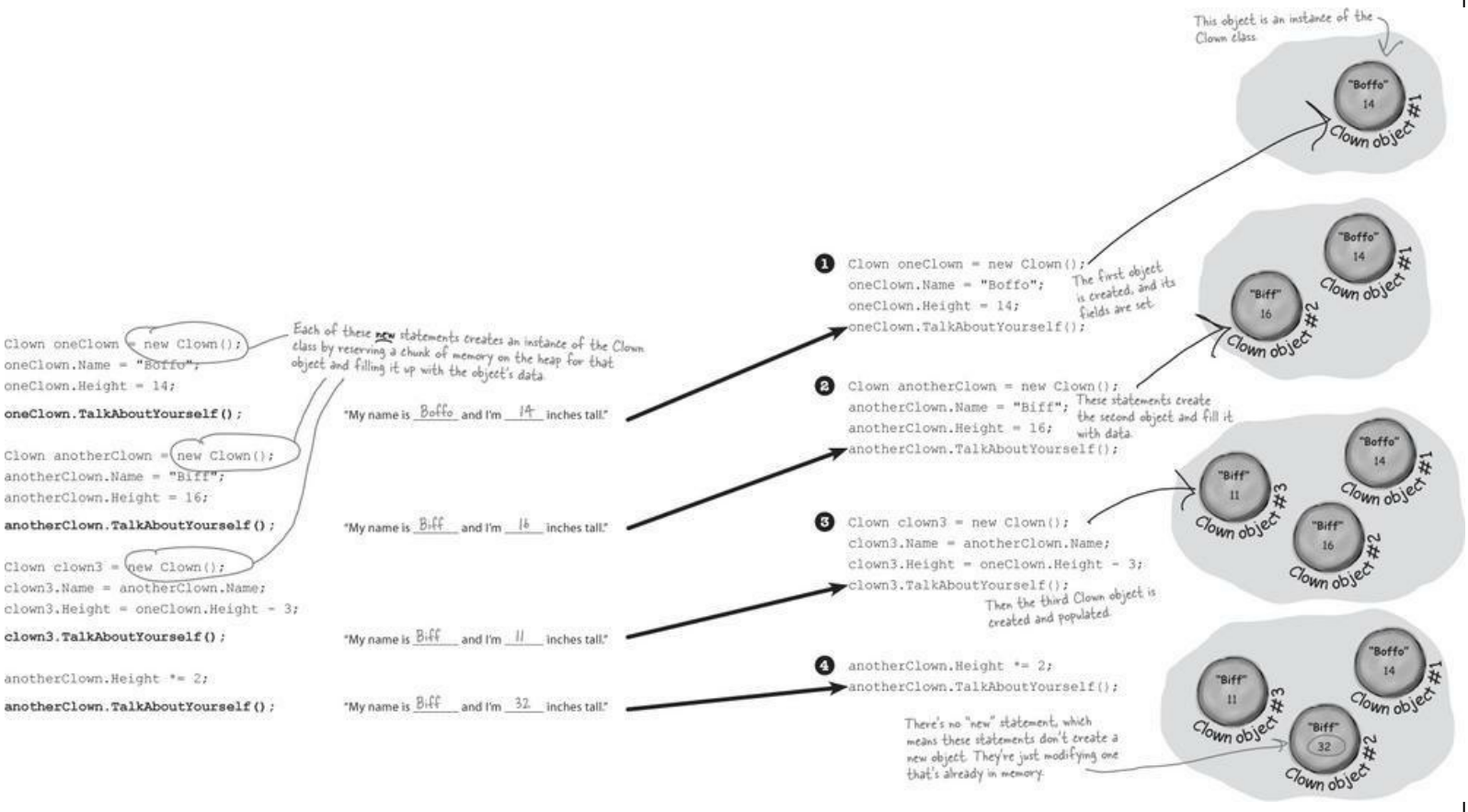
When your program creates an object, it lives in a part of the computer's memory called the **heap**. When your code creates an object with a `new` statement, C# immediately reserves space in the heap so it can store the data for that object.



SHARPEN YOUR PENCIL SOLUTION

Write down the contents of each message box that will be displayed after the statement next to it is executed.

Let's take a closer look at what happened here



When your program creates a new object, it gets added to the heap.

What's on your program's mind

Here's how your program creates a new instance of the `Clown` class:

```
Clown myInstance = new Clown();
```

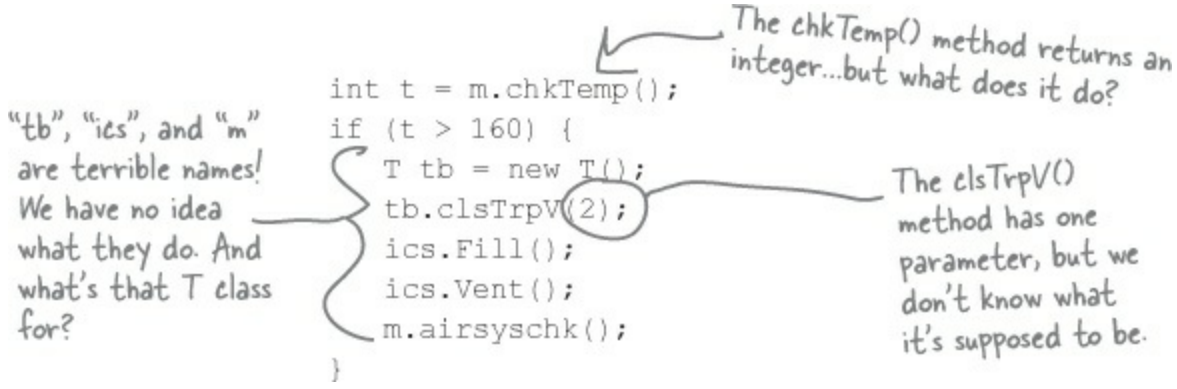
That's actually two statements combined into one. The first statement declares a variable of type `Clown` (`Clown myInstance;`). The second statement creates a new object and assigns it to the variable that was just created (`myInstance = new Clown();`). Here's what the heap looks like after each of these statements:

You can use class and method names to make your code intuitive

When you put code in a method, you're making a choice about how to structure your program. Do you use one method? Do you split it into more than one? Or do you even need a 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.

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.

- ❶ Here's a nice, compact chunk of code. It's from a control program that runs a machine that makes candy bars.



Take a second and look at that code. Can you figure out what it does?

- ❷ 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.

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 can look up the page in the specification manual that the programmer followed.

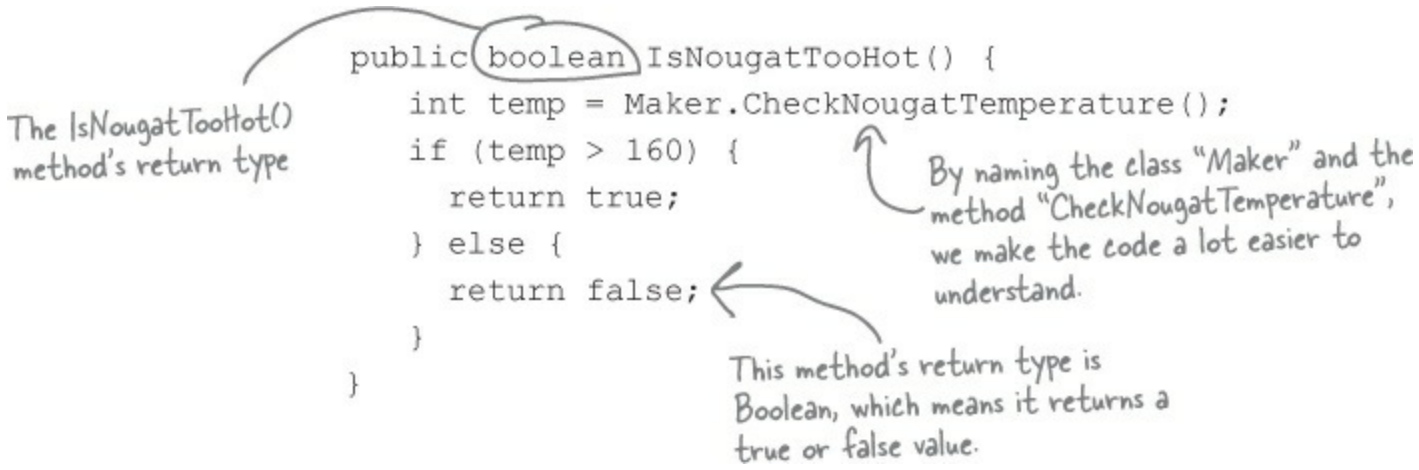
General Electronics Type 5 Candy Bar Maker Specification Manual

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**.

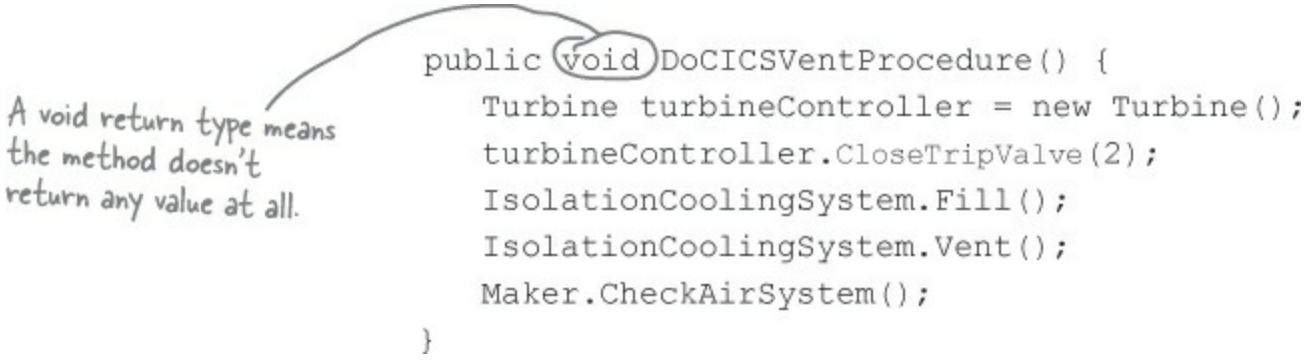
- Close the trip throttle valve on turbine #2.
- Fill the isolation cooling system with a solid stream of water.
- Vent the water.
- Verify that there is no evidence of air in the system.

- ❸ 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. 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.



- ❹ What does the specification 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):



- ❺ Now the code's 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**:

```
if (IsNougatTooHot() == true) {
   DoCICSVentProcedure();
}
```

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!

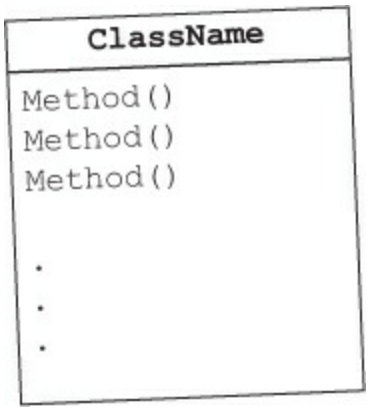
Give your classes a natural structure

Take a second and remind yourself why you want to make your methods intuitive: **because every program solves a problem or has a purpose.** It might not be a business problem — sometimes a program's purpose (like FlashyThing) is just to be cool or fun! But no matter what your program does, the more you can make your code resemble the problem you're trying to solve, the easier your program will be to write (and read, and repair, and maintain...).

USE CLASS DIAGRAMS TO PLAN OUT YOUR CLASSES

A class diagram is a simple way to draw your classes out on paper. It's a really valuable tool for designing your code BEFORE you start writing it.

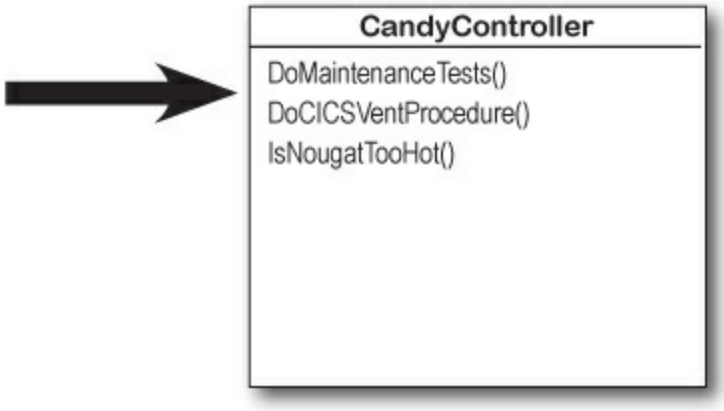
Write the name of the class at the top of the diagram. Then write each method in the box at the bottom. Now you can see all of the parts of the class at a glance!



Let's build a class diagram

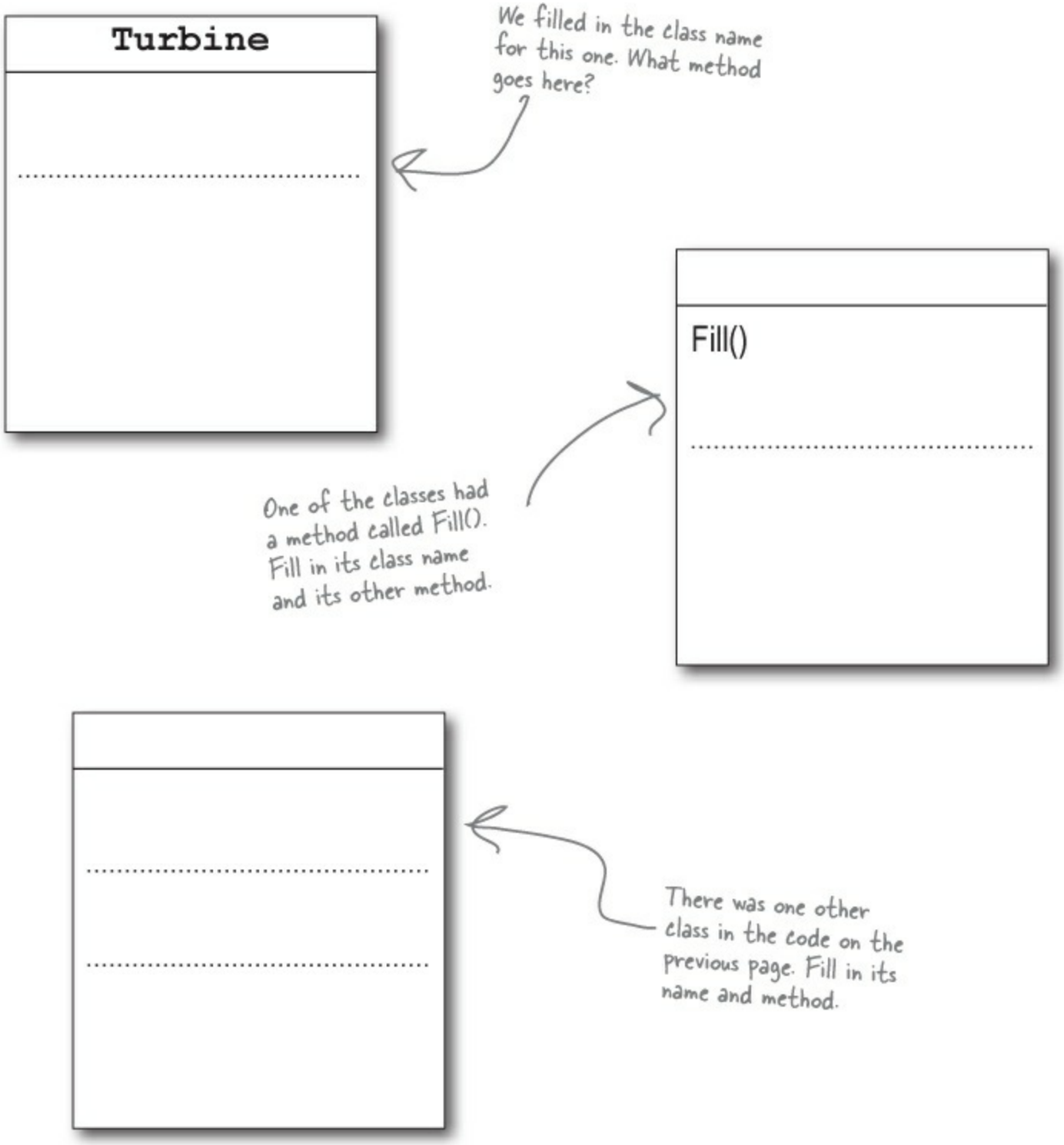
Take another look at the `if` statement in #5 on the previous page. You already know that statements always live inside methods, which always live inside classes, right? In this case, that `if` statement was in a method called `DoMaintenanceTests()`, which is part of the `CandyController` class. Now take a look at the code and the class diagram. See how they relate to each other?

```
class CandyController {

  public void DoMaintenanceTests() {
    ...
    if (IsNougatTooHot() == true) {
       DoCICSVentProcedure();
    }
    ...
  }

  public void DoCICSVentProcedure() ...

  public boolean IsNougatTooHot() ...

}
```



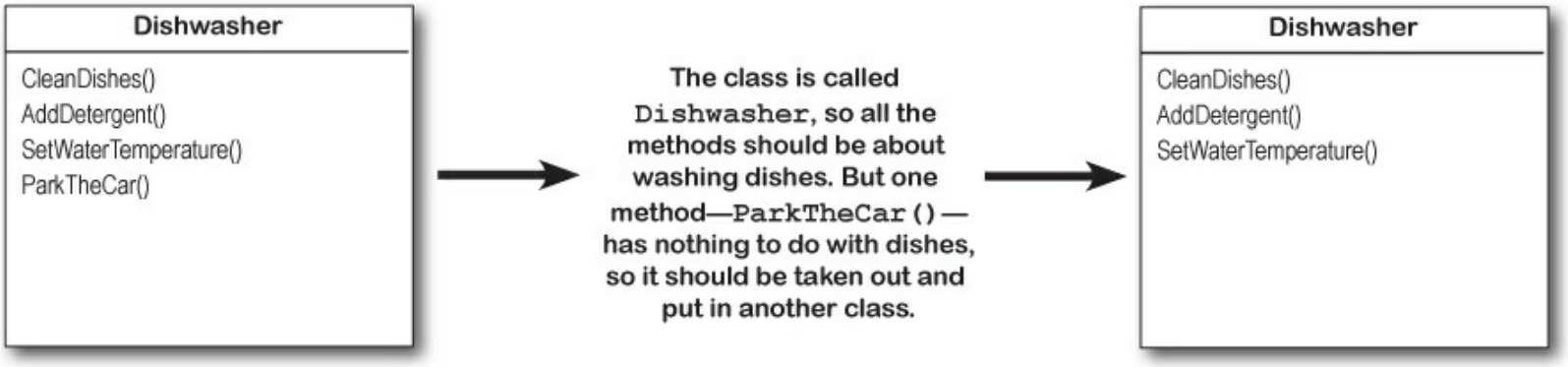
SHARPEN YOUR PENCIL

The code for the candy control system we built on the previous page called three other classes. Flip back and look through the code, and fill in their class diagrams.



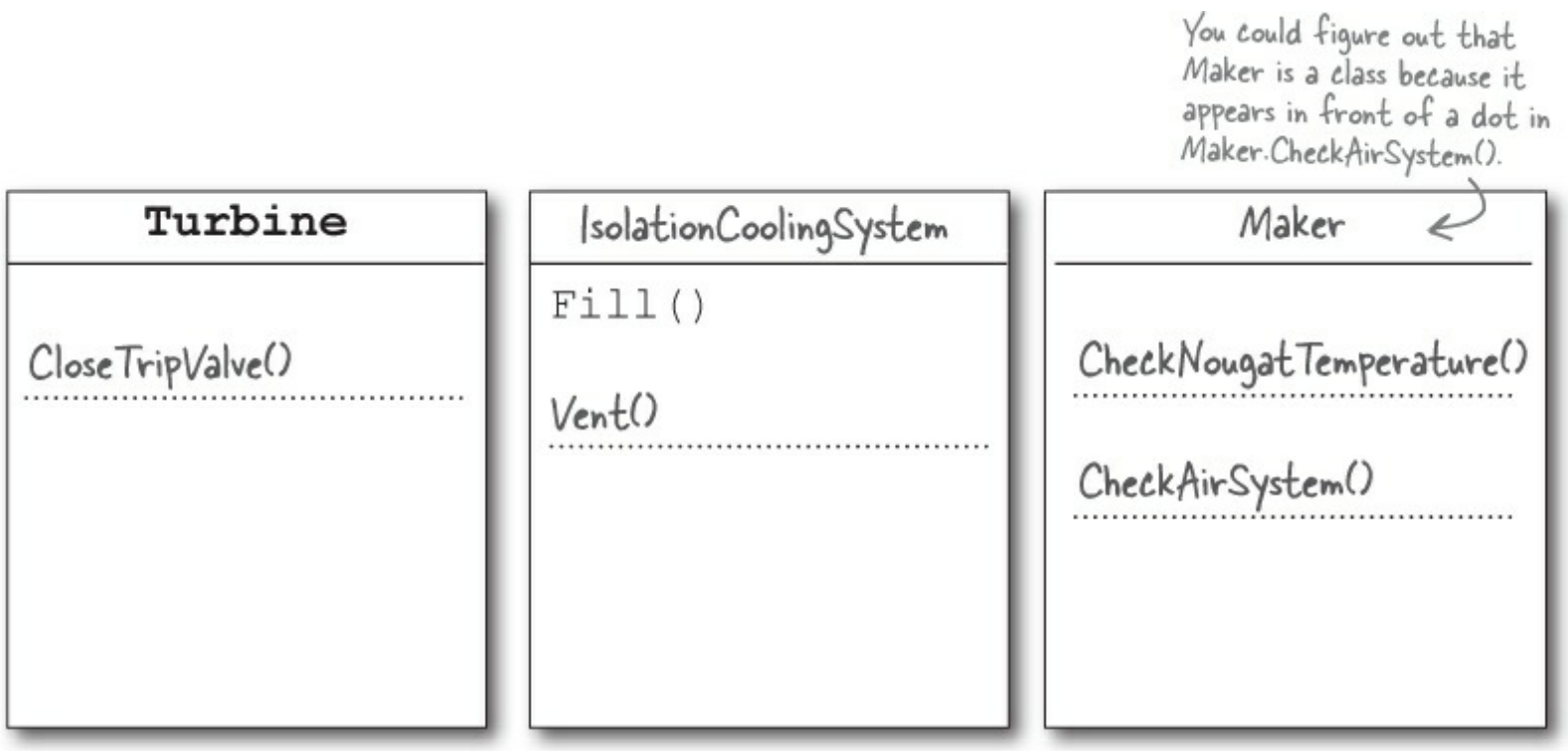
Class diagrams help you organize your classes so they make sense

Writing out class diagrams makes it a lot easier to spot potential problems in your classes **before** you write code. Thinking about your classes from a high level before you get into the details can help you come up with a class structure that will make sure your code addresses the problems it solves. It lets you step back and make sure that you're not planning on writing unnecessary or poorly structured classes or methods, and that the ones you do write will be intuitive and easy to use.



SHARPEN YOUR PENCIL SOLUTION

The code for the candy control system we built on the previous page called three other classes. Flip back and look through the code, and fill in their class diagrams.



SHARPEN YOUR PENCIL

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

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

_____

_____

_____

_____

These two classes are part of a system that a pizza parlor uses to track the pizzas that are out for delivery.

_____

_____

_____

_____

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

_____

_____

_____

_____

SHARPEN YOUR PENCIL SOLUTION

Here's how we corrected the classes. We show just one possible way to fix the problems — but there are plenty of other ways you could design these classes depending on how they'll be used.

This class is part of the candy manufacturing system from earlier. The class name doesn't describe what the class does. A programmer who sees a line of code that calls Class23.Go() will have no idea what that line does. We'd also rename the method to something that's more descriptive — we chose MakeTheCandy(), but it could be anything.	
These two classes are part of a system that a pizza parlor uses to track the pizzas that are out for delivery. It looks like the DeliveryGuy class and the DeliveryGirl class both do the same thing — they track a delivery person who's out delivering pizzas to customers. A better design would replace them with a single class that adds a field for gender.	
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.	

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 the output listed.

```
public partial class Form1 : Form {
  public Form1() {
    InitializeComponent();
  }
  private void button1_Click(object sender, EventArgs e) {
    string result = "";
    Echo e1 = new Echo();

    ____________________________
    int x = 0;
    while ( ____________ ) {
      result = result + e1.Hello() + "\n";

      ____________________________
      if ( _____________ ) {
        e2.Count = e2.Count + 1;
      }
      if ( _____________ ) {
        e2.Count = e2.Count + e1.Count;
      }
      x = x + 1;
    }
    MessageBox.Show(result + "Count: " + e2.Count);
  }
}
class _____________ {
  public int __________ = 0;
    public string ____________ {
```

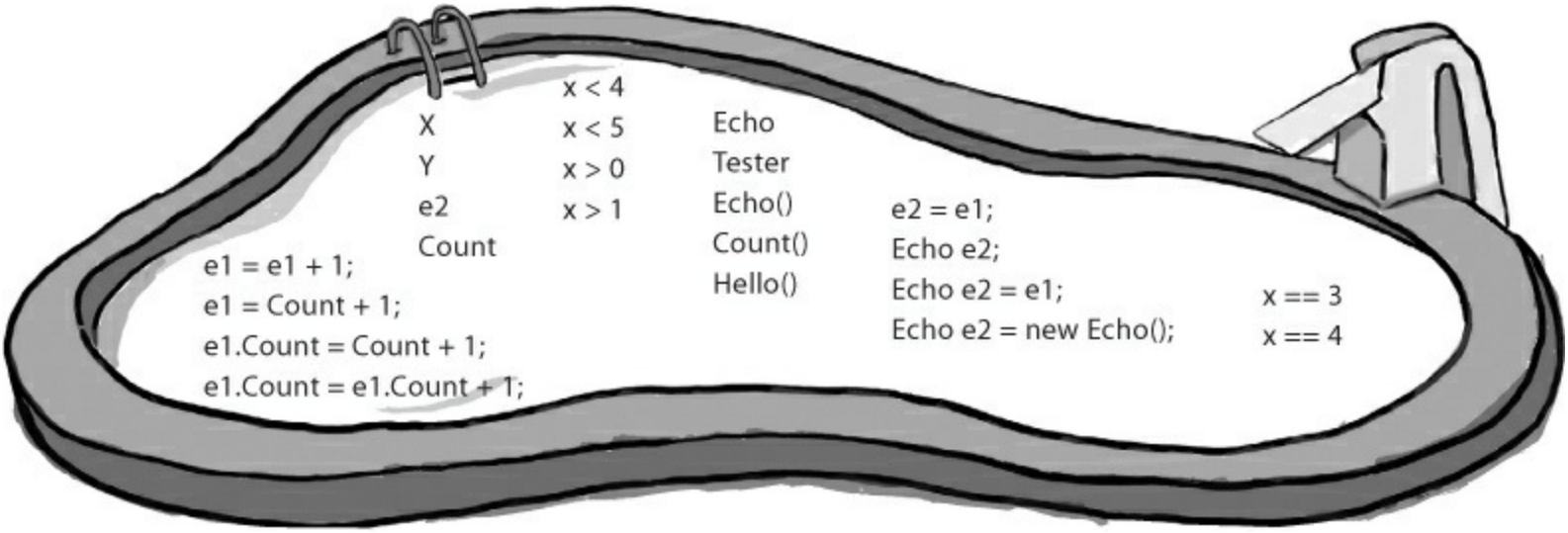
```
        return "helloooo...";
    }
}
```

Output

Bonus Question!

If the last line of output was **24** instead of **10,** how would you complete the puzzle? You can do it by changing just one statement.

Note: each snippet from the pool can be used more than once!



Answers in Pool Puzzle Solution.

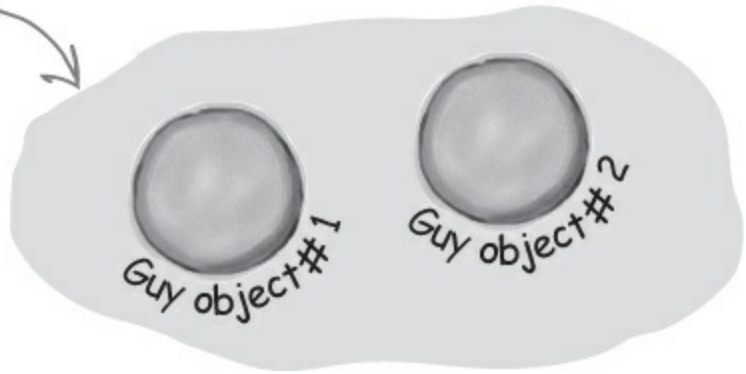
There are two possible solutions to this puzzle. Can you find them both?

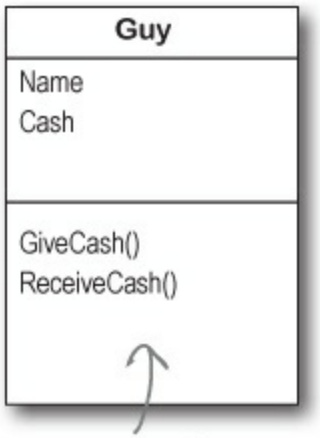
Build a class to work with some guys

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.

- **❶ We'll create a Guy class and add two instances of it to a form.**
  The form will have two fields, one called `joe` (to keep track of the first object), and the other called `bob` (to keep track of the second object).

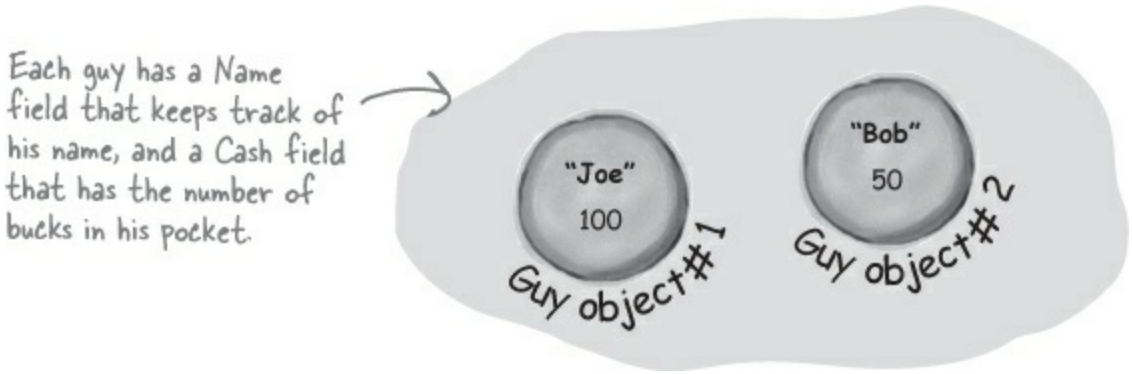
The new statements that create the two instances live in the code that gets run as soon as the form is created. Here's what the heap looks like after the form is loaded.



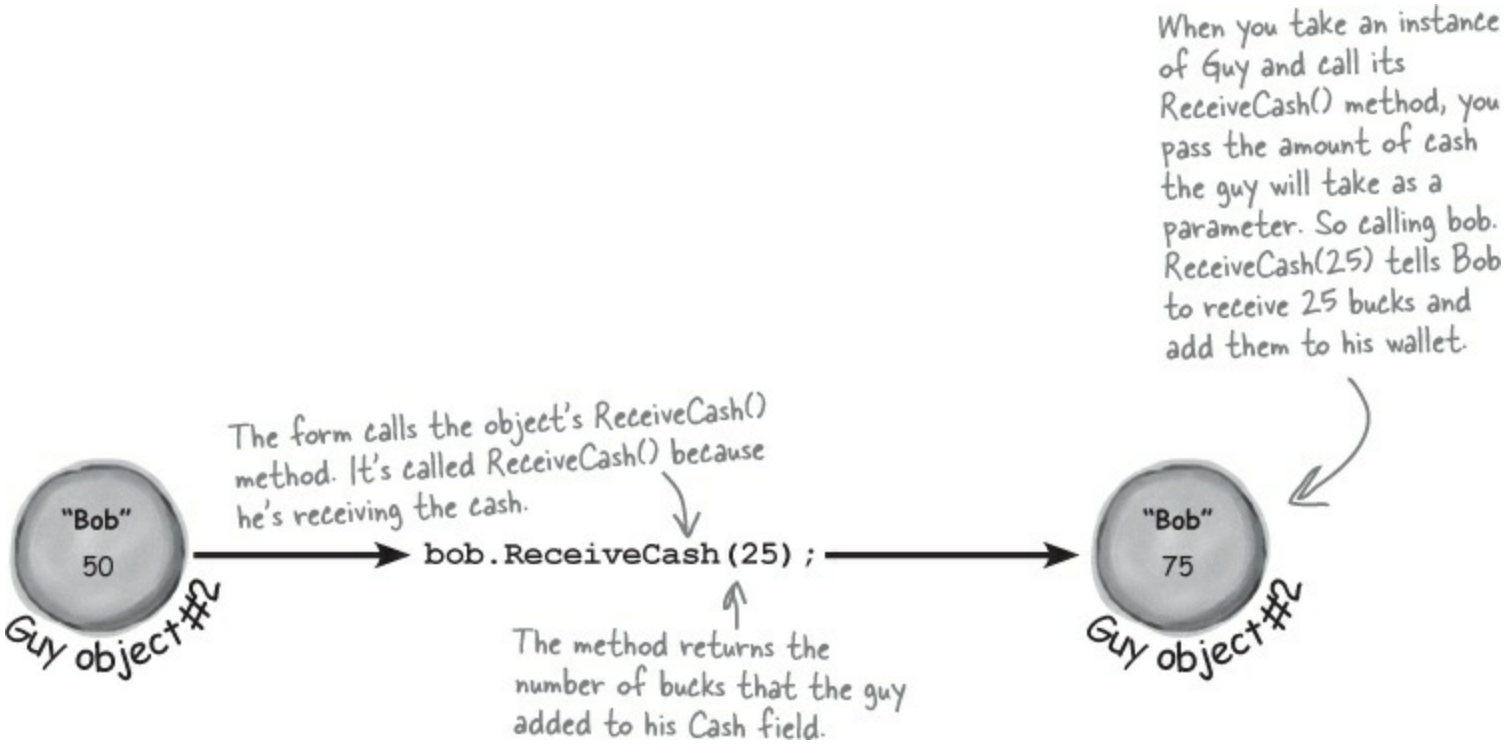


We chose names for the methods that make sense. You call a Guy object's GiveCash() method to tell him to give up some of his cash, and his ReceiveCash() method when you want him to take some cash back. We could have called them GiveCashToSomeone() and ReceiveCashFromSomeone(), but that would have been very long!

- **❷ 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.



- **❸ We'll give cash to the guys and take cash from them.**
  We'll use each guy's `ReceiveCash()` method to increase a guy's cash, and we'll use his `GiveCash()` method to reduce it.



Create a project for your guys

Create a new Windows Forms Application project (because we'll be using a form). Then use the Solution Explorer to add a new class to it called `Guy`. Make sure to add "`using System.Windows.Forms;`" to the top of the `Guy` class file. Then fill in the `Guy` class. Here's the code for it:

DO THIS!

The Guy class has two fields. The Name field is a string, and it'll contain the guy's name ("Joe"). And the Cash field is an int, which will keep track of how many bucks are in his pocket.

```
class Guy {
    public string Name;
    public int Cash;

    public int GiveCash(int amount) {
        if (amount <= Cash && amount > 0) {
            Cash -= amount;
            return amount;
        } else {
            MessageBox.Show(
             "I don't have enough cash to give you " + amount,
             Name + " says...");
            return 0;
        }
    }
```

The GiveCash() method has one parameter called amount that you'll use to tell the guy how much cash to give you.

He uses an if statement to check whether he has enough cash—if he does, he takes it out of his pocket and returns it as the return value.

The guy makes sure that you're asking him for a positive amount of cash—otherwise, he'd add to his cash instead of taking away from it.

If the guy doesn't have enough cash, he'll tell you so with a message box, and then he'll make GiveCash() return 0.

```
    public int ReceiveCash(int amount) {
        if (amount > 0) {
            Cash += amount;
            return amount;
        } else {
            MessageBox.Show(amount + " isn't an amount I'll take",
                  Name + " says...");
            return 0;
        }
    }
```

The ReceiveCash() method works just like the GiveCash() method. It's passed an amount as a parameter, checks to make sure that amount is greater than zero, and then adds it to his cash.

If the amount was positive, then the ReceiveCash() method returns the amount added. If it was zero or negative, the guy shows a message box and then returns 0.

Be careful with your curly brackets. It's easy to have the wrong number—make sure that every opening bracket has a matching closing bracket. When they're all balanced, the IDE will **automatically indent** them for you when you type the last closing bracket.

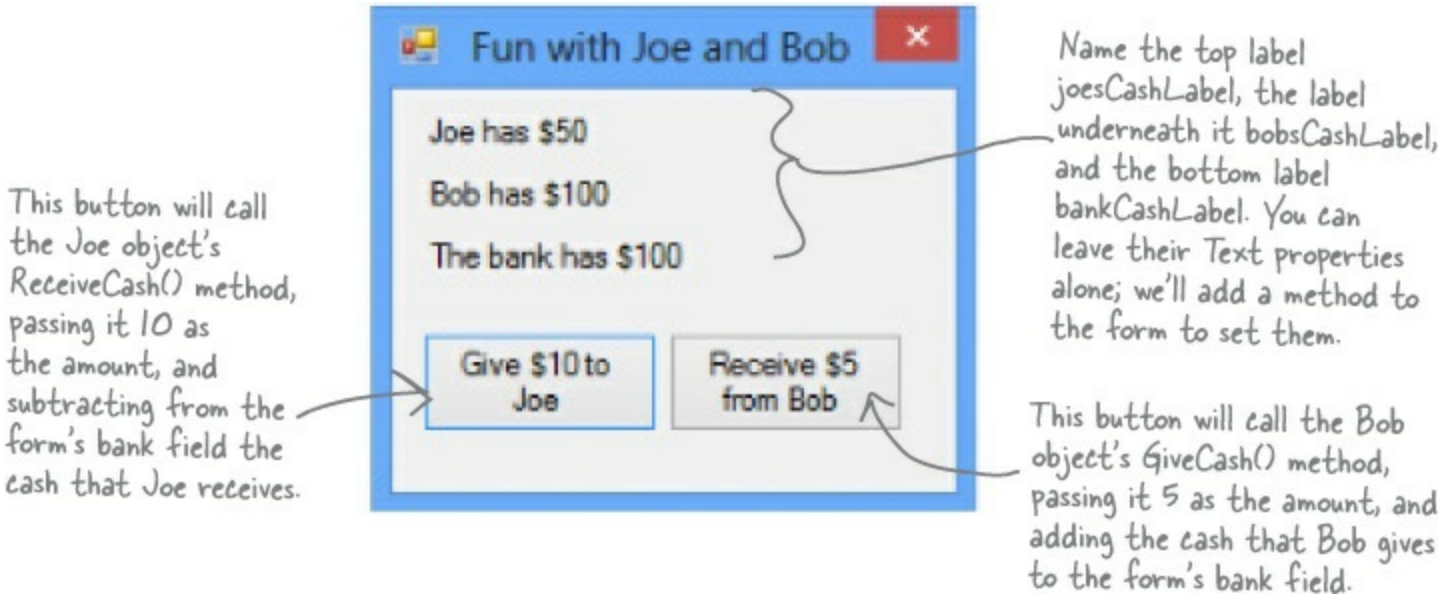
What happens if you pass a negative amount to a Guy object's ReceiveCash() or GiveCash() method?

Build a form to interact with the guys

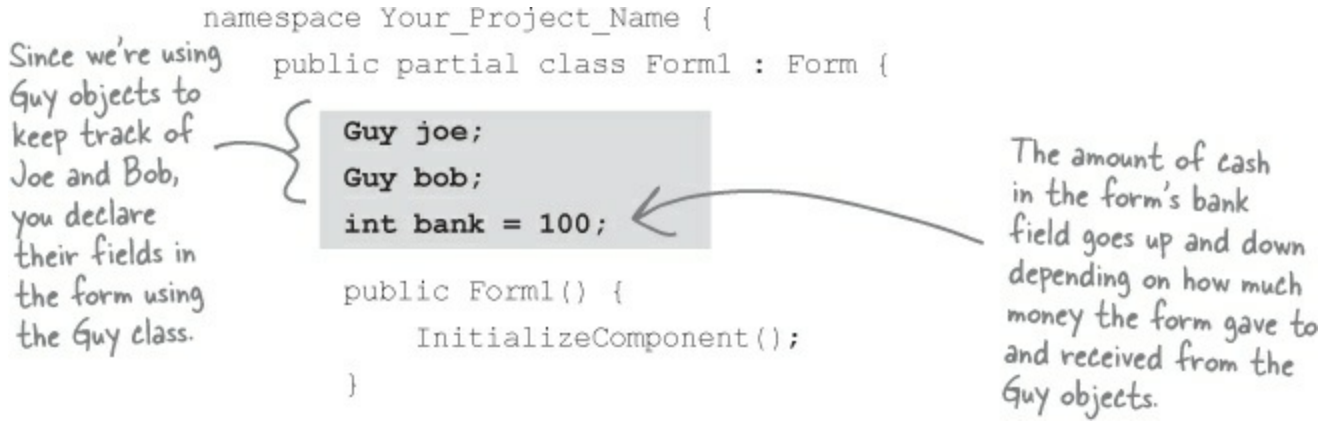
The `Guy` class is great, but it's just a start. Now put together a form that uses two instances of the `Guy` class. It's got labels that show you their names and how much cash they have, and buttons to give and take cash from them. They have to get their money from *somewhere* before they can lend it to each other, so we'll also need to add a bank.

BUILD THIS!

- **❶ Add two buttons and three labels to your form.**
  The top two labels show how much cash each guy has. We'll also add a field called `bank` to the form — the third label shows how much cash is in it. We're going to have you name some of the labels that you drag onto the forms. You can do that by **clicking on each label** that you want to name and **changing its "(Name)" row** in the Properties window. That'll make your code a lot easier to read, because you'll be able to use "joesCashLabel" and "bobsCashLabel" instead of "label1" and "label2".



- **❷ Add fields to your form.**
  Your form will need to keep track of the two guys, so you'll need a field for each of them. Call them `joe` and `bob`. Then add a field to the form called `bank` to keep track of how much money the form has to give to and receive from the guys.



- **❸ Add a method to the form to update the labels.**
  The labels on the lefthand side of the form show how much cash each guy has and how much is in the `bank` field. So add the `UpdateForm()` method to keep them up to date — **make sure the return type is `void`** to tell C# that the method doesn't return a value. Type this method into the form right underneath where you added the `bank` field:

```
public void UpdateForm() {
    joesCashLabel.Text = joe.Name + " has $" + joe.Cash;
    bobsCashLabel.Text = bob.Name + " has $" + bob.Cash;
    bankCashLabel.Text = "The bank has $" + bank;
}
```

Notice how the labels are updated using the Guy objects' Name and Cash fields.

This new method is simple. It just updates the three labels by setting their Text properties. You'll have each button call it to keep the labels up to date.

- **❹ Double-click on each button and add the code to interact with the objects.**
  Make sure the lefthand button is called `button1`, and the righthand button is called `button2`. Then double-click each of the buttons — when you do, the IDE will add two methods called `button1_Click()` and `button2_Click()` to the form. Add this code to each of them:

You already know that you can choose names for controls. Are button1 and button2 really the best names we can find? What names would *you* choose for these buttons?

```
private void button1_Click(object sender, EventArgs e) {
    if (bank >= 10) {
        bank -= joe.ReceiveCash(10);
        UpdateForm();
    } else {
        MessageBox.Show("The bank is out of money.");
    }
}
```

When the user clicks the "Give $10 to Joe" button, the form calls the Joe object's ReceiveCash() method—but only if the bank has enough money.

The bank needs at least $10 to give to Joe. If there's not enough, it'll pop up this message box.

```
private void button2_Click(object sender, EventArgs e) {
    bank += bob.GiveCash(5);
    UpdateForm();
}
```

The "Receive $5 from Bob" button doesn't need to check how much is in the bank, because it'll just add whatever Bob gives back.

If Bob's out of money, GiveCash() will return zero.

EXERCISE

5 Start Joe out with $50 and start Bob out with $100.

It's up to you to **figure out how to get Joe and Bob to start out with their `Cash` and `Name` fields set properly.** Put it right underneath `InitializeComponent()` in the form. That's part of that designer-generated method that gets run once, when the form is first initialized. Once you've done that, click both buttons a number of times — make sure that one button takes $10 from the bank and adds it to Joe, and the other takes $5 from Bob and adds it to the bank.

```
public Form1() {
    InitializeComponent();
    // Initialize joe and bob here!
}
```

Add the lines of code here to create the two objects and set their Name and Cash fields.

EXERCISE SOLUTION

It's up to you to **figure out how to get Joe and Bob to start out with their `Cash` and `Name` fields set properly.** Put it right underneath `InitializeComponent()` in the form.

```
public Form1() {
    InitializeComponent();

    bob = new Guy();
    bob.Name = "Bob";
    bob.Cash = 100;

    joe = new Guy();
    joe.Name = "Joe";
    joe.Cash = 50;

    UpdateForm();
}
```

Here's where we set up the first instance of Guy. The first line creates the object, and the next two set its fields.

Then we do the same for the second instance of the Guy class.

Make sure you call UpdateForm() so the labels look right when the form first pops up.

NOTE

Make sure you save the project now — we'll come back to it in a few pages.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Why doesn't the solution start with "`Guy bob = new Guy()`"? Why did you leave off the first "`Guy`"?**

A: **A:** Because you already declared the `bob` field at the top of the form. Remember how the statement "`int i = 5;`" is the same as the two statements "`int i`" and "`i = 5;`"? This is the same thing. You could try to declare the `bob` field in one line like this: "`Guy bob = new Guy();`". But you already have the first part of that statement ("`Guy bob;`") at the top of your form. So you only need the second half of the line, the part that sets the `bob` field to create a new instance of `Guy()`.

Q: **Q: OK, so then why not get rid of the "`Guy bob;`" line at the top of the form?**

A: **A:** Then a variable called `bob` will only exist inside that special "`public Form1()`" method. When you declare a variable inside a method, it's only valid inside the method — you can't access it from any other method. But when you declare it outside of your method but inside the form or a class that you added, then you've added a field accessible from ***any other method*** inside the form.

Q: **Q: What happens if I don't leave off that first "`Guy`"? What if it's `Guy bob = new Guy()` instead of `bob = new Guy()`?**

A: **A:** You'll run into problems — your form won't work, because it won't ever set the form's `bob` variable. If you have this code at the top of your form:

```
public partial class Form1 : Form {
    Guy bob;
```

and then you have this code later on, inside a method:

```
Guy bob = new Guy();
```

then you've declared ***two*** variables. It's a little confusing, because they both have the same name. But one of them is valid throughout the entire form, and the other one — the new one you added — is only valid inside the method. The next line (`bob.Name = "Bob";`) only updates that ***local*** variable, and doesn't touch the one in the form. So when you try to run your code, it'll give you a nasty error message ("NullReferenceException not handled"), which just means you tried to use an object before you created it with `new`.

There's an easier way to initialize objects

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.

Object initializers save you time and make your code more compact and easier to read...and the IDE helps you write them.

- ❶ Here's the original code that you wrote to initialize Joe's `Guy` object.

```
joe = new Guy();
joe.Name = "Joe";
joe.Cash = 50;
```

- ❷ Delete the second two lines and the semicolon after "`Guy()`," and add a right curly bracket.

```
joe = new Guy() {
```

- ❸ Press space. As soon as you do, the IDE pops up an IntelliSense window that shows you all of the fields that you're able to initialize.

```
joe = new Guy() {
```

- ❹ Press Tab to tell it to add the `Cash` field. Then set it equal to 50.

```
joe = new Guy() { Cash = 50
```

- ❺ Type in a comma. As soon as you do, the other field shows up.

```
joe = new Guy() { Cash = 50,
```

- ❻ Finish the object initializer. Now you've saved yourself two lines of code!

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

This new declaration does exactly the same thing as the three lines of code you wrote originally. It's just shorter and easier to read.

You used an object initializer in your "Save the Humans" game. Flip back and see if you can spot it!

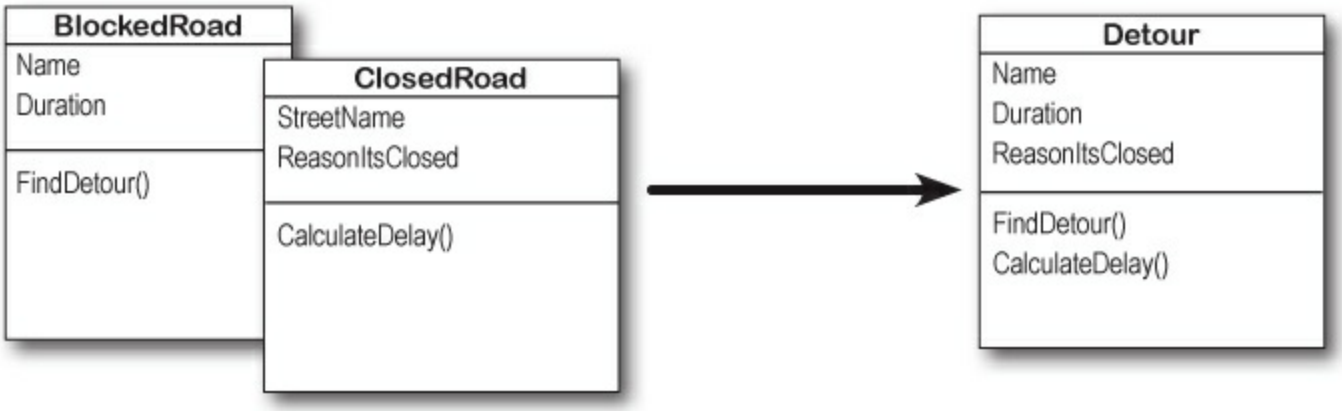
A few ideas for designing intuitive classes

- **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.

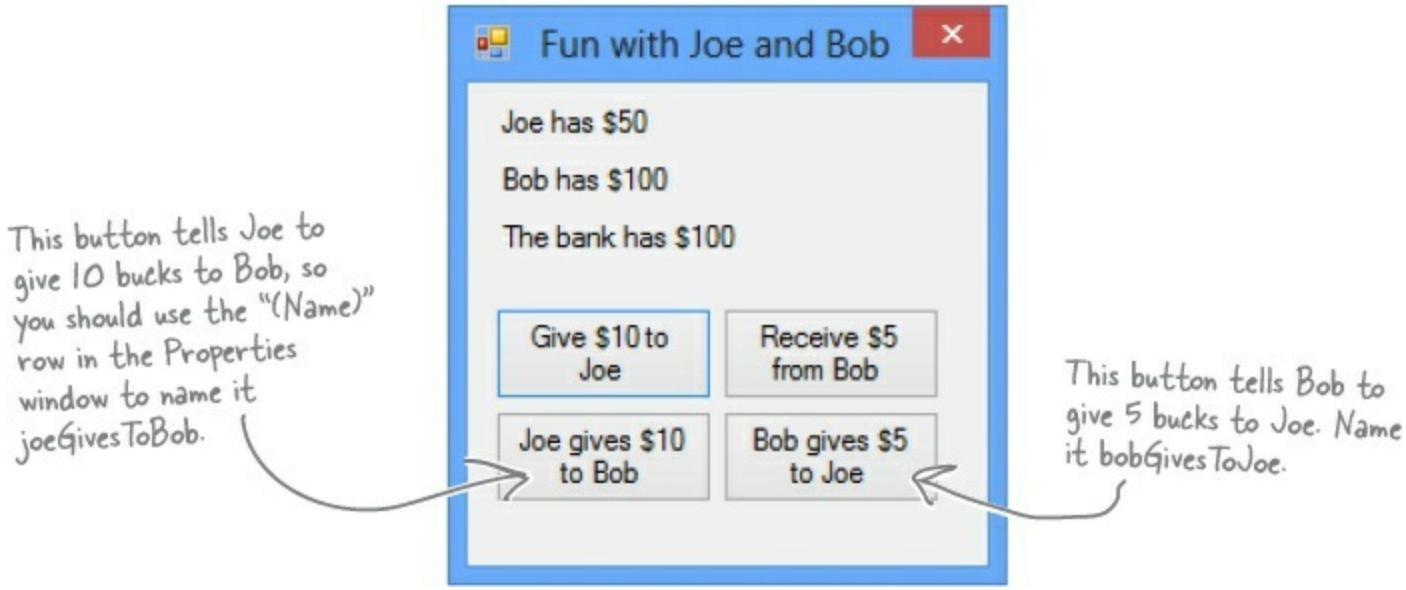

BlockedRoad
Name
Duration
FindDetour()
ClosedRoad
StreetName
ReasonItsClosed
CalculateDelay()
Detour
Name
Duration
ReasonItsClosed
FindDetour()
CalculateDelay()

EXERCISE

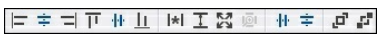
Add buttons to the "Fun with Joe and Bob" program to make the guys give each other cash.

- **❶ USE AN OBJECT INITIALIZER TO INITIALIZE BOB'S INSTANCE OF GUY.**
  You've already done it with Joe. Now make Bob's instance work with an object initializer too.
- **❷ ADD TWO MORE BUTTONS TO YOUR FORM.**
  The first button tells Joe to give 10 bucks to Bob, and the second tells Bob to give 5 bucks back to Joe. **Before you double-click on the button,** go to the Properties window and change each button's name using the "(Name)" row — it's **at the top** of the list of properties. Name the first button **`joeGivesToBob`**, and the second one **`bobGivesToJoe`**.

> **NOTE**
>
> If you already clicked the button, just delete it, add it back to your form, and rename it. Then delete the old button3_Click() method that the IDE added before, and use the new method it adds now.



- **❸ MAKE THE BUTTONS WORK.**
  Double-click on the `joeGivesToBob` button in the designer. The IDE will add a method to the form called `joeGivesToBob_Click()` that gets run any time the button's clicked. Fill in that method to make Joe give 10 bucks to Bob. Then double-click on the other button and fill in the new `bobGivesToJoe_Click()` method that the IDE creates so that Bob gives 5 bucks to Joe. Make sure the form updates itself after the cash changes hands.

Here's a tip for designing your forms. You can use these buttons on the IDE's toolbar in the form designer to align controls, make them equal sizes, space them evenly, and bring them to the front or back.


EXERCISE SOLUTION

Add buttons to the "Fun with Joe and Bob" program to make the guys give each other cash.

```
public partial class Form1 : Form {
    Guy joe;
    Guy bob;
    int bank = 100;

    public Form1() {
        InitializeComponent();
        bob = new Guy() { Cash = 100, Name = "Bob" };
        joe = new Guy() { Cash = 50, Name = "Joe" };
        UpdateForm();
    }

    public void UpdateForm() {
        joesCashLabel.Text = joe.Name + " has $" + joe.Cash;
        bobsCashLabel.Text = bob.Name + " has $" + bob.Cash;
        bankCashLabel.Text = "The bank has $" + bank;
    }

    private void button1_Click(object sender, EventArgs e) {
        if (bank >= 10) {
            bank -= joe.ReceiveCash(10);
            UpdateForm();
        } else {
            MessageBox.Show("The bank is out of money.");
        }
    }

    private void button2_Click(object sender, EventArgs e) {
        bank += bob.GiveCash(5);
        UpdateForm();
    }

    private void joeGivesToBob_Click(object sender, EventArgs e) {
        bob.ReceiveCash(joe.GiveCash(10));
        UpdateForm();
    }

    private void bobGivesToJoe_Click(object sender, EventArgs e) {
        joe.ReceiveCash(bob.GiveCash(5));
        UpdateForm();
    }

}
```

Here are the object initializers for the two instances of the Guy class. Bob gets initialized with 100 bucks and his name.

To make Joe give cash to Bob, we call Joe's GiveCash() method and send its results into Bob's ReceiveCash() method.

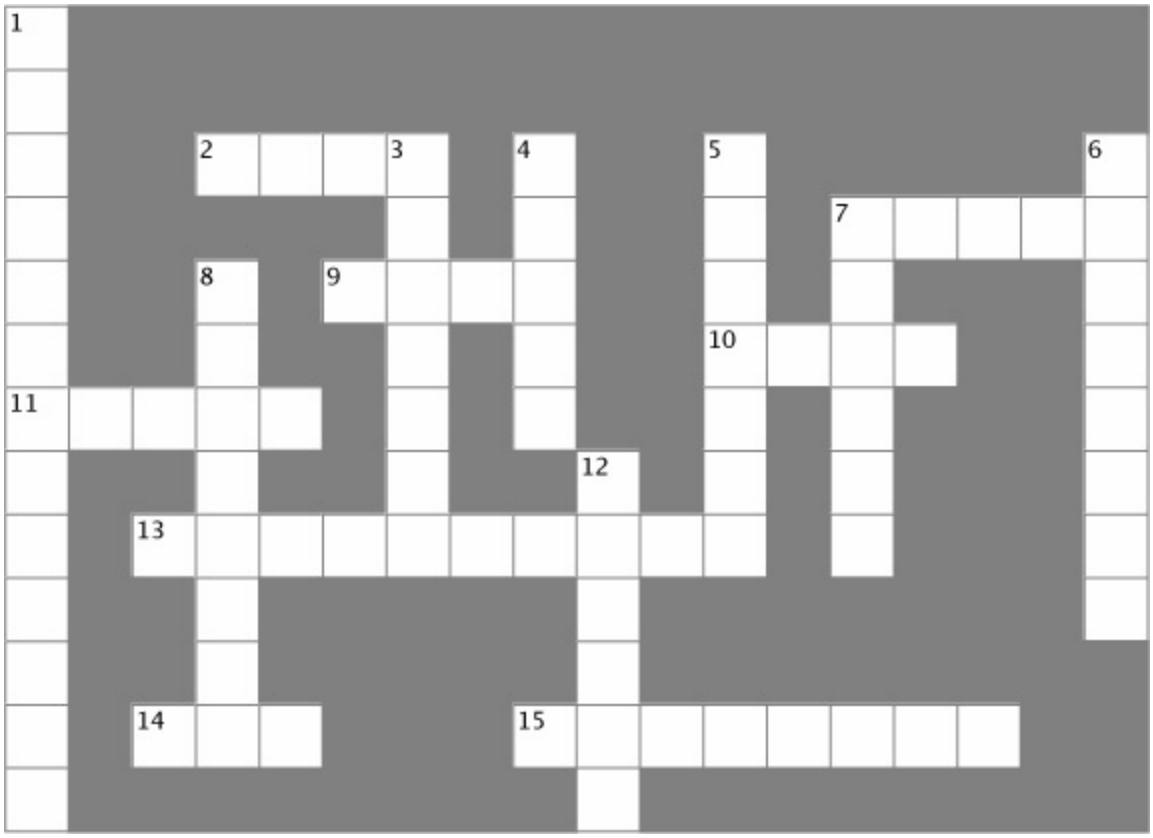
Take a close look at how the Guy methods are being called. The results returned by GiveCash() are pumped right into ReceiveCash() as its parameter.

The trick here is thinking through who's giving the cash and who's receiving it.

Before you go on, take a minute and flip to #2 in the "Leftovers" appendix, because there's some basic syntax that we haven't covered yet. You won't *need* it to move forward, but it's a good idea to see what's there.

OBJECTCROSS

It's time to give your left brain a break, and put that right brain to work: all the words are object-related and from this chapter.



Across	Down
2. If a method's return type is _____, it doesn't return anything 7. An object's fields define its _____ 9. A good method _____ makes it clear what the method does 10. Where objects live 11. What you use to build an object 13. What you use to pass information into a method 14. The statement you use to create an object 15. Used to set an attribute on controls and other classes	1. This form control lets the user choose a number from a range you set 3. It's a great idea to create a class _____ on paper before you start writing code 4. An object uses this to keep track of what it knows 5. These define what an object does 6. An object's methods define its _____ 7. Don't use this keyword in your class declaration if you want to be able to create instances of it 8. An object is an _____ of a class 12. This statement tells a method to immediately exit, and can specify the value that should be passed back to the statement that called the method

POOL PUZZLE SOLUTION

Your ***job*** was to take code snippets from the pool and place them into the blank lines in the code. Your ***goal*** was to make classes that will compile and run and produce the output listed.

```
public partial class Form1 : Form {
  public Form1() {
    InitializeComponent();
  }
  private void button1_Click(object sender, EventArgs e)
    string result = "";
    Echo e1 = new Echo();
    Echo e2 = new Echo();
    int x = 0;
    while ( x < 4 ) {
      result = result + e1.Hello() + "\n";
      e1.Count = e1.Count + 1;
      if ( x ==3 ) {
        e2.count = e2.count + 1;
      }
      if ( x > 0 ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    MessageBox.Show(result + "Count: " + e2.count);
  }
}
class Echo {
  public int Count = 0;
    public string Hello() {
      return "helloooo...";
  }
}
```

That's the correct answer.

And here's the bonus answer!

```
Echo e2 = e1;
```

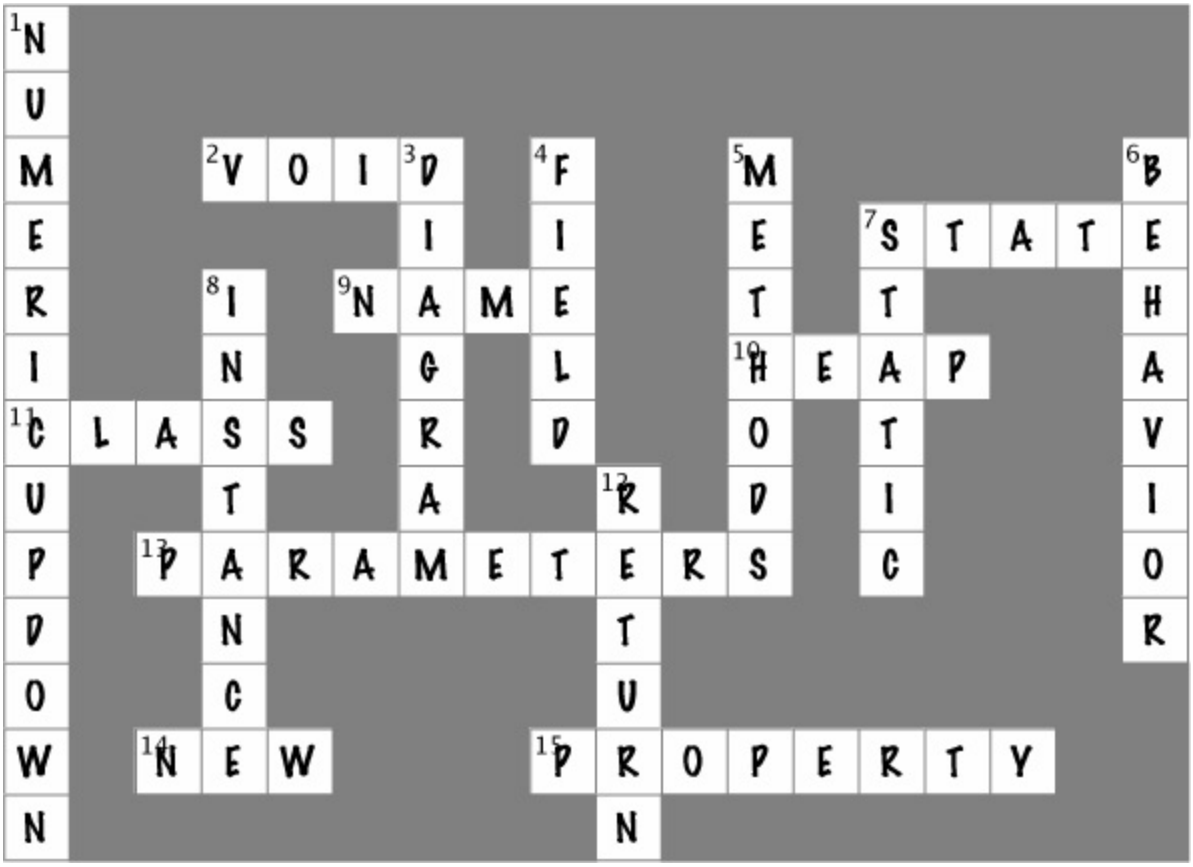
The alternate solution has this in the fourth blank:

```
x == 4
```

and this in the fifth:

```
x < 4
```

OBJECTCROSS SOLUTION



Chapter 4. Types and References: It's 10:00. Do you know where your data is?

Data type, database, Lieutenant Commander Data... it's all important stuff. Without data, your programs are useless. 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**, see how to work with data in your program, and even figure out a few dirty secrets about **objects** (*pssst...objects are data, too*).

The variable's type determines what kind of data it can store

All of the projects in this chapter are Windows Forms applications. If we tell you to create a new project in this chapter but don't specify what type of project to create, assume it's a Windows Forms Application created with Visual Studio for Windows Desktop.

There are a bunch of **types** built into C#, and each one stores a different kind of data. You've already seen some of the most common ones, and you know how to use them. But there are a few that you haven't seen, and they can really come in handy, too.

Types you'll use all the time

It shouldn't come as a surprise that `int`, `string`, `bool`, and `double` are the most common types.

- `int` can store any **whole** number from –2,147,483,648 to 2,147,483,647.

> **NOTE**
>
> A whole number doesn't have a decimal point.

- `string` can hold text of any length (including the empty string `""`).
- `bool` is a Boolean value — it's either `true` or `false`.
- `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. That range looks weird and complicated, but it's actually pretty simple. The "significant figures" part means the *precision* of the number: 35,048,410,000,000, 1,743,059, 14.43857, and 0.00004374155 all have seven significant figures. The 10^{308} thing means that you can store any number as large as 10^{308} (or 1 followed by 308 zeros) — as long as it only has 16 or fewer significant figures. On the other end of the range, 10^{-324} means that you can store any number as small as 10^{-324} (or a decimal point followed by 324 zeros followed by 1)...but, you guessed it, as long as it only has 16 or fewer significant figures.

> **NOTE**
>
> These numbers are called "floating-point"...as opposed to a "fixed point" number, which always has the same number of decimal places.

More types for whole numbers

Once upon a time, computer memory was really expensive, and processors were really slow. And, believe it or not, if you used the wrong type, it could seriously slow down your program. Luckily, times have changed, and most of the time if you need to store a whole number you can just use an `int`. But sometimes you really need something bigger...and once in a while, you need something smaller, too. That's why C# gives you more options:

> **NOTE**
>
> A lot of times, if you're using these types it's because you're solving a problem where it really helps to have the "wrapping around" effect that you'll read about in a few minutes.

- `byte` can store any **whole** number between 0 and 255.
- `sbyte` can store any **whole** number from –128 to 127.

> **NOTE**
>
> The "s" in sbyte stands for "signed," which means it can be negative (the "sign" is a minus sign).

- `short` can store any **whole** number from –32,768 to 32,767.
- `ushort` can store any **whole** number from 0 to 65,535.
- `uint` can store any **whole** number from 0 to 4,294,967,295.

> **NOTE**
>
> The "u" stands for "unsigned."

- `long` can store any **whole** number between minus and plus 9 billion billion.
- `ulong` can store any **whole** number between 0 and about 18 billion billion.

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 types to handle floating-point values:

- `float` can store any number from $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ with 7 significant digits.

> **NOTE**
>
> double is a lot more common than float. Many XAML properties use double values.

- `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.

> **NOTE**
>
> When your program needs to deal with currency, you usually want to use a decimal to store the number.

> **NOTE**
>
> When you used the Value property in your numericUpDown control, you were using a decimal.

Literals have types, too

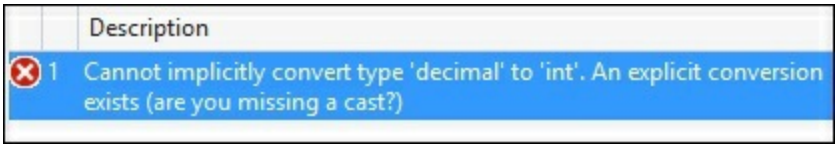
NOTE

A "literal" just means a number that you type into your code. So when you type "int i = 5;", the 5 is a literal.

When you type a number directly into your C# program, you're using a **literal**...and every literal is automatically assigned a type. You can see this for yourself — just enter this line of code that assigns the literal `14.7` to an `int` variable:

```
int myInt = 14.7;
```

Now try to build the program. You'll get this:



That's the same error you'll get if you try to set an `int` equal to a `double` variable. What the IDE is telling you is that the literal `14.7` has a type — it's a `double`. You can change its type to a `float` by sticking an `F` on the end (`14.7F`). And `14.7M` is a decimal.

NOTE

If you try to assign a float literal to a double or a decimal literal to a float, the IDE will give you a helpful message reminding you to add the right suffix. Cool!

NOTE

The "M" stands for "money" — seriously!

A few more useful built-in types

Sometimes you need to store a single character like Q or 7 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 write an escape sequence in your C# code using two characters, but your program stores each escape sequence as a single character in memory.

NOTE

You'll learn a lot more about how char and byte relate to each other in Chapter 9.

And finally, there's one more important type: **`object`**. You've already seen how you can create objects by creating instances of classes. Well, every one of those objects can be assigned to an `object` variable. You'll learn all about how objects and variables that refer to objects work later in this chapter.

BRAIN POWER

The Windows desktop calculator app has a really neat feature called “Programmer” mode, where you can see binary and decimal at the same time!

You can use the Windows desktop calculator to convert between decimal (normal, base-10) numbers and binary numbers (base-2 numbers written with only ones and zeros) — put it in Programmer mode, enter a number, and click the **Bin** radio button to convert to binary. Then click **Dec** to convert it back. Now **enter some of the upper and lower limits for the whole number types** (like –32,768 and 255) and convert them to binary. Can you figure out ***why*** C# gives you those particular limits?

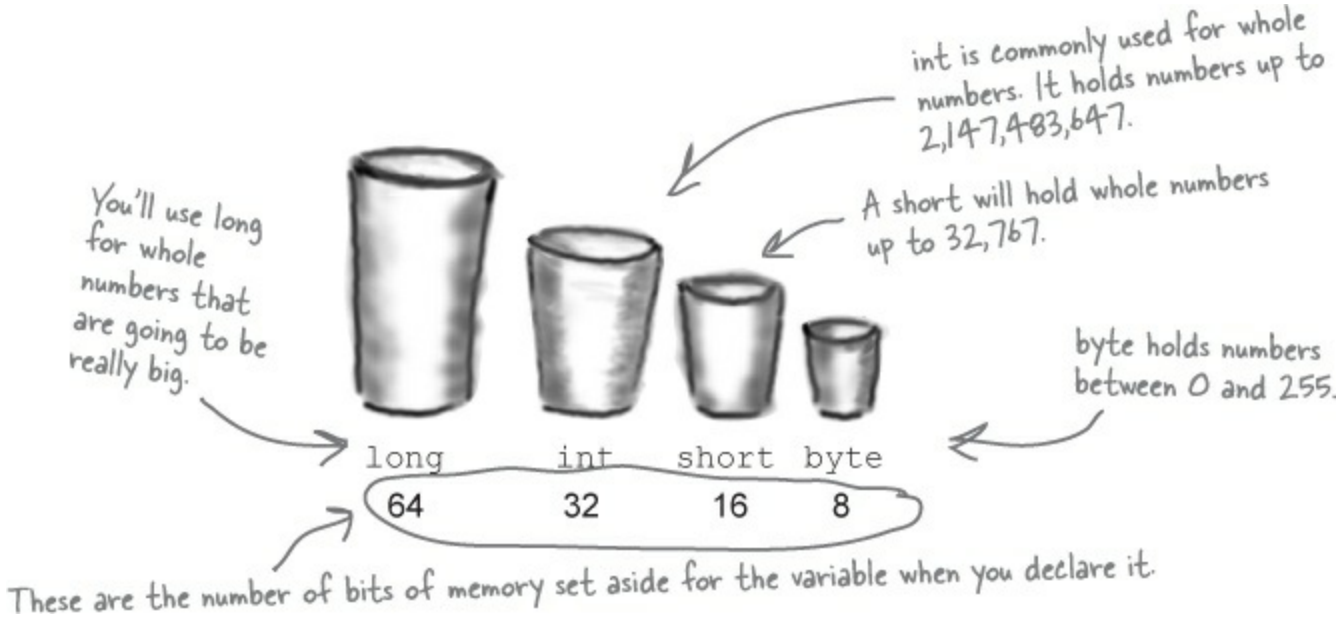
Windows 8 has two different calculators, a Windows Store app and a desktop app. Use Search to find the desktop app, and use its “View” menu to switch between its modes.

A variable is like a data to-go cup

All of your data takes up space in memory. (Remember the heap from 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.

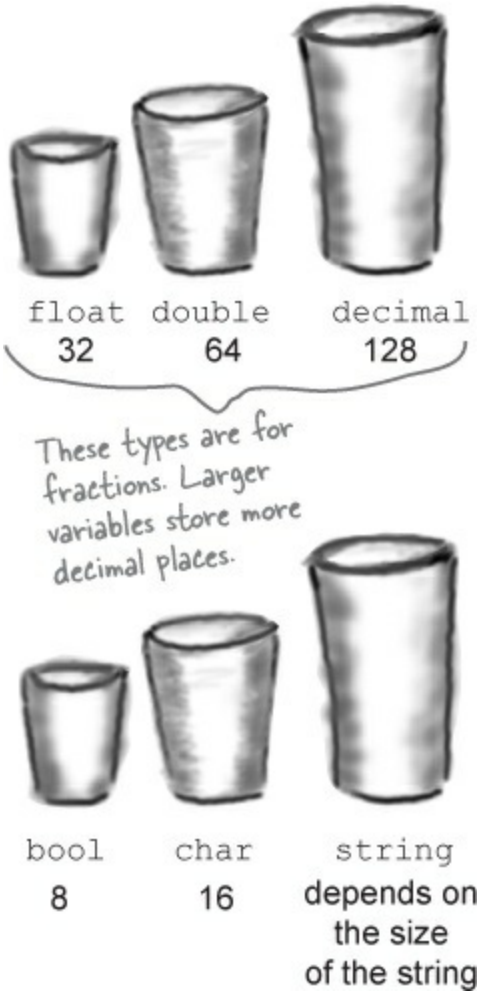
> **NOTE**
>
> Not all data ends up on the heap. Value types usually keep their data in another part of memory called the stack. You'll learn all about that in Chapter 14.

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.



Numbers that have decimal places are stored differently than whole numbers. 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 want to use the `decimal` type.

It's not always about numbers, though. (You wouldn't expect to get hot coffee in a plastic cup or cold coffee in a paper one.) 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.

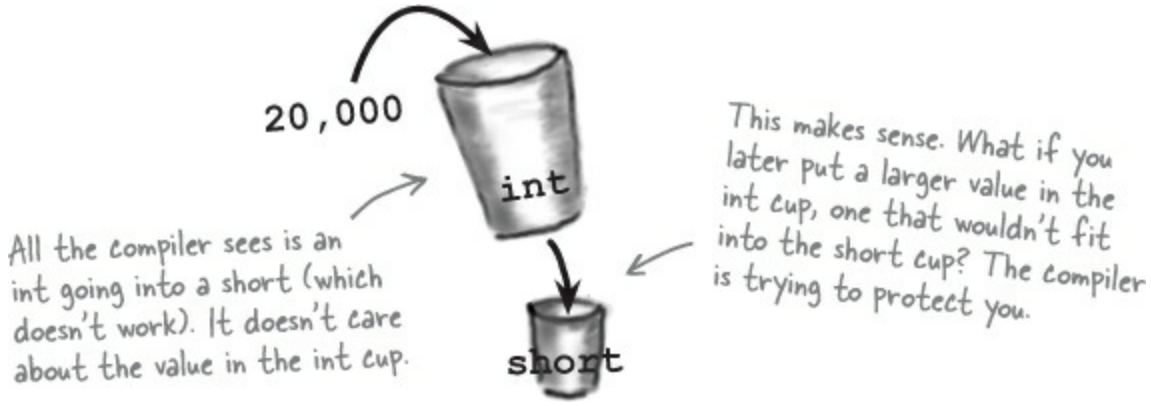

float
double
decimal
32
64
128
These types are for fractions. Larger variables store more decimal places.
bool
char
string
8
16
depends on the size of the string

10 pounds of data in a 5-pound bag

When you declare your variable as one type, that's how your compiler looks at it. 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`, the compiler sees it as `int`-sized and considers it too big to put in a `short` container. The compiler won't make those translations for you on the fly. You need to make sure that you're using the right type for the data you're working with.



SHARPEN YOUR PENCIL

Three of these statements won't compile, 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.

```
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";
```

Even when a number is the right size, you can't just assign it to <u>any</u> variable

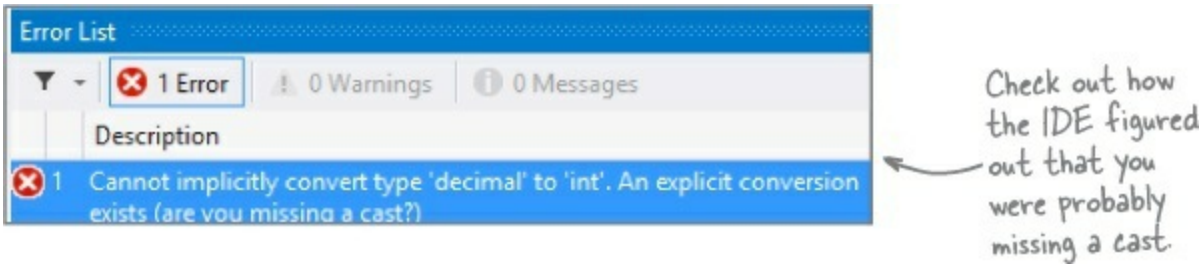
Let's see what happens when you try to assign a `decimal` value to an `int` variable.

DO THIS

- ❶ Create a new Windows Forms project and add a button to it. Then add these lines to the button's **`Click()`** method:

```
decimal myDecimalValue = 10;
int myIntValue = myDecimalValue;

MessageBox.Show("The myIntValue is " + myIntValue);
```

- ❷ Try building your program. Uh oh — you got an error that looks like this:



- ❸ Make the error go away by **casting** the `decimal` to an `int`. Once you change the second line so it looks like this, your program will compile and run:

```
int myIntValue = (int) myDecimalValue;
```

Here's where you **cast** the decimal value to an int.

So what happened?

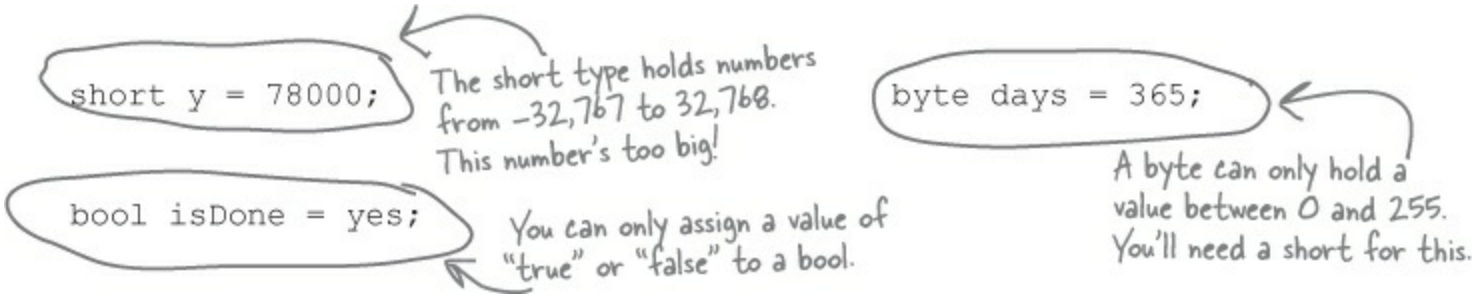
The compiler won't let you assign a value to a variable if it's the wrong type — even if that variable can hold the value just fine — because that's the underlying cause behind an enormous number of bugs, and **the compiler is helping** by nudging you in the right direction. When you use casting, you're 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.

NOTE

Take a minute to flip back to the beginning of the last chapter and check out how you used casting when you passed the NumericUpDown. Value to the Talker Tester form.

SHARPEN YOUR PENCIL SOLUTION

Three of these statements won't compile, 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.



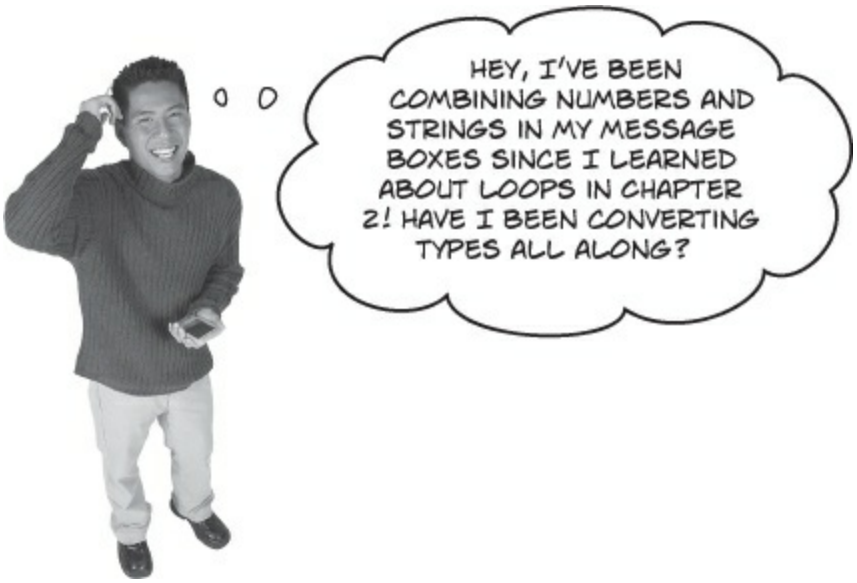
You can read more about the value types in C# here: http://msdn.microsoft.com/en-us/library/s1ax56ch.aspx

When you cast a value that's too big, C# will adjust it automatically

WRAP IT YOURSELF!

There's no mystery to how casting "wraps" the numbers — you can do it yourself. Just pop up the Windows desktop calculator, switch it to Scientific mode, and calculate 365 Mod 256 (using the Mod button, which does a modulo calculation). You'll get 109.

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.



Yes! The + operator converts for you.

What you've been doing is using the `+` operator, which **does a lot of converting for you automatically** — but it's especially smart about it. When you use + to add a number or Boolean to a string, then it'll automatically convert that value to a string, too. If you use + (or `*`, `/`, or `-`) with two different types, it **automatically converts the smaller type to the bigger one**. Here's an example:

When you're assigning a number value to a float, you need to add an F to the end of the number to tell the compiler that it's a float, and not a double. Otherwise, the code won't compile.

```
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`.

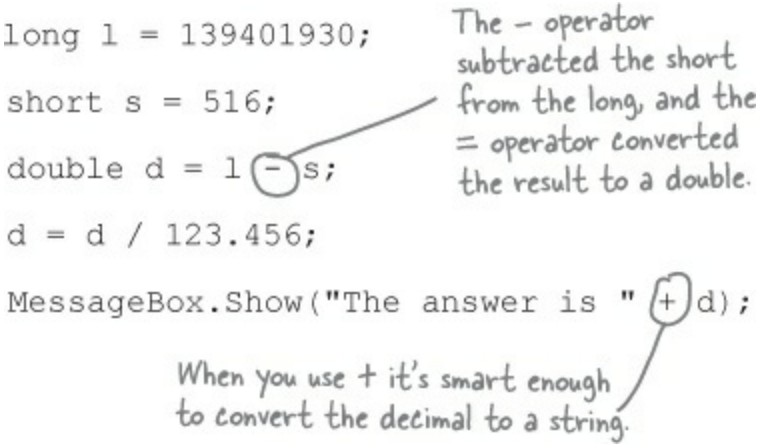
SHARPEN YOUR PENCIL

You can't always cast any type to any other type. Create a new project, drag a button onto a form, double-click on it, and type these statements in. Then build your program — it will give lots of errors. Cross out the ones that give errors. That'll help you figure out which types can be cast, and which can't!

```
int myInt = 10;

byte myByte = (byte)myInt;

double myDouble = (double)myByte;

bool myBool = (bool)myDouble;

string myString = "false";

myBool = (bool)myString;

myString = (string)myInt;

myString = myInt.ToString();

myBool = (bool)myByte;

myByte = (byte)myBool;

short myShort = (short)myInt;

char myChar = 'x';

myString = (string)myChar;

long myLong = (long)myInt;

decimal myDecimal = (decimal)myLong;

myString = myString + myInt + myByte

+ myDouble + myChar;
```

C# does some casting automatically

There are two important conversions that don't require you to do the casting. The first is done automatically any time you use arithmetic operators, like in this example:



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. The first two lines are fine, but the third one won't compile.

```
long x = 139401930;

MessageBox.Show("The answer is " + x);

MessageBox.Show(x);
```

The C# compiler spits out an error that mentions something about invalid arguments (an **argument** is what C# calls the value that you're passing into a method's parameter). That's because the parameter for `MessageBox.Show()` is a `string`, and this code passed a `long`, which is the wrong type for the method. But you can convert it to a string really easily by calling its `ToString()` method. That method is a member of every value type and object. (All of the classes you build yourself have a `ToString()` method that returns the class name.) That's how you can convert `x` to something that `MessageBox.Show()` can use:

```
MessageBox.Show(x.ToString());
```

SHARPEN YOUR PENCIL SOLUTION

You can't always cast any type to any other type. Create a new project, drag a button onto a form, and type these statements into its 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;
```

When you call a method, the arguments must be compatible with the types of the parameters

NOTE

A parameter is what you define in your method. An argument is what you pass to it. A method with an int parameter can take a byte argument.

Try calling `MessageBox.Show(123)` — passing `MessageBox.Show()` a literal `(123)` instead of a string. The IDE won't let you build your program. Instead, it'll show you an error in the IDE: "Argument '1': cannot convert from 'int' to 'string'." Sometimes C# can do the conversion automatically — like if your method expects an `int`, but you pass it a `short` — but it can't do that for `int`s and `string`s.

But `MessageBox.Show()` 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. Go ahead and try typing this completely valid method into a class:

```
public int MyMethod(bool yesNo) {

    if (yesNo) {
        return 45;
    } else {
        return 61;
    }

}
```

One reminder—the code that calls this parameter doesn't have to pass it a variable called yesNo. It just has to pass it a Boolean value or variable. The only place it's called yesNo is inside the method's code.

It works just fine if you pass it what it expects (a `bool`) — call `MyMethod(true)` or `MyMethod(false)`, and it compiles just fine.

But what happens if you pass it an integer or a string instead? The IDE gives you a similar error to the one that you got when you passed 123 to `MessageBox.Show()`. Now try passing it a Boolean, but assigning the return value to a string or passing it on to `MessageBox.Show()`. That won't work, either — the method returns an `int`, not a `long` or the `string` that `MessageBox.Show()` expects.

NOTE

You can assign anything to a variable, parameter, or field with the type object.

When the compiler gives you an "invalid arguments" error, it means that you tried to call a method with variables whose types didn't match the method's parameters.

IF STATEMENTS ALWAYS TEST TO SEE IF SOMETHING'S TRUE

You did this in the code you wrote in "Save the Humans" — go back and have a look; see if you can spot it.

Did you notice how we wrote our if statement like this:

```
if (yesNo) {
```

We didn't have to explicitly say "if (yesNo == true)". That's because an if statement always checks if something's true. You check if something's false using ! (an exclamation point, or the NOT operator). "if (!yesNo)" is the same thing as "if (yesNo == false)". In our code examples from now on, you'll usually just see us do "if (yesNo)" or "if (!yesNo)", and not explicitly check to see if a Boolean is true or false.

EXERCISE

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 do that with nonreserved names too, if you want to.

There are about 77 **reserved words called *keywords*** in C#. These are words reserved by the C# compiler; you can't use them for variable names. You'll know a lot of them really well by the time you finish the book. Here are some you've already used. Write down what you think these words do in C#.

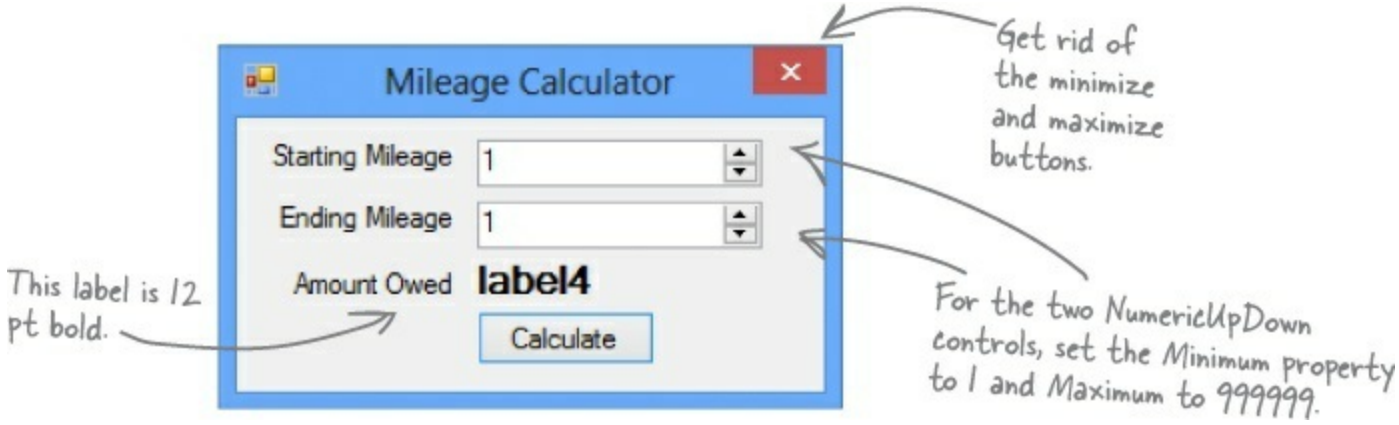
`namespace`	
`for`	
`class`	
`public`	
`else`	
`new`	
`using`	
`if`	
`while`	

Answers in Exercise Solution.

EXERCISE

Create a reimbursement calculator for a business trip. It should allow the user to enter a starting and ending mileage reading from the car's odometer. From those two numbers, it will calculate how many miles she's traveled and figure out how much she should be reimbursed if her company pays her $.39 for every mile she puts on her car.

- **❶ START WITH A NEW WINDOWS FORMS PROJECT.**
  Make the form look like this:



  When you're done with the form, double-click on the button to add some code to the project.

- **❷ CREATE THE FIELDS YOU'LL NEED FOR THE CALCULATOR.**
  Put the fields in the class definition at the top of `Form1`. You need two whole number values to track the starting odometer reading and the ending odometer reading. Call them `startingMileage` and `endingMileage`. You need three numbers that can hold decimal places. Make them `double`s and call them `milesTraveled`, `reimburseRate`, and `amountOwed`. Set the value for `reimburseRate` to .39.

- **❸ MAKE YOUR CALCULATOR WORK.**
  Add code in the `button1_Click()` method to:

  Make sure that the number in the `startingMileage` field is smaller than the number in the `endingMileage` field. If not, show a message box that says "The starting mileage must be less than the ending mileage." Make the title for the message box "Cannot Calculate Mileage."
  Subtract the starting number from the ending number and then multiply it by the reimburse rate using these lines:

```
milesTraveled = endingMileage -= startingMileage;

amountOwed = milesTraveled *= reimburseRate;

label4.Text = "$" + amountOwed;
```

- **❹ RUN IT.**
  Make sure it's giving the right numbers. Try changing the starting value to be higher than the ending value, and make sure it's giving you the message box.

EXERCISE SOLUTION

You were asked to create a reimbursement calculator for a business trip. Here's the code for the first part of the exercise.

```
public partial class Form1 : Form
{
    int startingMileage;
    int endingMileage;
    double milesTraveled;
    double reimburseRate = .39;
    double amountOwed;
    public Form1() {
        InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e){
      startingMileage = (int) numericUpDown1.Value;
      endingMileage = (int)numericUpDown2.Value;
        if (startingMileage < endingMileage){
            milesTraveled = endingMileage -= startingMileage;
            amountOwed = milesTraveled *= reimburseRate;
            label4.Text = "$" + amountOwed;
        } else {
           MessageBox.Show(
       "The starting mileage must be less than the ending mileage",
                           "Cannot Calculate Mileage");
        }
    }
}
```

int works great for whole numbers. This number could go all the way up to 999,999. So a short or a byte won't cut it.

Did you remember that you have to cast the decimal value from the numericUpDown control to an int?

This block is supposed to figure out how many miles were traveled and then multiply them by the reimbursement rate.

We used an alternate way of calling the MessageBox.Show() method here. We gave it two parameters: the first one is the message to display, and the second one goes in the title bar.

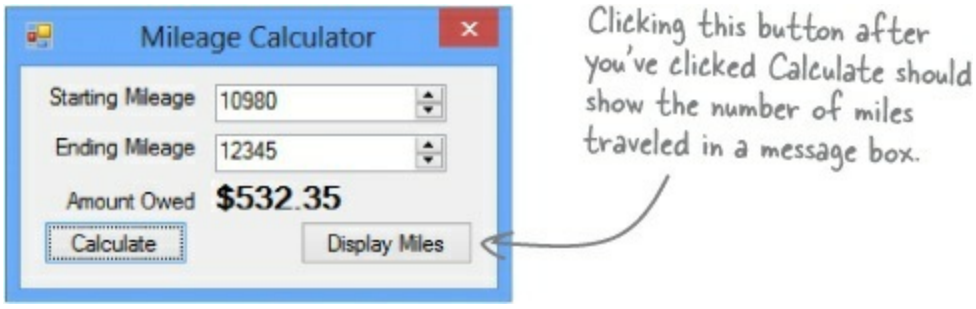
This button seems to work, but it has a pretty big problem. Can you spot it?

Debug the mileage calculator

There's something wrong with the mileage calculator. Whenever your code doesn't work the way you expect it to, there's always a reason for it, and your job is to figure out what that reason is. Let's figure out what went wrong here and see if we can fix it.

DO THIS

- **❶ NOW ADD ANOTHER BUTTON TO THE FORM.**
  Let's track down that problem by adding a button to your form that shows the value of the `milesTraveled` field. (You could also use the debugger for this!)

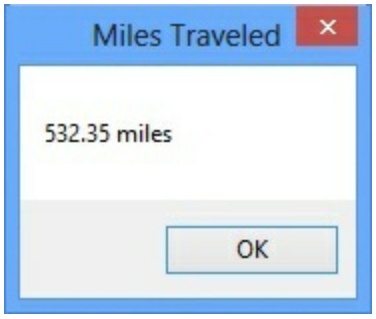


  When you're done with the form, double-click on the Display Miles button to add some code to the project.

- **❷ ONE LINE SHOULD DO IT.**
  All we need to do is get the form to display the `milesTraveled` variable, right? So this line should do that:

```
private void button2_Click(object sender, EventArgs e) {
       MessageBox.Show(milesTraveled + " miles", "Miles Traveled");
}
```

- **❸ RUN IT.**
  Type in some values and see what happens. First enter a starting mileage and ending mileage, and click the Calculate button. Then click the Display Miles button to see what's stored in the `milesTraveled` field.



- **❹ UM, SOMETHING'S NOT RIGHT...**
  No matter what numbers you use, the number of miles always matches the amount owed. Why?

Combining = with an operator

Take a good look at the operator we used to subtract ending mileage from starting mileage (-=). The problem is it doesn't just subtract, it also assigns a value to the variable on the left side of the subtraction sign. The same thing happens in the line where we multiply the number of miles traveled by the reimbursement rate. We should replace the -= and the *= with just - and *:

```
private void button1_Click(object sender, EventArgs e)
{
  startingMileage = (int) numericUpDown1.Value;
  endingMileage = (int)numericUpDown2.Value;
    if (startingMileage < endingMileage){
        milesTraveled = endingMileage -= startingMileage;
        amountOwed = milesTraveled *= reimburseRate;
        label4.Text = "$" + amountOwed;
    } else {
       MessageBox.Show("The starting mileage number must
                       be less than the ending mileage number",
                       "Cannot Calculate Mileage");
    }
```

These are called **compound operators**. This one subtracts startingMileage from endingMileage but also assigns the new value to endingMileage and milesTraveled at the same time.

This is better—now your code won't modify endingMileage and milesTraveled.

```
milesTraveled = endingMileage - startingMileage;
amountOwed = milesTraveled * reimburseRate;
```

So can good variable names help you out here? Definitely! Take a close look at what each variable is supposed to do. You already get a lot of clues from the name `milesTraveled` — you know that's the variable that the form is displaying incorrectly, and you've got a good idea of how that value ought to be calculated. So you can take advantage of that when you're looking through your code to try to track down the bug. It'd be a whole lot harder to find the problem if the incorrect lines looked like this instead:

```
mT = eM -= sM;
aO = mT *= rR;
```

Variables named like this are essentially useless in telling you what their purpose might be.

Objects use variables, too

So far, we've looked at objects separate from other types. But an object is just another data type. Your code treats objects exactly like it treats numbers, strings, and Booleans. It uses variables to work with them:

Using an int

- ❶ Write a statement to declare the integer.

```
int myInt;
```

- ❷ Assign a value to the new variable.

```
myInt = 3761;
```

- ❸ Use the integer in your code.

```
while (i < myInt) {
```

Using an object

- ❶ Write a statement to declare the object.

```
Dog spot;
```

When you have a class like Dog, you use it as the type in a variable declaration statement.

- ❷ Assign a value to the object.

```
spot = new Dog();
```

- ❸ Check one of the object's fields.

```
while (spot.IsHappy) {
```

Objects are just one more type of variable your program can use.

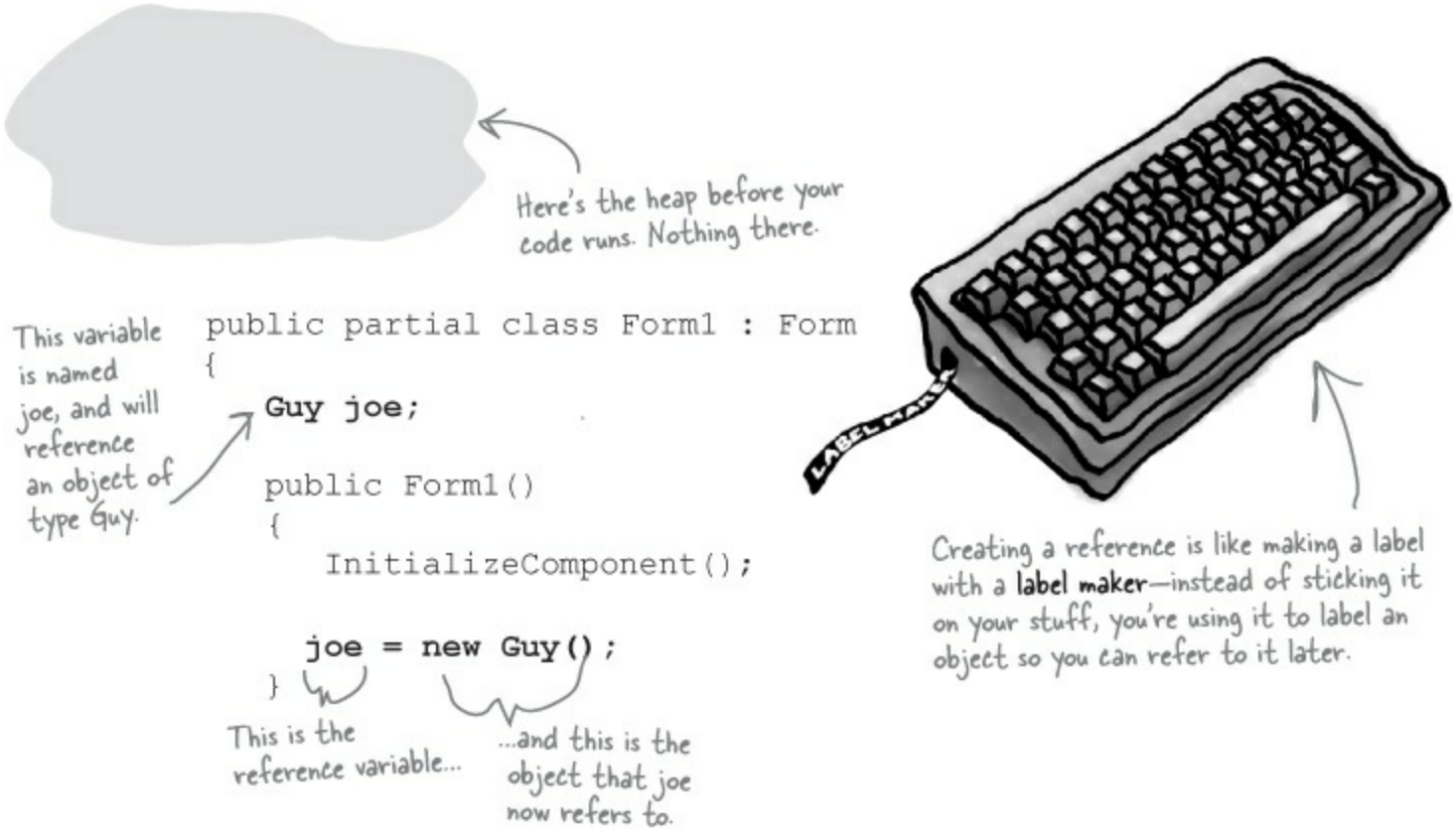
If your program needs to work with a whole number that's really big, use a `long`. If it needs a whole number that's small, use a `short`. If it needs a yes/no value, use a `boolean`. And if it needs something that barks and sits, use a `Dog`. No matter what type of data your program needs to work with, it'll use a variable.

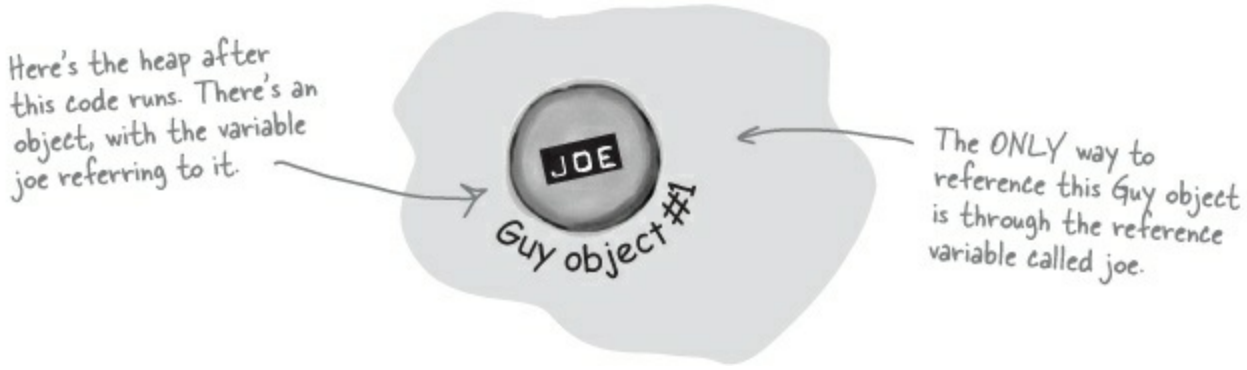
Refer to your objects with reference variables

When you create a new object, you use code like `new Guy()`. But that's not enough; even though that code creates a new `Guy` object on the heap, it doesn't give you a way to *access* that object. ***You need a reference to the object***. So you create 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`.

> **NOTE**
>
> That's called instantiating the object.

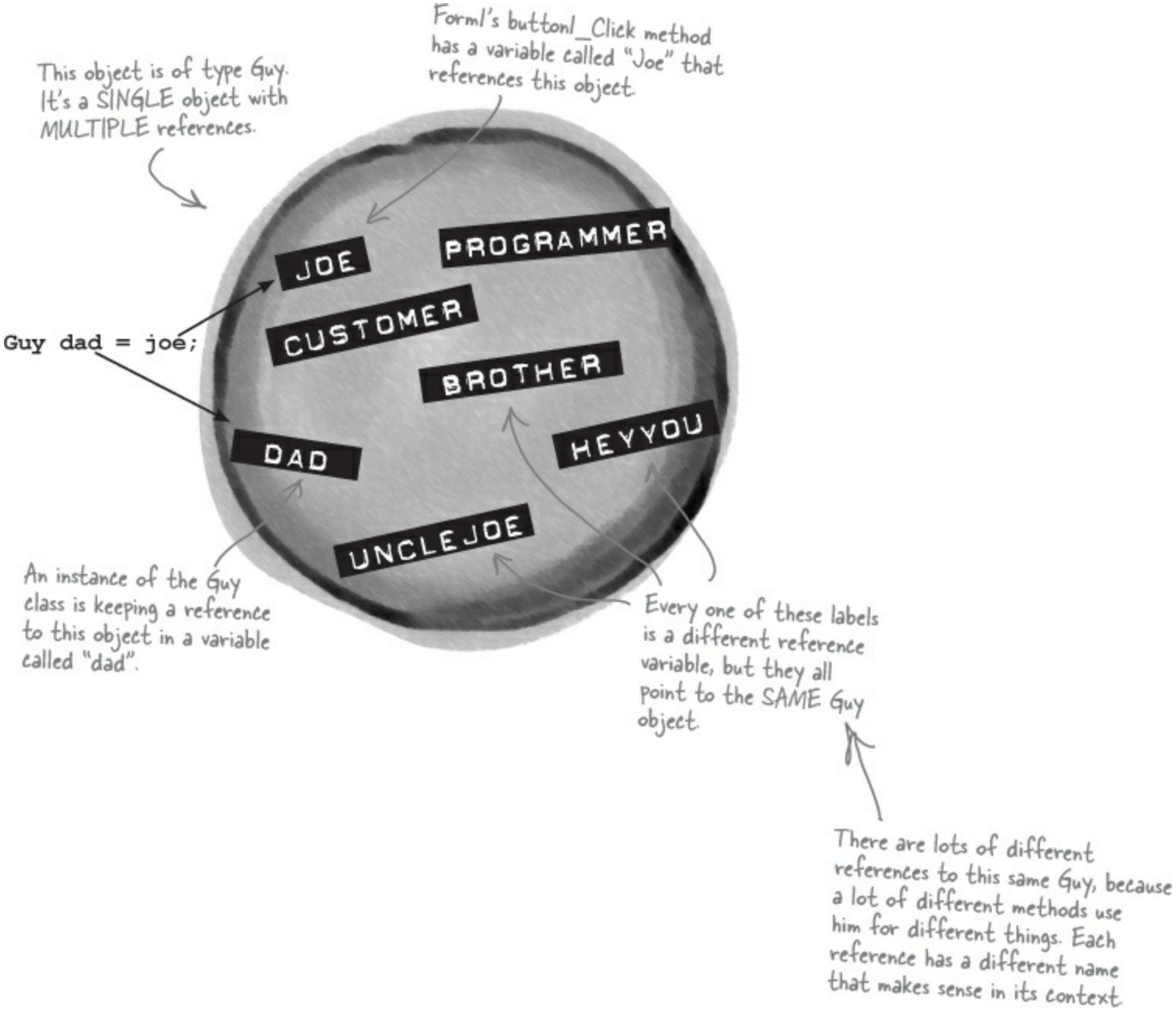
So when you have a variable that is an object type, it's a reference variable: a reference to a particular object. Take a look:





References are like labels for your object

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.



When your code needs to work with an object in memory, it uses a reference, which is a variable whose type is a class of the object it's going to point to. A reference is like a label that your code uses to talk about a specific object.

You never refer to your object directly. For example, you can't write code like `Guy.GiveCash()` if `Guy` is your object type. The C# compiler doesn't know which `Guy` you're talking about, since you might have several instances of `Guy` on the heap. So you need a reference variable, like `joe`, that you assign to a specific instance, like `Guy joe = new Guy()`.

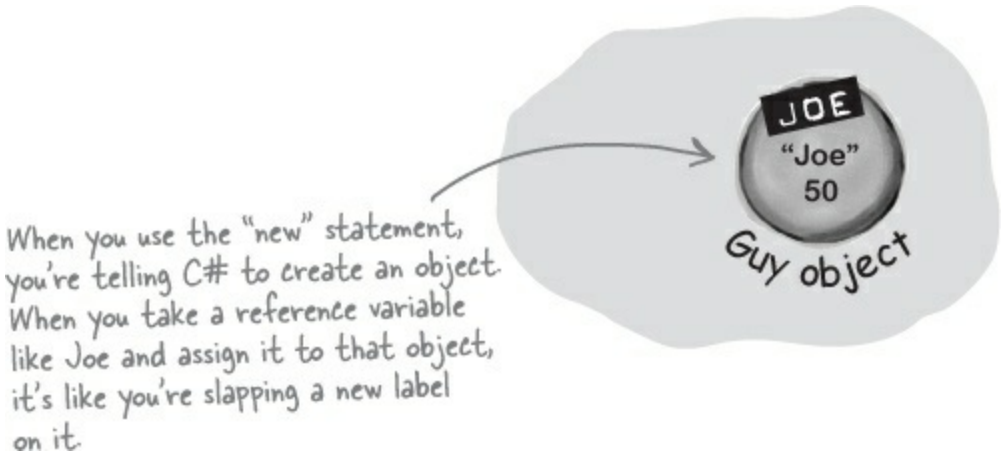
Now you can call (non-static) methods like `joe.GiveCash()`. `joe` refers to a specific instance of the `Guy` class, and your C# compiler knows exactly which instance to use. And, as you saw above, you might have ***multiple labels pointing to the same instance***. So you could say `Guy dad = joe`, and then call `dad.GiveCash()`. That's OK, too — that's what Joe's kid does every day.

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.

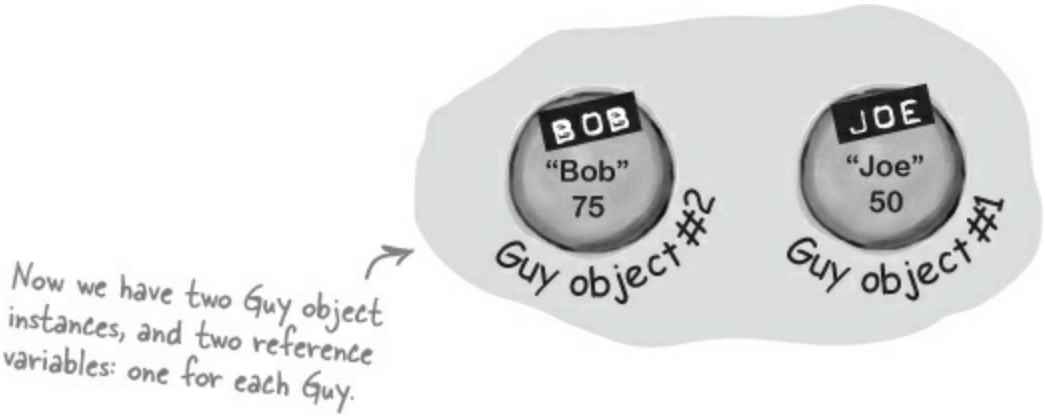
- **❶ Here's some code that creates an object.**

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

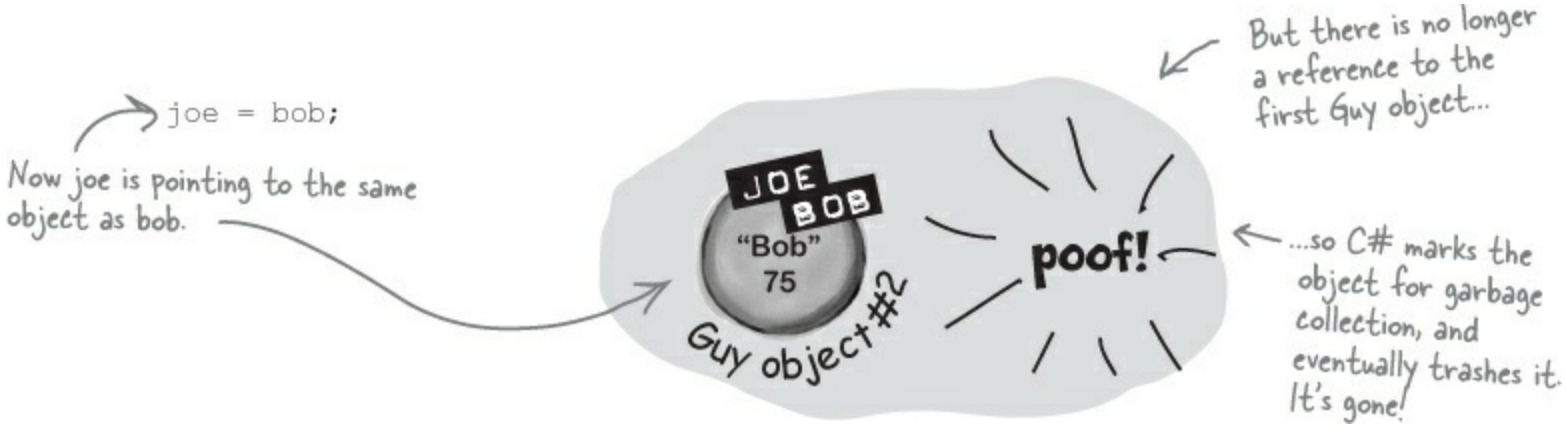


- **❷ Now let's create a second object.**

```
Guy bob = new Guy()
 { Name = "Bob", Cash = 75 };
```



- **❸ Let's take the reference to the first object, and change it to point at the second object.**

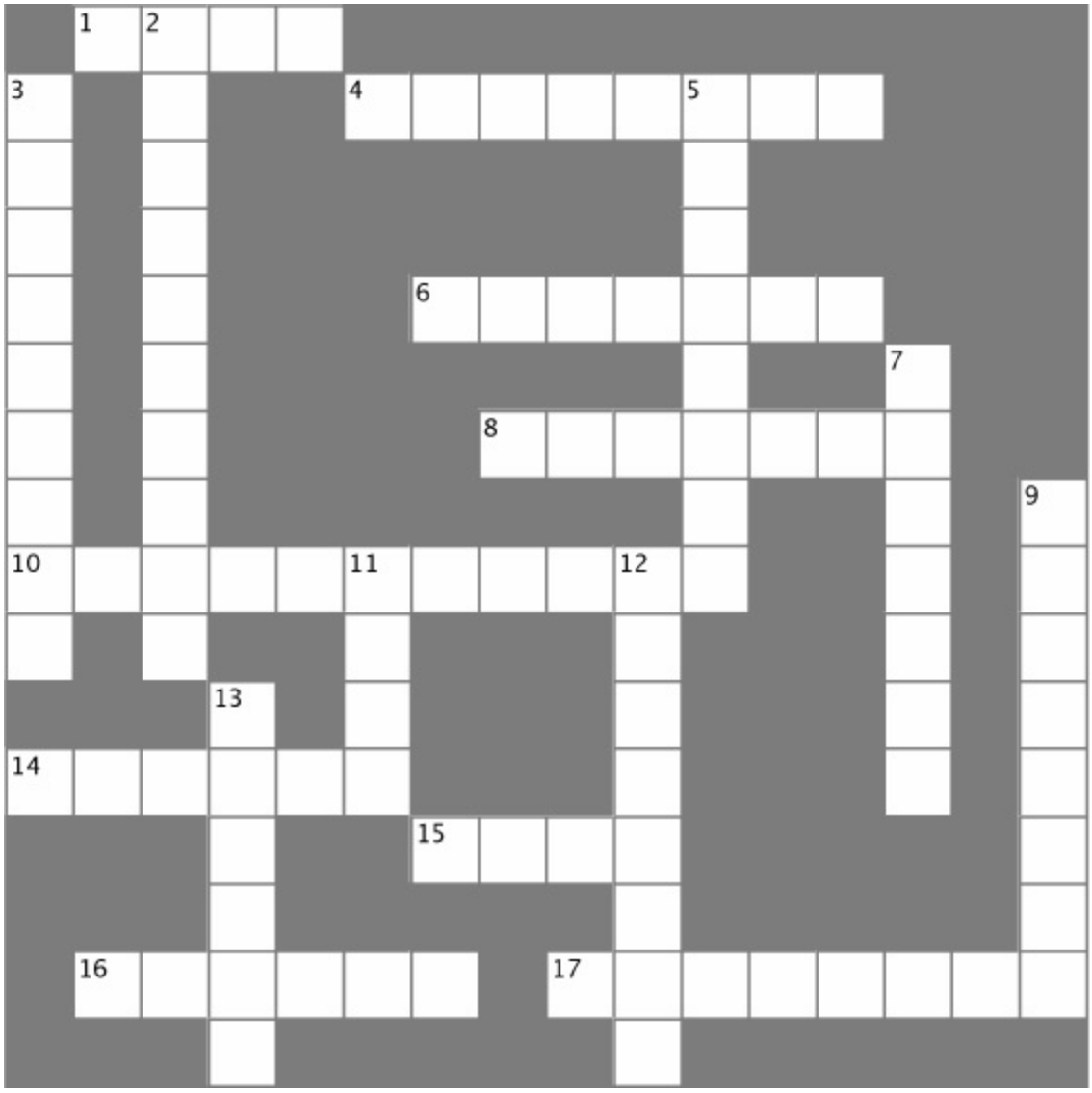


For an object to stay in the heap, it has to be referenced. Some time after the last reference to the object disappears, so does the object.

TYPECROSS

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 this chapter.

When you're done, turn the page and take on the rest of the chapter.



Across	Down
1. The second part of a variable declaration	2. You can combine the variable declaration and the _____ into one statement
4. `namespace`, `for`, `while`, `using`, and `new` are examples of _____ words	3. A variable that points to an object
6. What (`int`) is doing in this line of code: `x = (int) y;`	5. What your program uses to work with data that's in memory
8. When an object no longer has any references pointing to it, it's removed from the heap using _____ collection	7. If you want to store a currency value, use this type
10. What you're doing when you use the + operator to stick two strings together	9. += and -= are this kind of operator
14. The numeric type that holds the biggest numbers	11. A variable declaration always starts with this
15. The type that stores a single letter or number	12. Every object has this method that converts it to a string
16. `\n` and `\r` are _____ sequences	13. When you've got a variable of this type, you can assign any value to it
17. The four whole-number types that only hold positive numbers	

Answers in Typecross Solution.

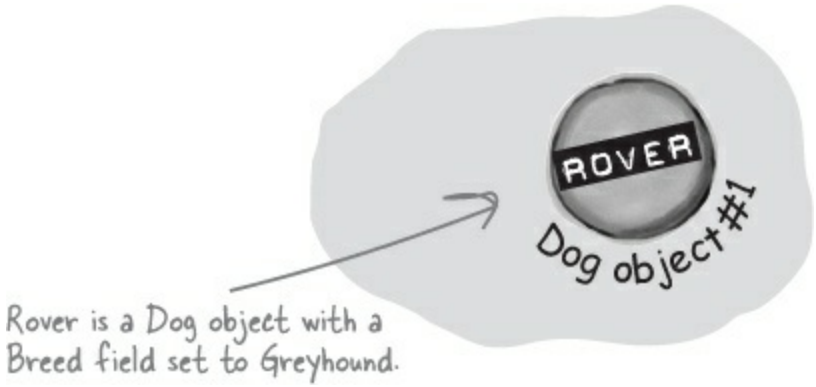
Multiple references and their side effects

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:

- ❶

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
```

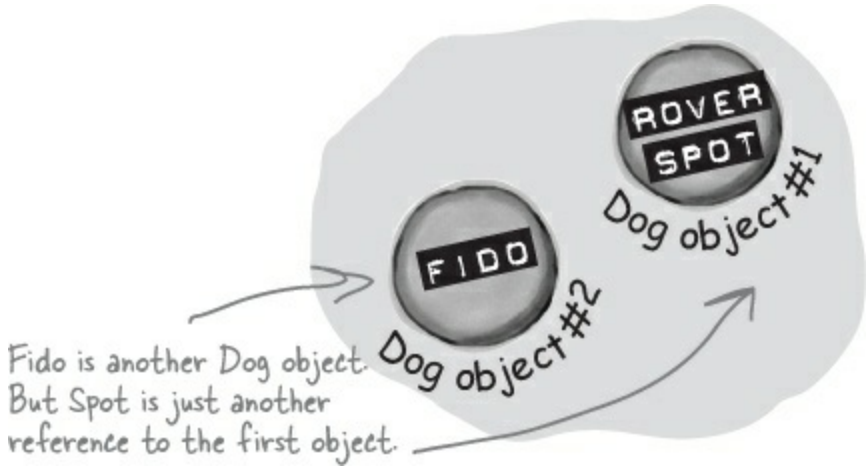
Objects: 1
References: 1



- ❷

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

Objects: 2
References: 3

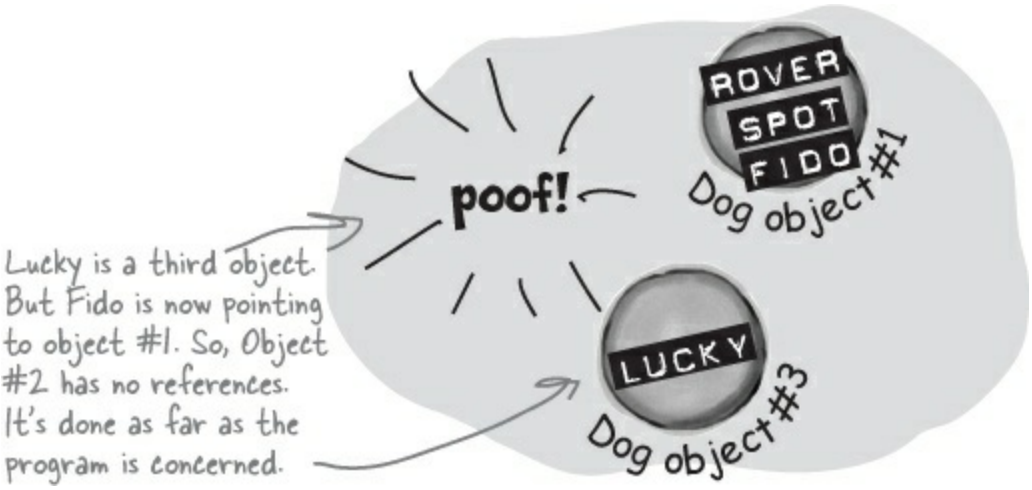


- ❸

```
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 righthand side, draw a picture of the objects and labels in the heap.

- ❶

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog quentin = fido;
```

Objects:_____
References:_____

- ❷

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

Objects:_____
References:_____

- ❸

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

Objects:_____
References:_____

- ❹

```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

Objects:_____
References:_____

- ❺

```
charlie = laverne;
lucky = rinTinTin;
```

Objects:_____
References:_____

SHARPEN YOUR PENCIL SOLUTION

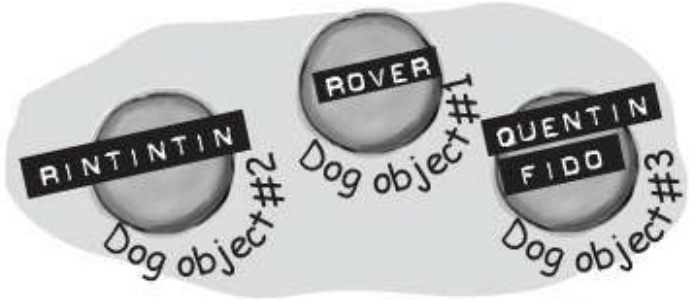
Now it's your turn. Here's one long block of code. Figure out how many objects and references there are at each stage. On the righthand side, draw a picture of the objects and labels in the heap.

1

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog quentin = fido;
```

Objects: 3

References: 4



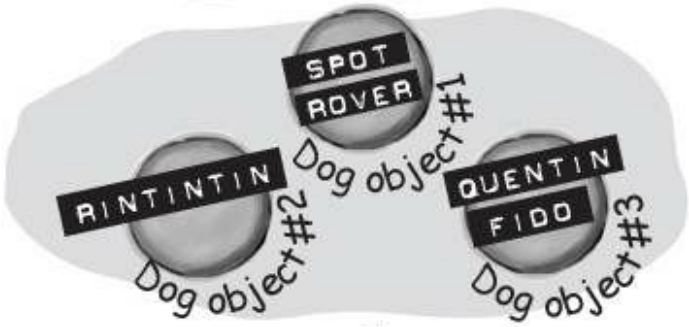
One new Dog object is created, but Spot is the only reference to it. When Spot is set to Rover, that object goes away.

2

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

Objects: 3

References: 5



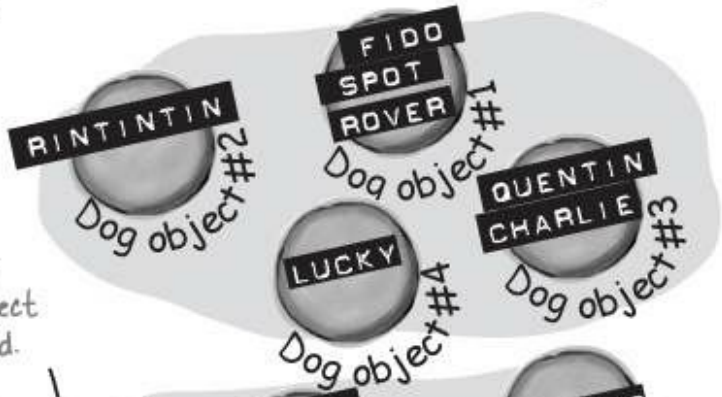
Here a new Dog object is created, but when Fido is set to Rover, Fido's object from #1 goes away.

3

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

Objects: 4

References: 7



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

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

4

```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

Objects: 4

References: 8

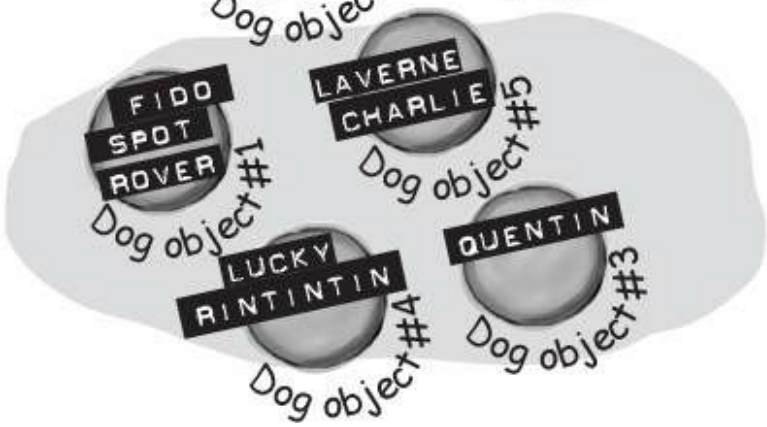
When Rin Tin Tin moved to Lucky's object, the old Rin Tin Tin object disappeared.

5

```
charlie = laverne;
lucky = rinTinTin;
```

Objects: 4

References: 8

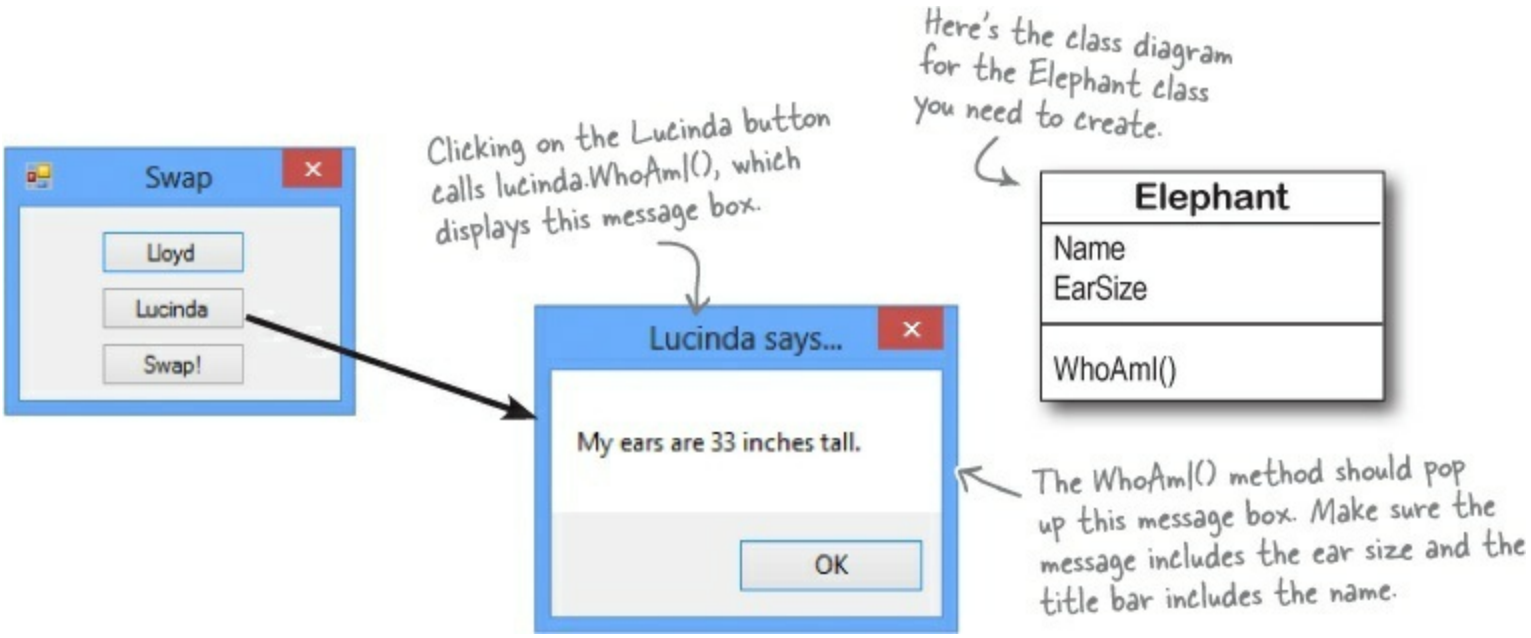


Here the references move around, but no new objects are created. And setting Lucky to Rin Tin Tin did nothing because they already pointed to the same object.

EXERCISE

Create a program with an `Elephant` class. Make two `Elephant` instances and then swap the reference values that point to them, ***without*** getting any `Elephant` instances garbage-collected.

- **❶ Start with a new Windows Forms Application project.**
  Make the form look like this:



- **❷ Create 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`. (Make sure both are public.) Then add a method called `WhoAmI()` that displays a message box that tells you the name and ear size of the elephant.
- **❸ Create two Elephant instances and a reference.**
  Add two `Elephant` fields to the `Form1` class (in the area right below the class declaration) named `Lloyd` and `Lucinda`. Initialize them so they have the right name and ear size. Here are the `Elephant` object initializers to add to your form:

```
lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };
```

- **❹ Make the Lloyd and Lucinda buttons work.**
  Have the Lloyd button call `lloyd.WhoAmI()` and the Lucinda button call `lucinda.WhoAmI()`.
- **❺ Hook up the swap button.**
  Here's the hard part. Make the Swap button ***exchange*** the two references, so that when you click Swap, the `Lloyd` and `Lucinda` variables swap objects and a "Objects swapped" box is displayed. Test out your program by clicking the Swap button and then clicking the other two buttons. The first time you click Swap, the Lloyd button should pop up Lucinda's message box, and the Lucinda button should pop up Lloyd's message box. If you click the Swap button again, everything should go back.

C# garbage-collects any object with no references to it. So here's your hint: If you want to pour a glass of beer into another glass that's currently full of water, you'll need a third glass to pour the water into....

EXERCISE SOLUTION

Create a program with an `Elephant` class. Make two `Elephant` instances and then swap the reference values that point to them, ***without*** getting any `Elephant` instances garbage-collected.

This is the Elephant class definition code in the Elephant.cs file we added to the project. Don't forget the "using System.Windows.Forms;" line at the top of the class. Without it, the MessageBox statement won't work.

```
using System.Windows.Forms;

class Elephant {

  public int EarSize;
  public string Name;

  public void WhoAmI() {
     MessageBox.Show("My ears are " + EarSize + " inches tall.",
        Name + " says…");
  }
}
```

You can put the `using` statements inside the namespace curly brackets if you want.

Here's the Form1 class code from Form1.cs.

```
public partial class Form1 : Form {

    Elephant lucinda;
    Elephant lloyd;

    public Form1()
    {
        InitializeComponent();
        lucinda = new Elephant()
            { Name = "Lucinda", EarSize = 33 };
        lloyd = new Elephant()
            { Name = "Lloyd", EarSize = 40 };
    }

    private void button1_Click(object sender, EventArgs e) {
        lloyd.WhoAmI();
    }

    private void button2_Click(object sender, EventArgs e) {
        lucinda.WhoAmI();
    }

    private void button3_Click(object sender, EventArgs e) {
        Elephant holder;
        holder = lloyd;
        lloyd = lucinda;
        lucinda = holder;
        MessageBox.Show("Objects swapped");
    }
}
```

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 the Holder reference hold onto the Lloyd object until Lucinda can get there.

There's no "new" statement for the reference because we don't want to create another instance of Elephant.

BRAIN POWER

Why do you think we didn't add a **`Swap()`** method to the **`Elephant`** class?

Two references means TWO ways to change an 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. Watch:

DO THIS

- ❶ Add another button to your form.

- ❷ Add this code for the button. Can you guess what's going to happen when you click it?

```
private void button4_Click(object sender, EventArgs e)
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
```

This statement says to set EarSize to 4321 on whatever object the lloyd reference happens to point to.

After this code runs, both the lloyd and lucinda variables reference the SAME Elephant object.

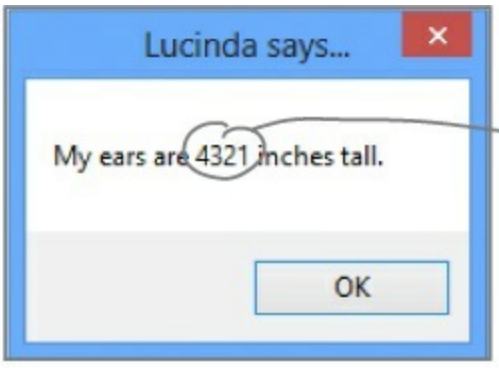
You're calling the WhoAmI() method from the lloyd object.

But lloyd points at the same thing that lucinda does.

Elephant object

- ❸ OK, go ahead and click the new button. Wait a second, that's the Lucinda message box. Didn't we call the `WhoAmI()` method from Lloyd?

It's lucinda's message box...



But we set this EarSize using the lloyd reference! What gives?

Elephant object

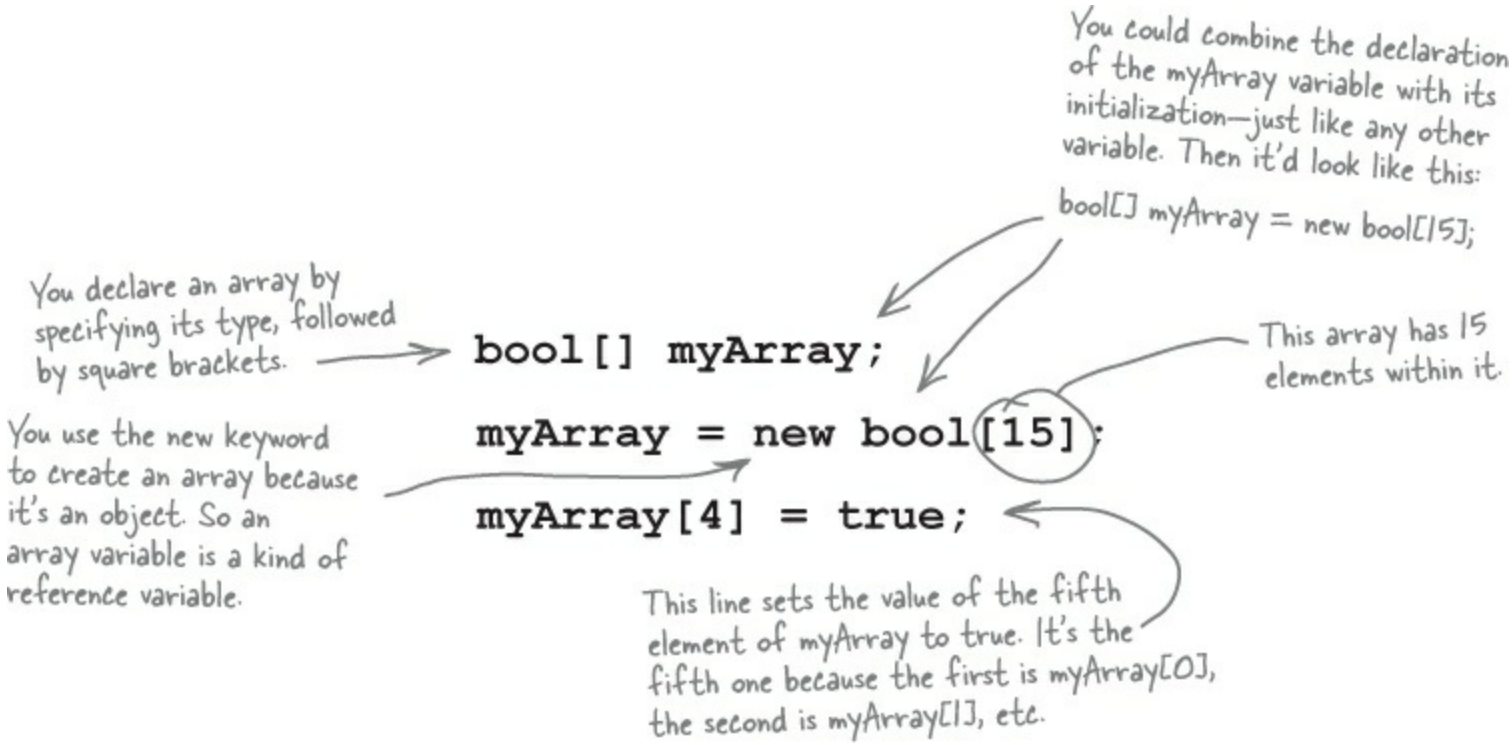
lloyd and lucinda are now interchangeable. Changes to one affect the object that BOTH are pointing at... there's no longer a real difference between lloyd and lucinda, since they point to the SAME object.

Note that the data is NOT being overwritten—the only things changing are the references.

A special case: arrays

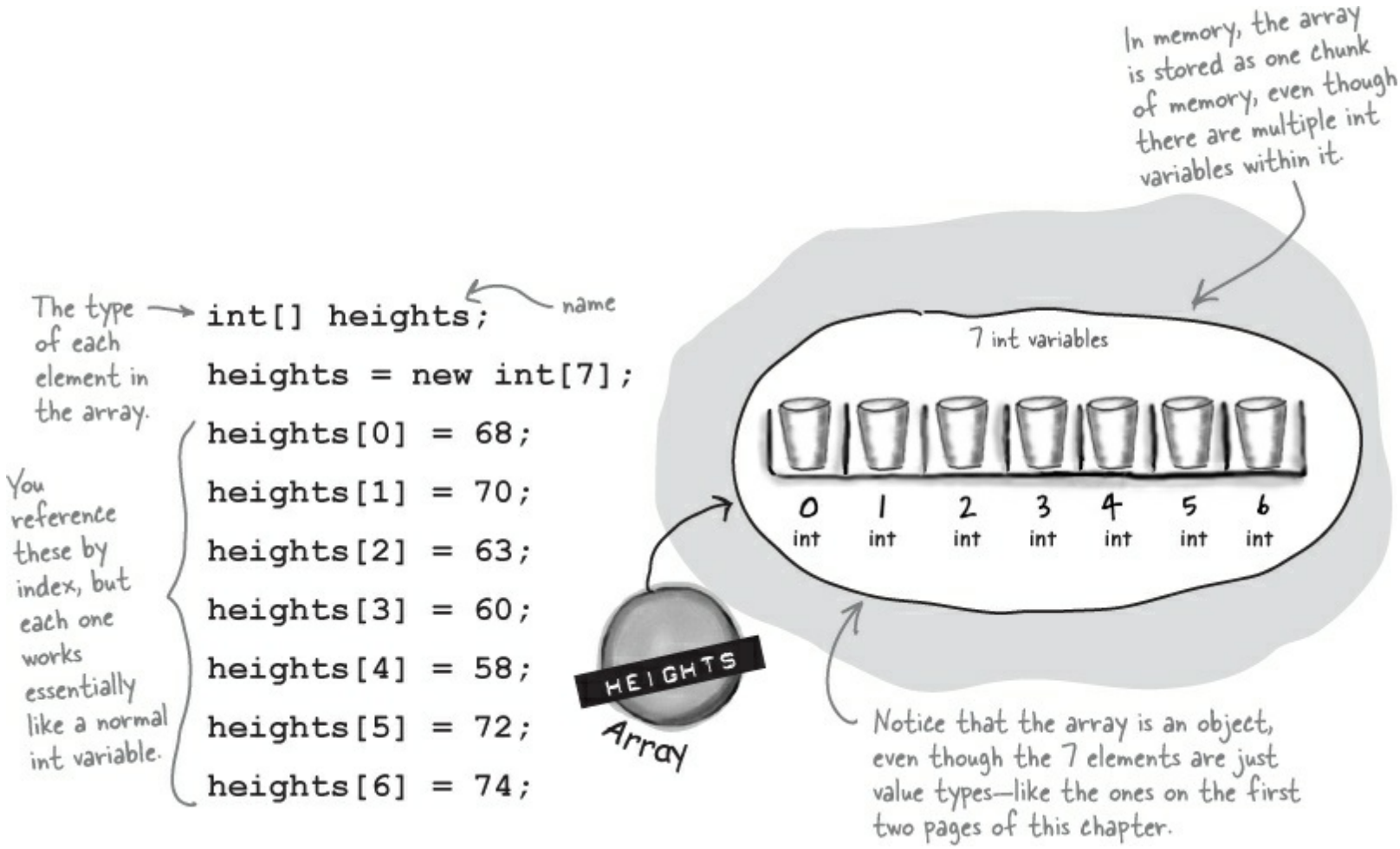
If you have to keep track of a lot of data of the same type, like a list of heights or a group of dogs, you can do it in an **array**. What makes an array special is that it's a **group of variables** that's treated as one object. An array gives you a way of storing and changing more than one piece of 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:

Strings and arrays are different from the other data types you've seen so far, because they're the only ones without a set size (think about that for a bit).



Use each element in an array like it is a normal variable

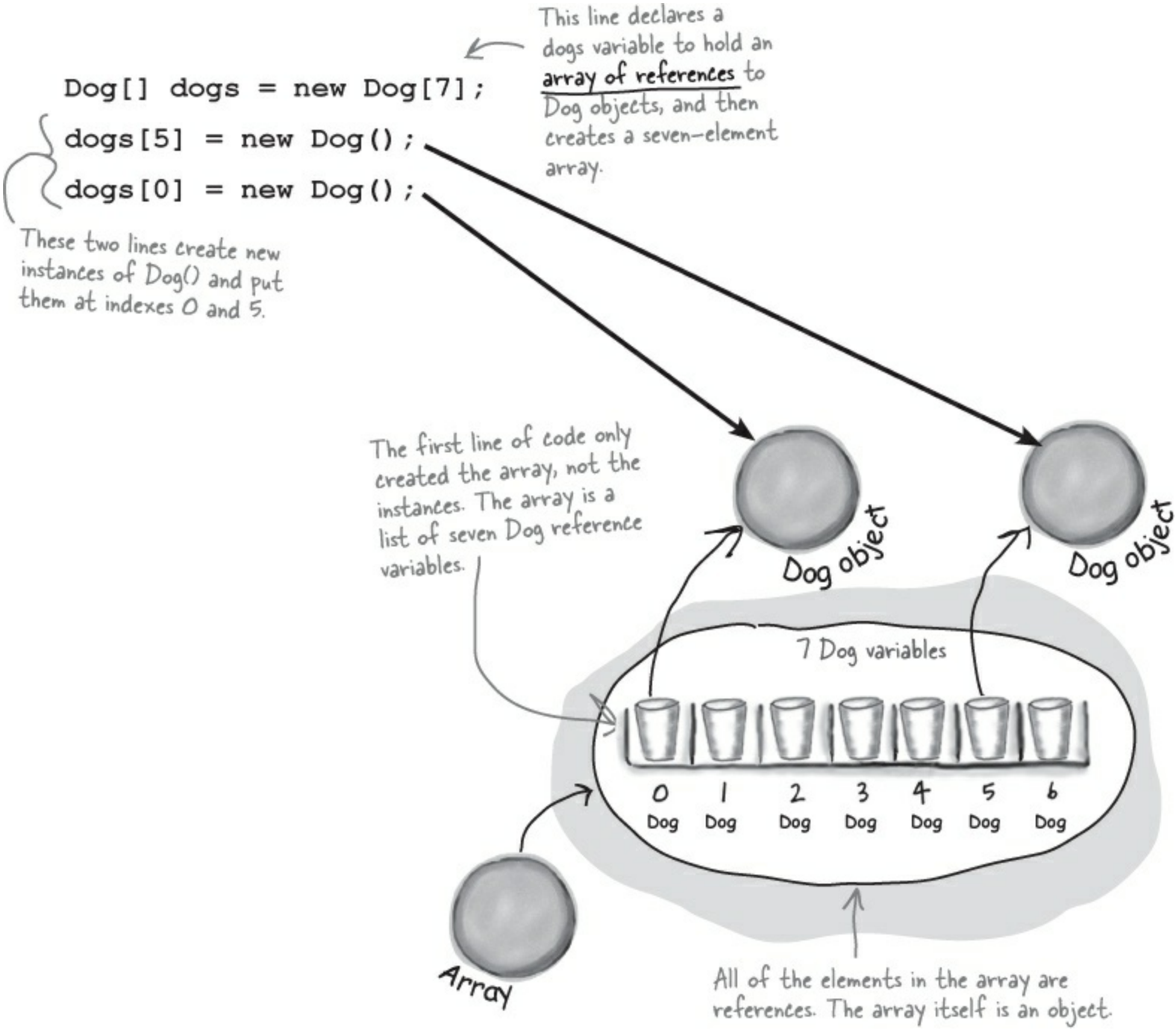
When you use an array, first you need to **declare a reference variable** that points to the array. Then you need to **create the array object** using the `new` statement, specifying how big you want the array to be. Then you can **set the elements** in the array. Here's an example of code that declares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



Arrays can contain a bunch of reference variables, too

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 `int`s, or an array of `Duck` objects, with no problem.

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



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.

AN ARRAY'S LENGTH

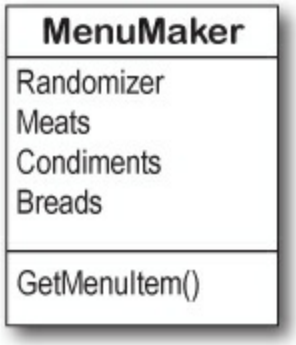
You can find out how many elements are in an array using its Length property. So if you've got an array called heights, then you can use heights. Length to find out how long it is. If there are seven elements in the array, that'll give you 7 — which means the array elements are numbered 0 to 6.

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?

DO THIS

- **❶ Start a new project and add a MenuMaker class.**
  If you need to build a menu, you need ingredients. And arrays would be perfect for those lists. We'll also need some way of choosing random ingredients to combine together into a sandwich. Luckily, the .NET Framework has a built-in class called `Random` that generates random numbers. So we'll have four fields in our class: a Randomizer field that holds a reference to a `Random` object, and three arrays of `string`s to hold the meats, condiments, and breads.



The class has three fields to store three different arrays of strings. It'll use them to build the random menu items.

The field called Randomizer holds a **reference** to a Random object. Calling its Next() method will generate random numbers.

```
class MenuMaker {
    public Random Randomizer;

    string[] Meats = { "Roast beef", "Salami", "Turkey", "Ham", "Pastrami" };
    string[] Condiments = { "yellow mustard", "brown mustard",
                            "honey mustard", "mayo", "relish", "french dressing" };
    string[] Breads = { "rye", "white", "wheat", "pumpernickel",
                        "italian bread", "a roll" };
}
```

Remember, use square brackets to access a member of an array. The value of Breads[2] is "wheat".

Notice how you're initializing these arrays? That's called a **collection initializer**, and you'll learn all about it in Chapter 8.

- **❷ Add a GetMenuItem() method to the class that generates a random sandwich.**
  The point of the class is to generate sandwiches, so let's add a method to do exactly that. It'll use the `Random` object's `Next()` method to choose a random meat, condiment, and bread from each array. When you pass an `int` parameter to `Next()`, the method returns a random number that's less than that parameter. So if your `Random` object is called `Randomizer`, then calling `Randomizer.Next(7)` will return a random number between 0 and 6.
  So how do you know what parameter to pass into the `Next()` method? Well, that's easy — just pass in each array's `Length`. That will return the index of a random item in the array.

```
public string GetMenuItem() {
    string randomMeat = Meats[Randomizer.Next(Meats.Length)];
    string randomCondiment = Condiments[Randomizer.Next(Condiments.Length)];
    string randomBread = Breads[Randomizer.Next(Breads.Length)];
    return randomMeat + " with " + randomCondiment + " on " + randomBread;
}
```

The GetMenuItem() method returns a string that contains a sandwich built from random elements in the three arrays.

The method puts a random item from the Meats array into randomMeat by passing Meats.Length to the Random object's Next() method. Since there are five items in the Meats array, Meats.Length is 5, so Next(5) will return a random number between 0 and 4.

HOW IT WORKS...

The randomizer.Next(7) method gets a random int that's less than 7. Meats.Length returns the number of elements in Meats. So randomizer.Next(Meats.Length) gives you a random number that's greater than or equal to zero, but less than the number of elements in the Meats array.

```
Meats[Randomizer.Next(Meats.Length)]
```

Meats is an array of strings. It's got five elements, numbered from 0 to 4. So Meats[0] equals "Roast Beef", and Meats[3] equals "Ham".

- **❸ Build your form.**
  Add six labels to the form, `label1` through `label6`. Then add code to set each label's `Text` property using a `MenuMaker` object. You'll need to initialize the object using a new instance of the `Random` class. Here's the code:

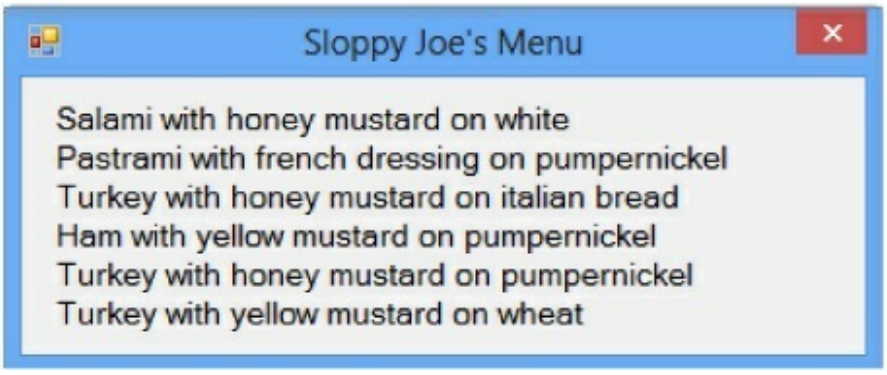
Use an object initializer to set the MenuMaker object's Randomizer field to a new instance of the Random class.

```
public Form1() {
    InitializeComponent();

    MenuMaker menu = new MenuMaker() { Randomizer = new Random() };

    label1.Text = menu.GetMenuItem();
    label2.Text = menu.GetMenuItem();
    label3.Text = menu.GetMenuItem();
    label4.Text = menu.GetMenuItem();
    label5.Text = menu.GetMenuItem();
    label6.Text = menu.GetMenuItem();
}
```

Now you're all set to generate six different random sandwiches using the GetMenuItem() method.

Here's something to think about. What would happen if you forgot to initialize the MenuMaker object's Randomizer field? Can you think of a way to keep this from happening?

When you run the program, the six labels show six different random sandwiches.



Objects use references to talk to each other

So far, you've seen forms talk to objects by using reference variables to call their methods and check their fields. Objects can call one another's methods using references, too. In fact, there's nothing that a form can do that your objects can't do, because **your form is just another object**. 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.

- **❶ HERE'S A METHOD TO TELL AN ELEPHANT TO SPEAK.**
  Let's add a method to the `Elephant` class. Its first parameter is a message from an `elephant`. Its second parameter is the `elephant` that said it:

```
public void TellMe(string message, Elephant whoSaidIt) {
    MessageBox.Show(whoSaidIt.Name + " says: " + message);
}
```

  Here's what it looks like when it's called. You can add to `button4_Click()`, but add it **before the statement that resets the references!** (`lloyd = lucinda;`)

```
lloyd.TellMe("Hi", lucinda);
```

  We called Lloyd's `TellMe()` method, and passed it two parameters: "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.
- **❷ HERE'S A METHOD THAT CALLS ANOTHER METHOD.**
  Now let's add this `SpeakTo()` method to the `Elephant` class. It uses a special keyword: **`this`**. That's a reference that **lets an object talk about itself**.

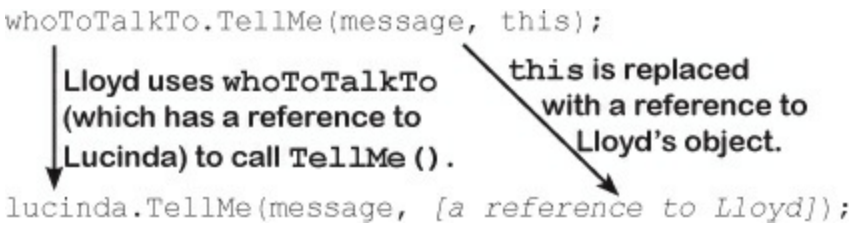
```
public void SpeakTo(Elephant whoToTalkTo, string message) {
    whoToTalkTo.TellMe(message, this);
}
```

This method in the Elephant class calls another elephant's TellMe() method. It lets one elephant communicate with another one.

  Let's take a closer look at how this works.

```
lloyd.SpeakTo(lucinda, "Hello");
```

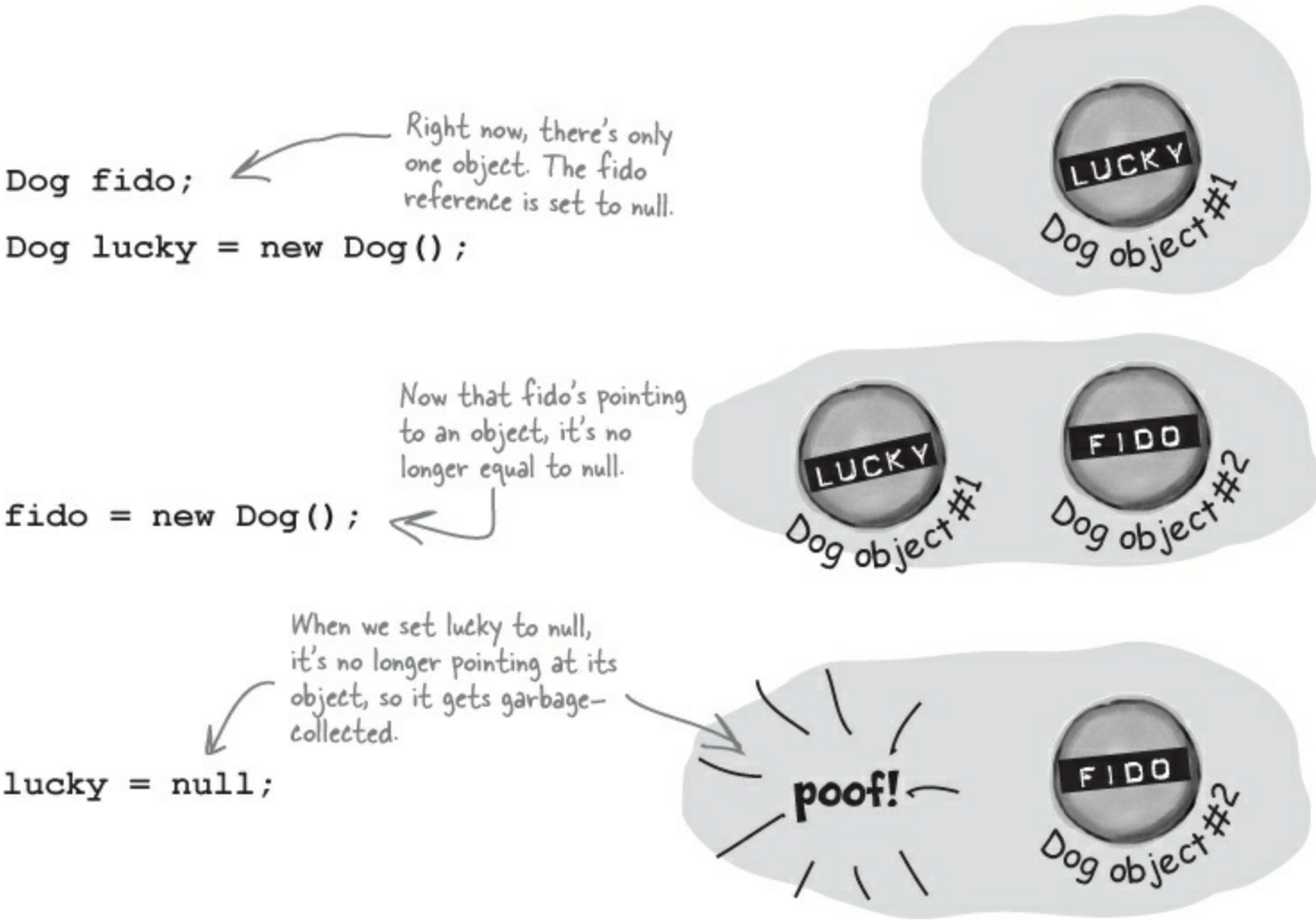
  When Lloyd's `SpeakTo()` method is called, it uses its `whoToTalkTo` parameter (which has a reference to Lucinda) to call Lucinda's `TellMe()` method.



So Lucinda acts as if she was called with (`"Hello", lloyd`), and shows this message:

Where no object has gone before

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 anything.



THERE ARE NO DUMB QUESTIONS

Q: **Q: One more time — my form is an object?**

A: **A:** Yes! That's why your form's code starts with a class declaration. Open up code for a form and see for yourself. Then open up *Program.cs* in any program you've written so far and look inside the `Main()` method — you'll find "`new Form1()`".

Q: **Q: Why would I ever use `null`?**

A: **A:** There are a few ways you see `null` used in typical programs. The most common way is testing for it:

```
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).

Q: **Q: You keep talking about garbage collecting, but what's actually doing the collecting?**

A: **A:** Remember how we talked about the **Common Language Runtime (or CLR)** back at the beginning of Chapter 2? That's the virtual machine that runs all .NET programs. A *virtual machine* is a way for it to isolate running programs from the rest of the operating system. One thing that virtual machines do is manage the memory that they use. 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.

THERE ARE NO DUMB QUESTIONS

Q: **Q: I'm still not sure I get how references work.**

A: **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 you have a (nonstatic) method called `Dog.Bark()` or `Dog.Beg()`, you can create a reference called `spot`. Then you can use that to access `spot.Bark()` or `spot.Beg()`. You could also change information in the fields for the object using the reference. So you could change a Breed field using `spot.Breed`.

Q: **Q: Wait, 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: **A:** Yes. If `rover` is a reference to the same object as spot, changing `rover.Breed` to "beagle" would make it so that `spot.Breed` was "beagle."

Q: **Q: I still don't get that stuff about different types holding different sized values. What's the deal with that?**

A: **A:** OK. 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.

Q: **Q: Remind me again — what does `this` do?**

A: **A:** `this` is a special variable that you can only use inside an object. When you're inside a class, you use `this` to refer to any field or method of that particular instance. It's especially useful when you're working with a class whose methods call other classes. One object can use it to send **a reference to itself** to another object. So if `Spot` calls one of `Rover`'s methods passing `this` as a parameter, he's giving `Rover` a reference to the `Spot` object.

Any time you've got code in an object that's going to be instantiated, the instance can use the special this variable that has a reference to itself.

BULLET POINTS

- When you declare a variable you specify a type and a variable name. Sometimes you combine it with setting the value on the same line of code.

> **NOTE**
>
> There's a very specific case where you don't declare a type. You'll learn about it when you use the var keyword in Chapter 14.

- 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).
- There are a few types (like `short` to `int`) that C# knows how to convert automatically. 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.
- There are some words 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.
- References are like labels: you can have as many references to an object as you want, and they all refer to the same thing.
- If an object doesn't have any references to it, it eventually gets garbage-collected.

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 void button1_Click(object sender, EventArgs e)
{
  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++)
  {
    if (elephants[i].EarSize > biggestEars.EarSize)
    {
      biggestEars = elephants[i];
    }
  }
  MessageBox.Show(biggestEars.EarSize.ToString());
}
```

We're creating an array of seven Elephant() references.

Every array starts with index 0, so the first elephant in the array is Elephants[0].

This line makes the biggestEars reference point at whatever elephant elephants[i] points to.

Be careful—this loop starts with the **second element** of the array (at index 1) and iterates six times until i is equal to the length of the array.

Iteration #1 biggestEars.EarSize = _____

Iteration #2 biggestEars.EarSize = _____

Iteration #3 biggestEars.EarSize = _____

Iteration #4 biggestEars.EarSize = _____

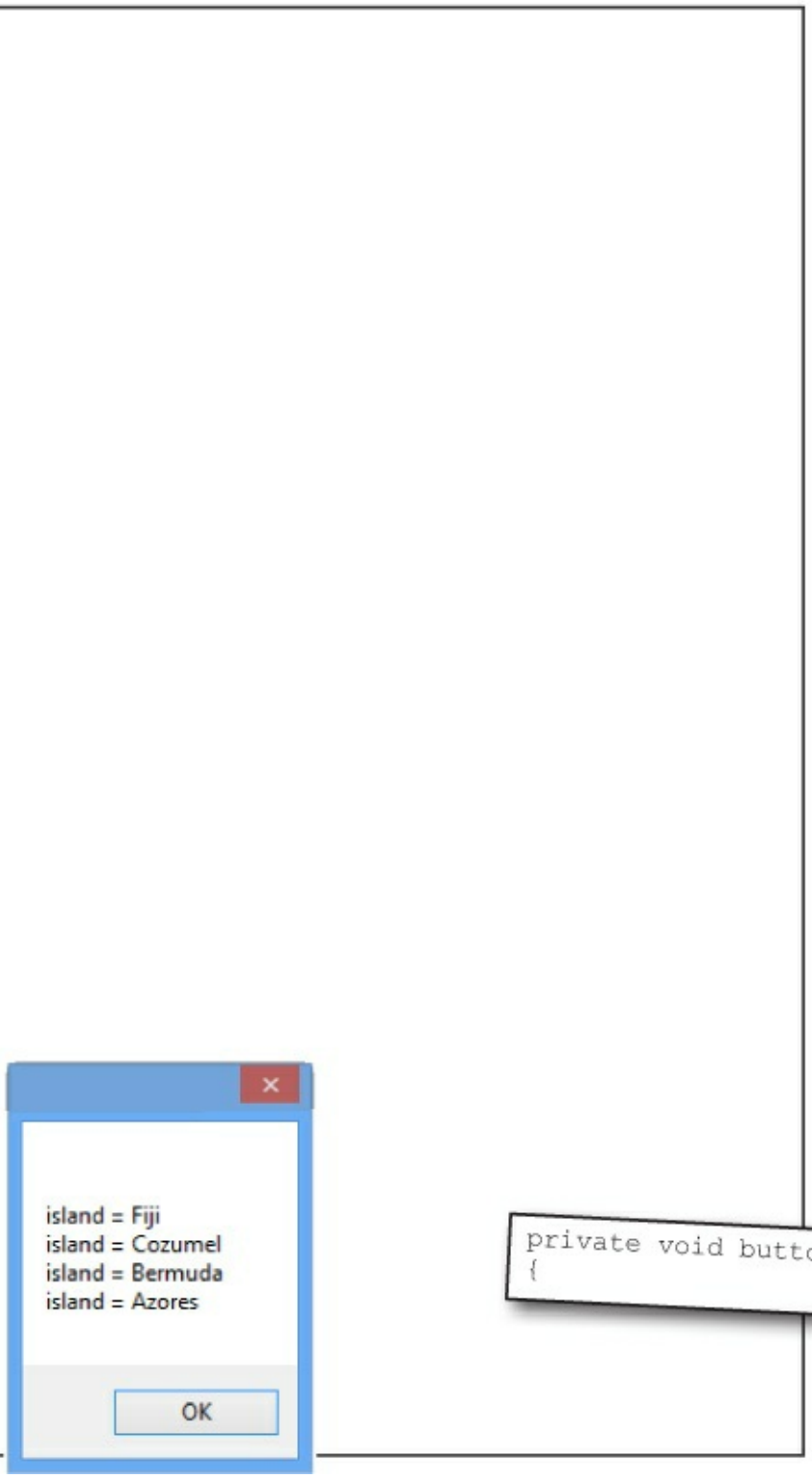
Iteration #5 biggestEars.EarSize = _____

Iteration #6 biggestEars.EarSize = _____

Answers in Sharpen your pencil Solution.

CODE MAGNETS

The code for a button is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working method that produces the output listed below?



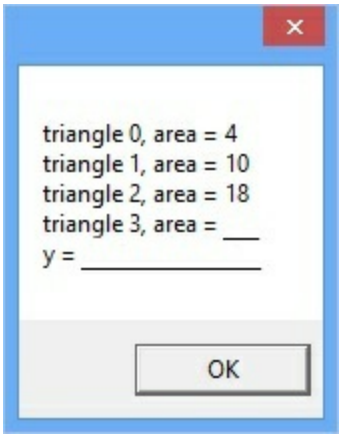

```
int y = 0;
refNum = index[y];
islands[0] = "Bermuda";
islands[1] = "Fiji";
islands[2] = "Azores";
islands[3] = "Cozumel";
int refNum;
while (y < 4) {
result += islands[refNum];
MessageBox.Show(result);
index[0] = 1;
index[1] = 3;
index[2] = 0;
index[3] = 2;
}
}
string[] islands = new string[4];
result += "\nisland = ";
int[] index = new int[4];
y = y + 1;
private void button1_Click (object sender, EventArgs e)
{
string result = "";
```

Answers in Code Magnets Solution.

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 a class that will compile and run, and produce the output listed.

Output



Bonus Question!

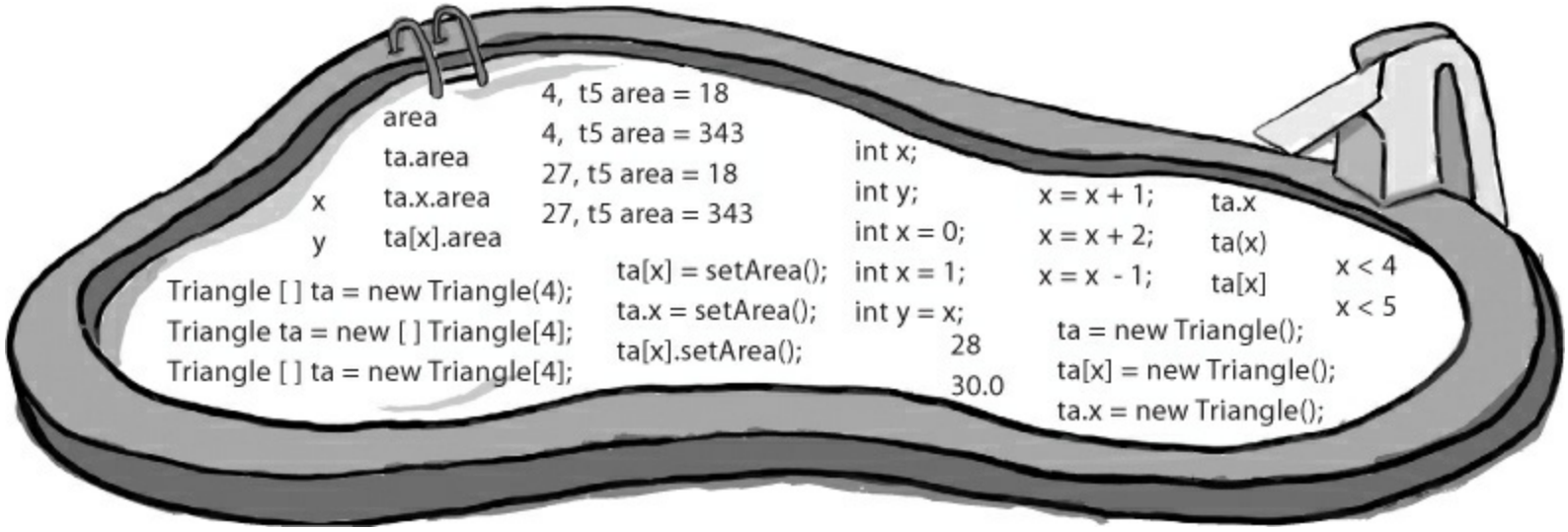
For extra bonus points, use snippets from the pool to fill in the two blanks missing from the output.

```
class Triangle
{
    double area;
    int height;
    int length;
    public static void Main(string[] args)
    {
        string results = "";

        __________

        _____________________________
        while ( _________ )
        {

            ____________________________
            _____.height = (x + 1) * 2;
            _____.length = x + 4;

            ____________________
            results += "triangle " + x + ", area";
            results += " = " + _____.area + "\n";

            ____________

        }

        ____________
        x = 27;
        Triangle t5 = ta[2];
        ta[2].area = 343;
        results += "y = " + y;
        MessageBox.Show(results +
            ", t5 area = " + t5.area);
    }
    void setArea()
    {
        _____________ = (height * length) / 2;
    }
}
```

Here's the entry point for the application. Assume it's in a file with the right "using" lines at the top.

Hint: SetArea() is NOT a static method. Flip back to Chapter 3 for a refresher on what the static keyword means.

Note: each snippet from the pool can be used more than once.

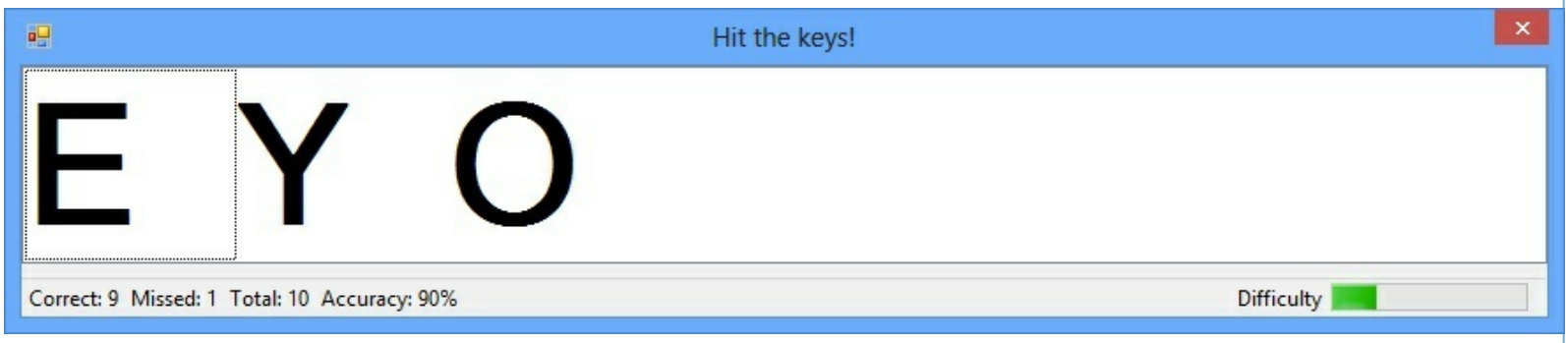


Answers in Pool Puzzle Solution.

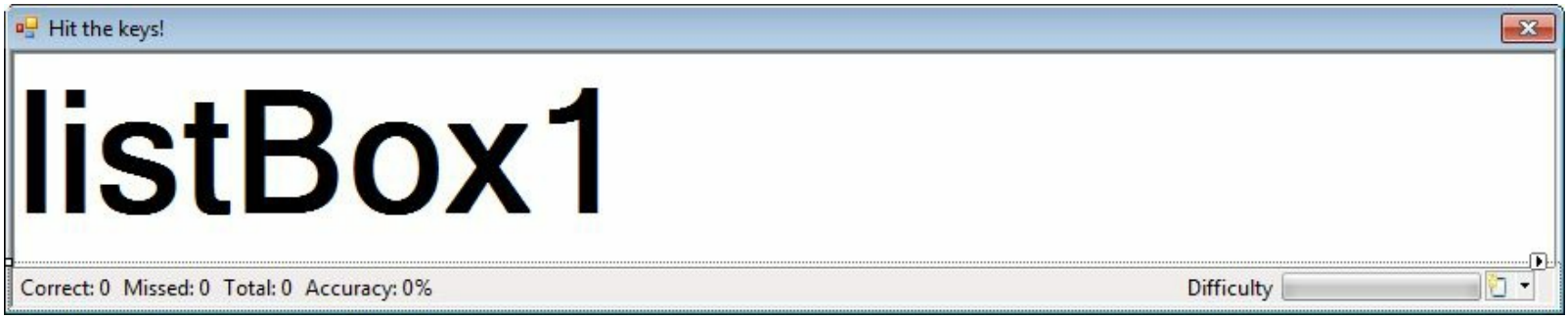
Build a typing game

You've reached a milestone...you know enough to build a game! Here's how your game will work. The form will display random letters. If the player types one of them, it disappears and the accuracy rate goes up. If the player types an incorrect letter, the accuracy rate goes down. As the player keeps typing letters, the game goes faster and faster, getting more difficult with each correct letter. If the form fills up with letters, the game is over!

DO THIS



- **❶ BUILD THE FORM.**
  Here's what the form will look like in the form designer:



You'll need to:

Turn off the minimize box and maximize box. Then set the form's **`FormBorderStyle`** property to **`Fixed3D`**. That way, the player won't be able to accidentally drag and resize it. Then resize it so that it's much wider than it is tall (we set our form's size to 876, 174).

Drag a ListBox out of the toolbox onto the form. Set its **`Dock`** property to Fill, and its **`MultiColumn`** property to True. Set its **`Font`** to 72 point bold.

In the toolbox, expand the All Windows Forms group at the top. This will display many controls. Find the Timer control and double-click on it to add it to your form.

Find the StatusStrip in the All Windows Forms group in the toolbox and double-click on it to add a status bar to your form. You should now see the StatusStrip and Timer icons in the gray area at the bottom of the form designer:

See how you can use a Timer to make your form do more than one thing at once? Take a minute and flip to #4 in the "Leftovers" appendix to learn about another way to do that.

RELAX

You'll be using three new controls, but they're easy to work with!

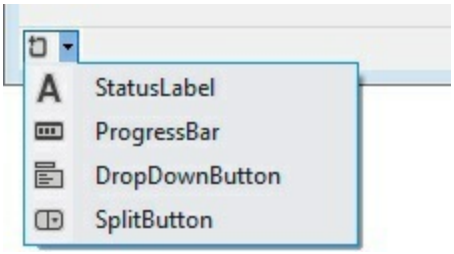
Even though you haven't seen a ListBox, StatusStrip, or Timer before, you already know how to set their properties and work with them in your code. You'll learn a lot more about them in the next few chapters.

- **❷ SET UP THE STATUSSTRIP CONTROL.**
  Take a closer look at the status bar at the bottom of the screenshot. On one side, it's got a series of labels:

  And on the other side, it's got a label and a progress bar:

  Add a StatusLabel to your StatusStrip by clicking its drop-down and selecting StatusLabel. Then do the following:

  Use the Properties window to set its `(Name)` to correctLabel and its `Text` to "Correct: 0". Add three more StatusLabels: `missedLabel`, `totalLabel`, and `accuracyLabel`, and set their `Text` properties to "Missed: 0", "Total: 0", and "Accuracy: 0%".



  Add one more StatusLabel. Set its `Spring` to True, `TextAlign` to MiddleRight, and `Text` to "Difficulty". Finally, add a ProgressBar and name it `difficultyProgressBar`.
  Set the StatusStrip's `SizingGrip` property to False (hit Escape if you've got a child StatusLabel or ProgressBar selected to return the IDE's focus to the parent StatusStrip).

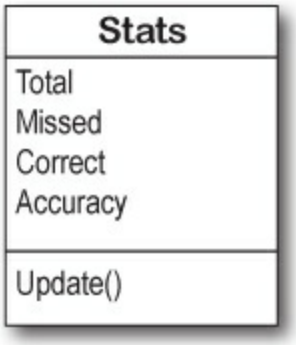
- **❸ SET UP THE TIMER CONTROL.**
  Did you notice how your Timer control didn't show up on your form? That's because the Timer is a *nonvisual control*. It doesn't actually change the look and feel of the form. It does exactly one thing: it **calls a method over and over again**. Set the Timer control's **`Interval`** property to 800, so that it calls its method every 800 milliseconds. Then **double-click on the *timer1* icon** in the designer. The IDE will do what it always does when you double-click on a control: it will add a method to your form. This time, it'll add one called **`timer1_Tick`**. Here's the code for it:

```
private void timer1_Tick(object sender, EventArgs e)
{
    // Add a random key to the ListBox
    listBox1.Items.Add((Keys)random.Next(65, 90));
    if (listBox1.Items.Count > 7)
    {
        listBox1.Items.Clear();
        listBox1.Items.Add("Game over");
        timer1.Stop();
    }
}
```

You'll add a field called "random" in just a minute. Can you guess what its type will be?

The Timer class has a Start() method, but you don't need to call it for this project. Instead, you'll set its Enabled property to True, which makes it start automatically.

- **❹ ADD A CLASS TO KEEP TRACK OF THE PLAYER STATS.**
  If the form is going to display the total number of keys the player pressed, the number that were missed and the number that were correct, and the player's accuracy, then we'll need a way to keep track of all that data. Sounds like a job for a new class! Add a class called `Stats` to your project. It'll have four `int` fields called `Total`, `Missed`, `Correct`, and `Accuracy`, and a method called `Update` with one `bool` parameter: `true` if the player typed a correct letter that was in the ListBox, or `false` if the player missed one.



```
class Stats
{
    public int Total = 0;
    public int Missed = 0;
    public int Correct = 0;
    public int Accuracy = 0;

    public void Update(bool correctKey)
    {
        Total++;

        if (!correctKey)
        {
            Missed++;
        }
        else
        {
            Correct++;
        }

        Accuracy = 100 * Correct / Total;
    }
}
```

Every time the Update() method is called, it recalculates the % correct and puts it in the Accuracy field.

- **❺ ADD FIELDS TO YOUR FORM TO HOLD A STATS OBJECT AND A RANDOM OBJECT.**

  You'll need an instance of your new **`Stats`** class to actually store the information, so add a field called **`stats`** to store it. And you already saw that you'll need a field called **`random`** — it'll contain a **`Random`** object.

  Add the two fields to the top of your form:

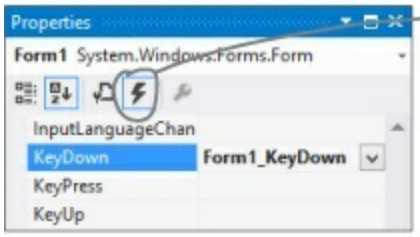
```
public partial class Form1 : Form
{
    Random random = new Random();
    Stats stats = new Stats();
    ...
```

Before you go on, there are three properties you need to set. Set the Timer control's `Enabled` property to `True`, the ProgressBar control's `Maximum` property to `701`, and the Form's `KeyPreview` property to `True`. Take a minute and figure out why you need those properties. What happens if you don't set them?

- **❻ HANDLE THE KEYSTROKES.**

  There's one last thing your game needs to do: any time the player hits a key, it needs to check if that key is correct (and remove the letter from the ListBox if it is), and update the stats on the StatusStrip.

  Go back to the form designer and select the form. Then go to the Properties window and click on the lightning bolt button. Scroll to the **KeyDown** row and **double-click on it**. This tells the IDE to add a method called `Form1_KeyDown()` that gets called every time the user presses a key. Here's the code for the method:



Click this button to change the Properties window's view. The button to the left of it switches the Properties window back to showing you properties.

These are called events, and you'll learn a lot more about them later on.

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    // If the user pressed a key that's in the ListBox, remove it
    // and then make the game a little faster
    if (listBox1.Items.Contains(e.KeyCode))
    {
        listBox1.Items.Remove(e.KeyCode);
        listBox1.Refresh();
        if (timer1.Interval > 400)
            timer1.Interval -= 10;
        if (timer1.Interval > 250)
            timer1.Interval -= 7;
        if (timer1.Interval > 100)
            timer1.Interval -= 2;
        difficultyProgressBar.Value = 800 - timer1.Interval;

        // The user pressed a correct key, so update the Stats object
        // by calling its Update() method with the argument true
        stats.Update(true);
    }
    else
    {
        // The user pressed an incorrect key, so update the Stats object
        // by calling its Update() method with the argument false
        stats.Update(false);
    }

    // Update the labels on the StatusStrip
    correctLabel.Text = "Correct: " + stats.Correct;
    missedLabel.Text = "Missed: " + stats.Missed;
    totalLabel.Text = "Total: " + stats.Total;
    accuracyLabel.Text = "Accuracy: " + stats.Accuracy + "%";
}
```

This if statement checks the ListBox to see if it contains the key the player pressed. If it does, then the key gets removed from the ListBox and the game difficulty is increased.

This is the part that increases the difficulty as the player gets more keys right. You can make the game easier by reducing the amounts that are subtracted from timer1.Interval, or make it harder by increasing them.

When the player presses a key, the Form1_KeyDown() method calls the Stats object's Update() method to update the player stats, and then it displays them in the StatusStrip.

This game only runs once. Can you figure out how to modify it so the player can start a new game when it's displaying "Game Over"?

- **❼ RUN YOUR GAME.**

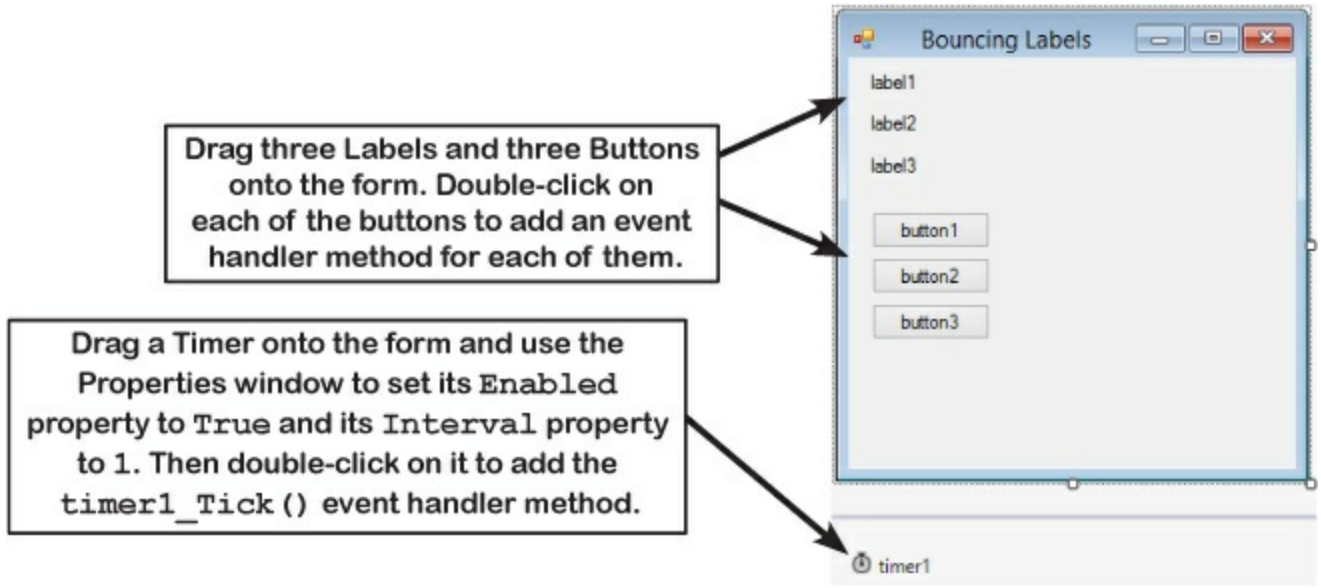
Your game's done! Give it a shot and see how well you do. You may need to adjust the font size of the ListBox to make sure it holds exactly seven letters, and you can change the difficulty by adjusting the values that are subtracted from `timer1.Interval` in the `Form1_KeyDown()` method.

Controls are objects, just like any other object

You've built plenty of forms by dragging controls out of the toolbox. It turns out that those controls are just regular old objects. And since they're objects, you can add references to them and work with them like you'd work with an instance of a class that you wrote yourself. Let's see a live example of that by building a program that animates some Label controls by bouncing them back and forth on a form.

DO THIS

- ❶ Create a new Windows Forms Application and **build this form**.



- ❷ **Add a class** called `LabelBouncer`. Here's the code for it:

```
using System.Windows.Forms;

class LabelBouncer {
    public Label MyLabel;

    public bool GoingForward = true;

    public void Move() {
        if (MyLabel != null) {
            if (GoingForward == true) {
                MyLabel.Left += 5;
                if (MyLabel.Left >= MyLabel.Parent.Width - MyLabel.Width) {
                    GoingForward = false;
                }
            }
            else
            {
                MyLabel.Left -= 5;
                if (MyLabel.Left <= 0) {
                    GoingForward = true;
                }
            }
        }
    }
}
```

You'll need this "using" line because Label is in this namespace.

This class has a field called MyLabel with the type Label, which means it holds a reference to a Label object. Like all references, it starts out null. It will get set to one of the labels on the form.

This Boolean flips from true to false to true again as the label bounces back and forth across the form.

The Move() method figures out if the label has hit the right edge of the form by using >= to check if its Left property is greater than or equal to the width of the form.

Why do you think we need to subtract the width of the label from the width of the form?

When you drag a control around a form, the IDE sets the Top and Left properties. Your programs can use these properties to move controls around the form.

All you need to do to bounce a label across a form is to create a new instance of the `LabelBouncer` class, set its `MyLabel` field to point to a Label control on the form, and then call its `Move()` method over and over again.
Each time the `Move()` method is called, the `LabelBouncer` nudges the label by changing its `Left` property. If the `GoingForward` field is `true`, then it nudges it to the right by adding 5; otherwise, it nudges it to the left by subtracting 5.
Every control has a `Parent` property that contains a reference to the form, **because the form is an object too**!

- ❸ **Here's the code for the form**. See if you can figure out exactly what's going on here. It uses an array of `LabelBouncer` objects to bounce labels back and forth, and has the Timer's `Tick` event handler method call their `Move()` methods over and over again.

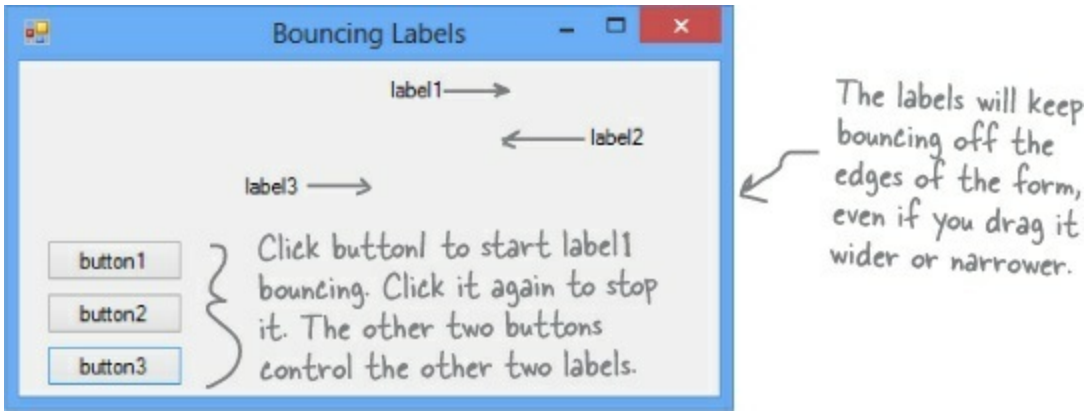
The form stores an array of `LabelBouncer` references in a field called `bouncers`. When the `ToggleBouncing()` method is called, it uses the `index` parameter to check an element of the array. If the element is null, it creates a new `LabelBouncer` object and stores its reference in the array; otherwise, it clears the element by setting it to `null`.

```
public partial class Form1 : Form {

    public Form1() {
        InitializeComponent();
    }

    LabelBouncer[] bouncers = new LabelBouncer[3];

    private void ToggleBouncing(int index, Label labelToBounce) {
        if (bouncers[index] == null) {
            bouncers[index] = new LabelBouncer();
            bouncers[index].MyLabel = labelToBounce;
        }
        else {
            bouncers[index] = null;
        }
    }

    private void button1_Click(object sender, EventArgs e) {
        ToggleBouncing(0, label1);
    }

    private void button2_Click(object sender, EventArgs e) {
        ToggleBouncing(1, label2);
    }

    private void button3_Click(object sender, EventArgs e) {
        ToggleBouncing(2, label3);
    }

    private void timer1_Tick(object sender, EventArgs e) {
        for (int i = 0; i < 3; i++) {
            if (bouncers[i] != null) {
                bouncers[i].Move();
            }
        }
    }
}
```

Each button calls the ToggleBouncing() method, passing it an index of an array and a reference to one of the Labels on the form.

Can you follow exactly what's going on with the button event handlers? Your job is to figure out how they turn the bouncing on and off for the labels.

The Timer uses a for loop to call each LabelBouncer's Move() method, but only if it's not null. Setting the element to null stops it from bouncing on the form.



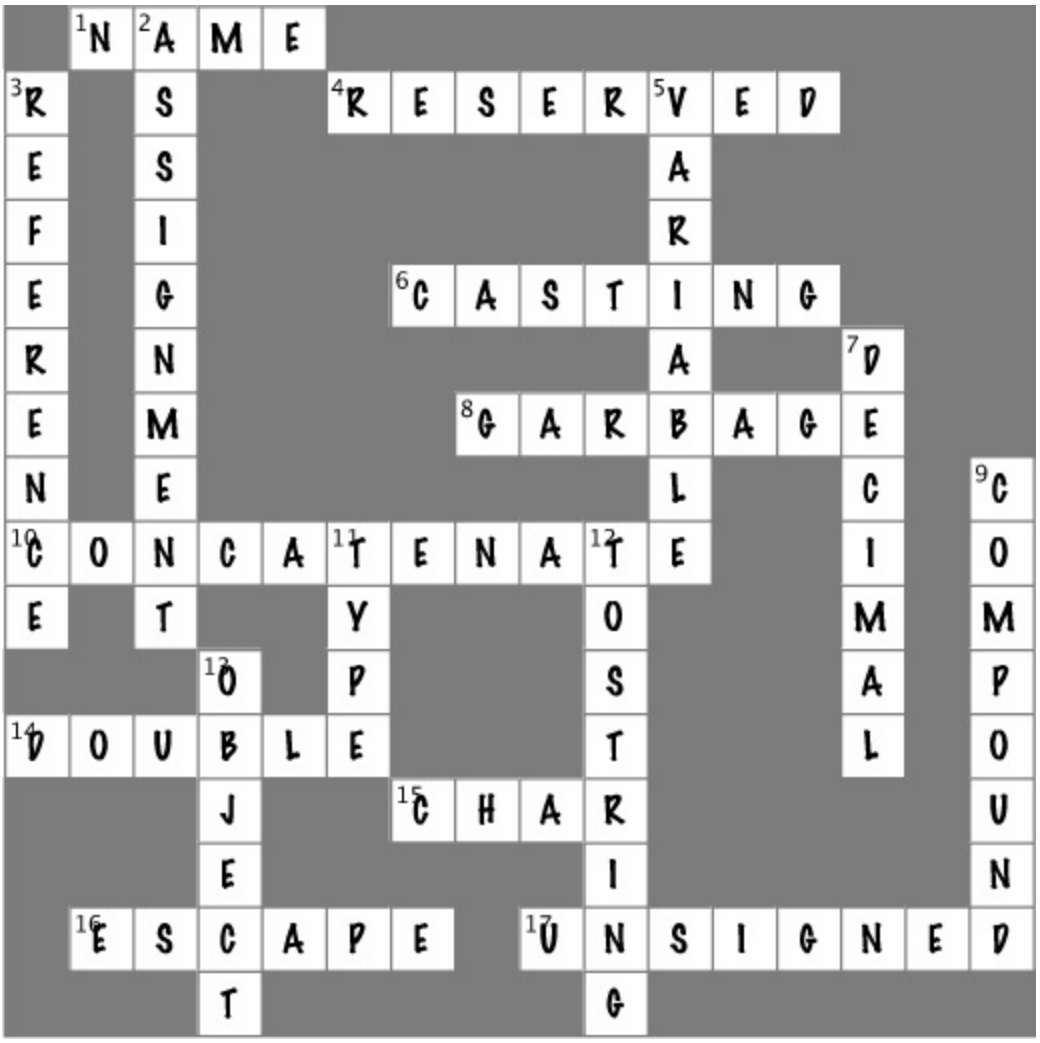
Since controls are just objects, you can pass references to them as method parameters and store them in arrays, fields, and variables.

EXERCISE SOLUTION

There are about 77 **reserved words called *keywords*** in C#. These are words reserved by the C# compiler; you can't use them for variable names. You'll know a lot of them really well by the time you finish the book. Here are some you've already used. Write down what you think these words do in C#.

namespace	Namespaces make sure that the names you are using in your program don't collide with the ones in the .NET Framework or other external classes you've used in your program. All of the classes and methods in a program are inside a namespace.
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	A class is how you define an object. Classes have properties and methods. Properties are what they know and methods are what they do.
public	A public class can be used by every other class in the project. When a variable or method is declared as public, it can be used by classes and called by methods that are outside of the one it's being declared in.
else	Code that starts with else 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. using lets you use code from the .NET Framework and predefined classes from third parties as well as classes you can make yourself.
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 in them is true.

TYPECROSS SOLUTION



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 void button1_Click(object sender, EventArgs e)
{
  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++)
  {
    if (elephants[i].EarSize > biggestEars.EarSize)
    {
      biggestEars = elephants[i];
    }
  }
  MessageBox.Show(biggestEars.EarSize.ToString());
}
```

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

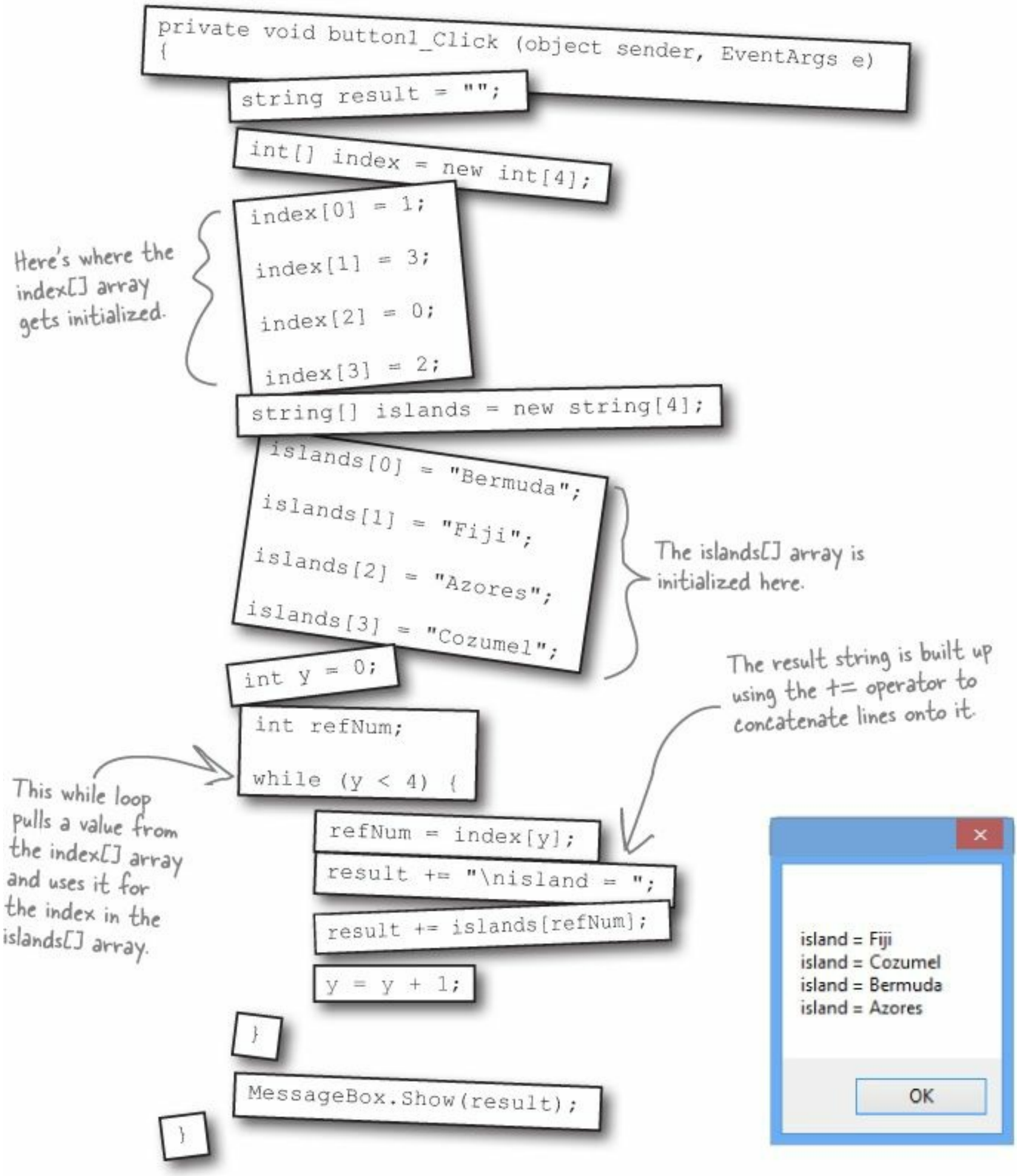
The biggestEars reference is used to keep track of which element we've seen while going through the for loop has the biggest ears so far.

Use the debugger to check this! Put your breakpoint here and watch biggestEars.EarSize.

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.

CODE MAGNETS SOLUTION

The code for a button is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working method that produces the output listed below?



POOL PUZZLE SOLUTION

Notice how this class contains the entry point, but it also creates an instance of itself? That's completely legal in C#.

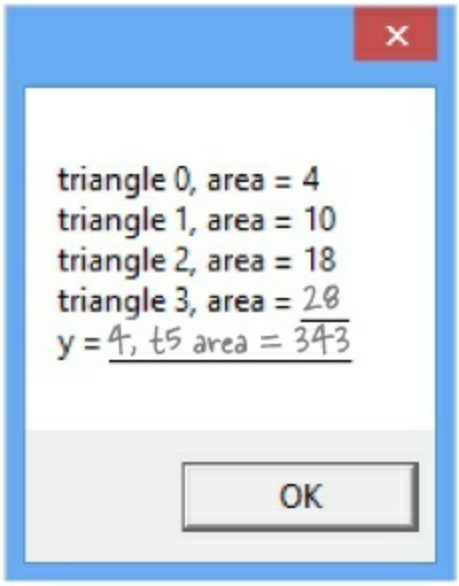
```
class Triangle
{
    double area;
    int height;
    int length;
    public static void Main(string[] args)
    {
        string results = "";
        int x = 0;
        Triangle[] ta = new Triangle[4];
        while ( x < 4 )
        {
            ta[x] = new Triangle();
            ta[x].height = (x + 1) * 2;
            ta[x].length = x + 4;
            ta[x].setArea();
            results += "triangle " + x + ", area";
            results += " = " + ta[x].area + "\n";
            x = x + 1;
        }
        int y = x;
        x = 27;
        Triangle t5 = ta[2];
        ta[2].area = 343;
        results += "y = " + y;
        MessageBox.Show(results +
            ", t5 area = " + t5.area);
    }
    void setArea()
    {
        area = (height * length) / 2;
    }
}
```

After this line, we've got an array of four Triangle references—but there aren't any Triangle objects yet!

The while loop creates the four instances of Triangle by calling the new statement four times.

The setArea() method uses the height and length fields to set the area field. Since it's not a static method, it can only be called from inside an instance of Triangle.

Bonus Answer



Part I. C# Lab: A Day at the Races

Name:	_____	Date:	_____

This lab gives you a spec that describes a program for you to build, using the knowledge you've gained over the last few chapters.

This project is bigger than the ones you've seen so far. So read the whole thing before you get started, and give yourself a little time. And don't worry if you get stuck — there's nothing new in here, so you can move on in the book and come back to the lab later.

We've filled in a few design details for you, and we've made sure you've got all the pieces you need...and nothing else.

It's up to you to finish the job. You can download the graphics files we used in our solution...but we won't give you code for a solution.

> **NOTE**
>
> But other readers have claimed their bragging rights by publishing their solutions on CodePlex, GitHub, and other collaborative source code hosting sites, in case you need a hint!

The Spec: Build a Racetrack Simulator

Joe, Bob, and Al love going to the track, but they're tired of losing all their money. They need you to build a simulator for them so they can figure out winners *before* they lay their money down. And, if you do a good job, they'll cut you in on their profits.

Here's what you're going to build for them...

The Guys

Joe, Bob, and Al want to bet on a dog race. Joe starts with 50 bucks, Bob starts with 75 bucks, and Al starts with 45 bucks. Before each race, they'll each decide if they want to bet, and how much they want to put down. The guys can change their bets right up to the start of the race...but once the race starts, all bets are final.

The Betting Parlor

The betting parlor keeps track of how much cash each guy has, and what bet he's placed. There's a minimum bet of 5 bucks. The parlor only takes one bet per person for any one race.

The parlor checks to make sure that the guy who's betting has enough cash to cover his bet — so the guys can't place a bet if they don't have the cash to cover the bet.

Betting

Every bet is double-or-nothing — either the winner doubles his money, or he loses what he bet. There's a minimum bet of 5 bucks, and each guy can bet up to 15 bucks on a single dog. If the dog wins, the bettor ends up with twice the amount that he bets (after the race is complete). If he loses, that amount disappears from his pile.

NOTE

Say a guy places a $10 bet at the window. At the end of the race, if his dog wins, his cash goes up by $10 (because he keeps the original $10 he bet, plus he gets $10 more from winning). If he loses, his cash goes down by $10.

The Race

There are four dogs that run on a straight track. The winner of the race is the first dog to cross the finish line. The race is totally random, there are no handicaps or odds, and a dog isn't more likely to win his next race based on his past performance.

NOTE

If you want to build a handicap system, by all means do it! It'll be really good practice writing some fun code.

Sound fun? We've got more details coming up...

You'll need three classes and a form

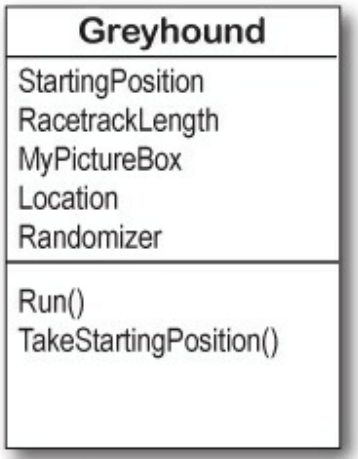
You'll build three main classes in the project, as well as a GUI for the simulator. You should have an array of three `Guy` objects to keep track of the three guys and their winnings, and an array of four `Greyhound` objects that actually run the race. Also, each instance of `Guy` should have its own `Bet` object that keeps track of his bet and pays out (or takes back) cash at the end of the race.

You'll need to add `using System.Windows.Forms;` to the top of the `Greyhound` and `Guy` classes. You'll also need to add the `public` keyword in front of each of your class declarations.

We've gotten you started with class descriptions and some snippets of code to work from. You've got to finish everything up.

We've given you the skeleton of the class you need to build. Your job is to fill in the methods.

Make sure you add public to each class declaration.



See how the class diagram matches up with the code?

```
public class Greyhound {
  public int StartingPosition; // Where my PictureBox starts
  public int RacetrackLength; // How long the racetrack is
  public PictureBox MyPictureBox = null; // My PictureBox object
  public int Location = 0; // My Location on the racetrack
  public Random Randomizer; // An instance of Random

  public bool Run() {
    // Move forward either 1, 2, 3 or 4 spaces at random
    // Update the position of my PictureBox on the form like this:
    //    MyPictureBox.Left = StartingPosition + Location;
    // Return true if I won the race
  }

  public void TakeStartingPosition() {
    // Reset my location to 0 and my PictureBox to starting position
  }
}
```

You only need one instance of Random—each Greyhound's Randomizer reference should point to the same Random object.

We've added comments to give you an idea of what to do.

Don't overthink this... sometimes you just need to set two fields, and you're done.

Initialize your arrays of Greyhound and Guy objects

The `Greyhound` class keeps track of its position on the racetrack during the race, and it updates the location of the `PictureBox` representing the dog to move down the race track. Each instance of `Greyhound` uses a field called `MyPictureBox` to reference the `PictureBox` control on the form that shows the picture of the dog. It also needs to know its starting position and the length of the racetrack, which it can determine using the `PictureBox` for the racetrack (we named it `racetrackPictureBox`). Here's the object initializer for one of the Greyhound objects in the array (we called it `GreyhoundArray`):

> **NOTE**
>
> This works just like LabelBouncer: the form passes a reference to a PictureBox to the Greyhound object, which uses its Left property to make it move.

```
GreyhoundArray[0] = new Greyhound() {
    MyPictureBox = pictureBox1,
    StartingPosition = pictureBox1.Left,
    RacetrackLength = racetrackPictureBox.Width - pictureBox1.Width,
    Randomizer = MyRandomizer
};
```

This Greyhound object controls pictureBox1.

You'll need to do this for each object in the array of Greyhounds. You'll also need to initialize your three Guy objects. Don't forget to set each guy's MyRadioButton and MyLabel to the right control!

```
public class Guy {
   public string Name; // The guy's name
   public Bet MyBet; // An instance of Bet that has his bet
   public int Cash; // How much cash he has

   // The last two fields are the guy's GUI controls on the form
   public RadioButton MyRadioButton; // My RadioButton
   public Label MyLabel; // My Label

   public void UpdateLabels() {
     // Set my label to my bet's description, and the label on my
     // radio button to show my cash ("Joe has 43 bucks")
   }

   public void ClearBet() {} // Reset my bet so it's zero

   public bool PlaceBet(int BetAmount, int DogToWin) {
     // Place a new bet and store it in my bet field
     // Return true if the guy had enough money to bet
   }

   public void Collect(int Winner) {
     // Ask my bet to pay out, clear my bet, and update my labels
   }
```

This works exactly like the MyLabel field in LabelBouncer from Chapter 4.

Once you set MyLabel to one of the labels on the form, you'll be able to change the label's text using MyLabel.Text. The same goes for MyRadioButton.

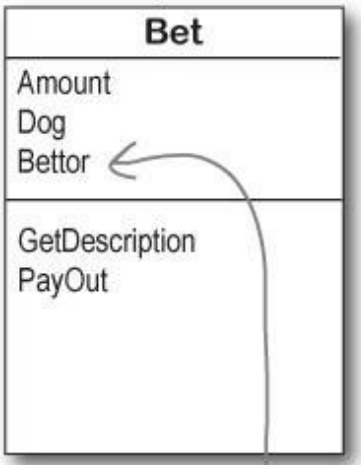
When you initialize the Guy object, make sure you set its MyBet field to null, and call its UpdateLabels() method as soon as it's initialized.

Add your code here.

Remember that bets are represented by instances of Bet.

This is the object that Guy uses to represent bets in the application.

The key here is to use the Bet object... let it do the work.



The object initializer for Bet just sets the amount, dog, and bettor.

```
public class Bet {
   public int Amount; // The amount of cash that was bet
   public int Dog; // The number of the dog the bet is on
   public Guy Bettor; // The guy who placed the bet

   public string GetDescription() {
     // Return a string that says who placed the bet, how much
     // cash was bet, and which dog he bet on ("Joe bets 8 on
     // dog #4"). If the amount is zero, no bet was placed
     // ("Joe hasn't placed a bet").
   }

   public int PayOut(int Winner) {
     // The parameter is the winner of the race. If the dog won,
     // return the amount bet. Otherwise, return the negative of
     // the amount bet.
   }
}
```

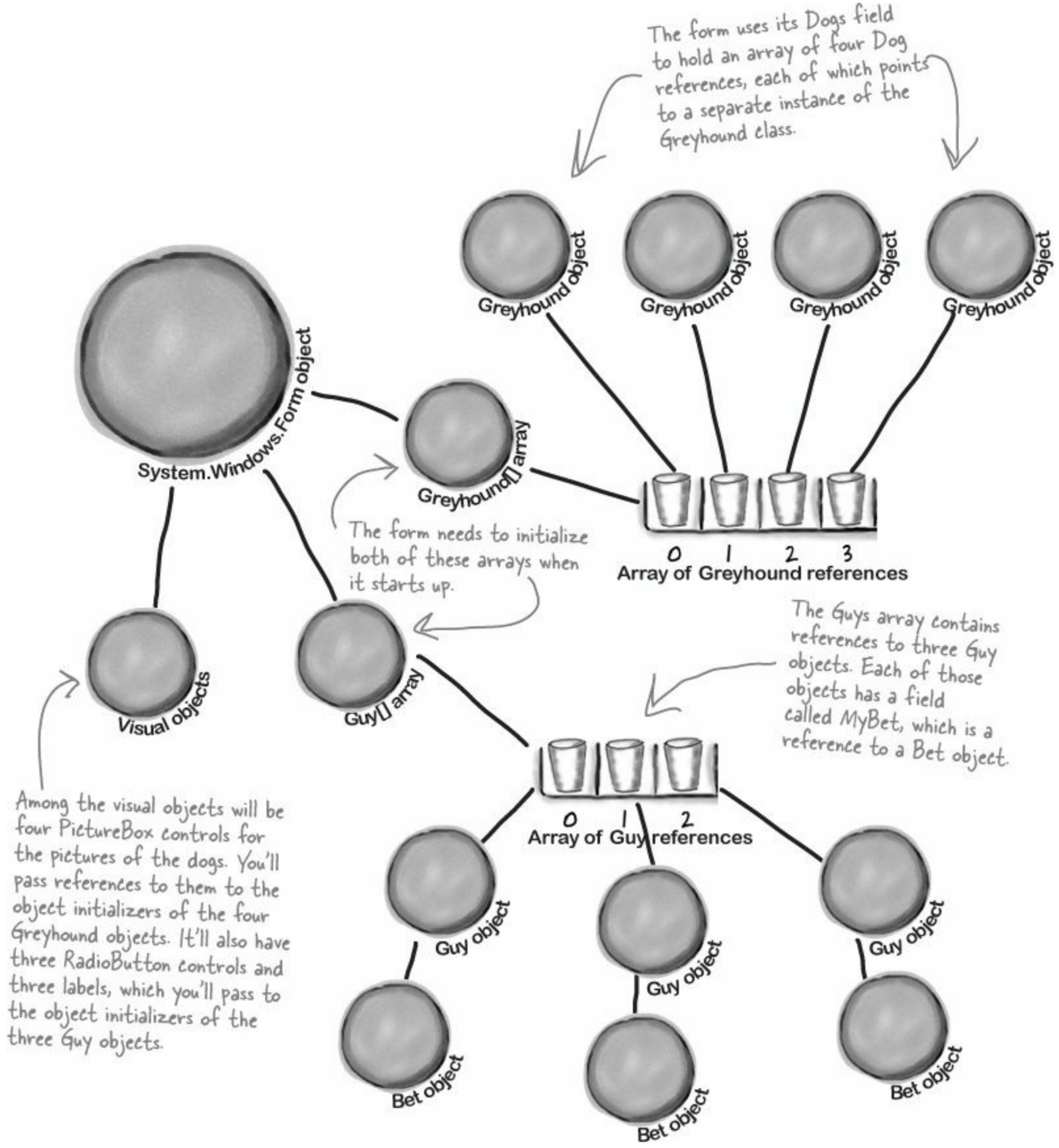
This is a common programming task: assembling a string or message from several individual bits of data.

Hint: you'll instantiate Bet in the Guy code. Guy will use the this keyword to pass a reference to himself to the Bet's initializer.

Remember: the form keeps the dogs in an array that starts at index 0. Dog #1 is at index 0, dog #2 is at index 1, etc. You'll need to add 1 to the array index to get the winner.

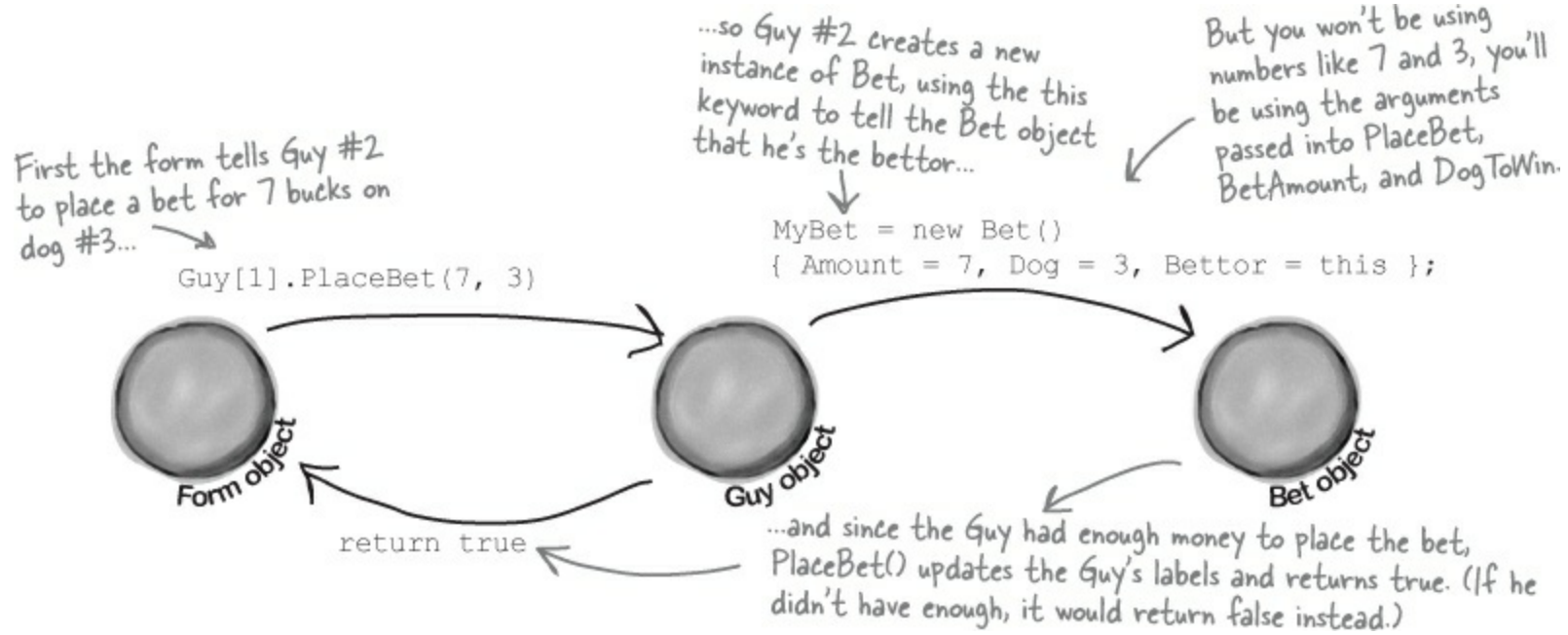
Here's your application architecture

Spend some time looking closely at the architecture. It looks pretty complicated at first, but there's nothing here you don't know. Your job is to recreate this architecture yourself, starting with the `Greyhound` and `Guy` arrays in your main form.

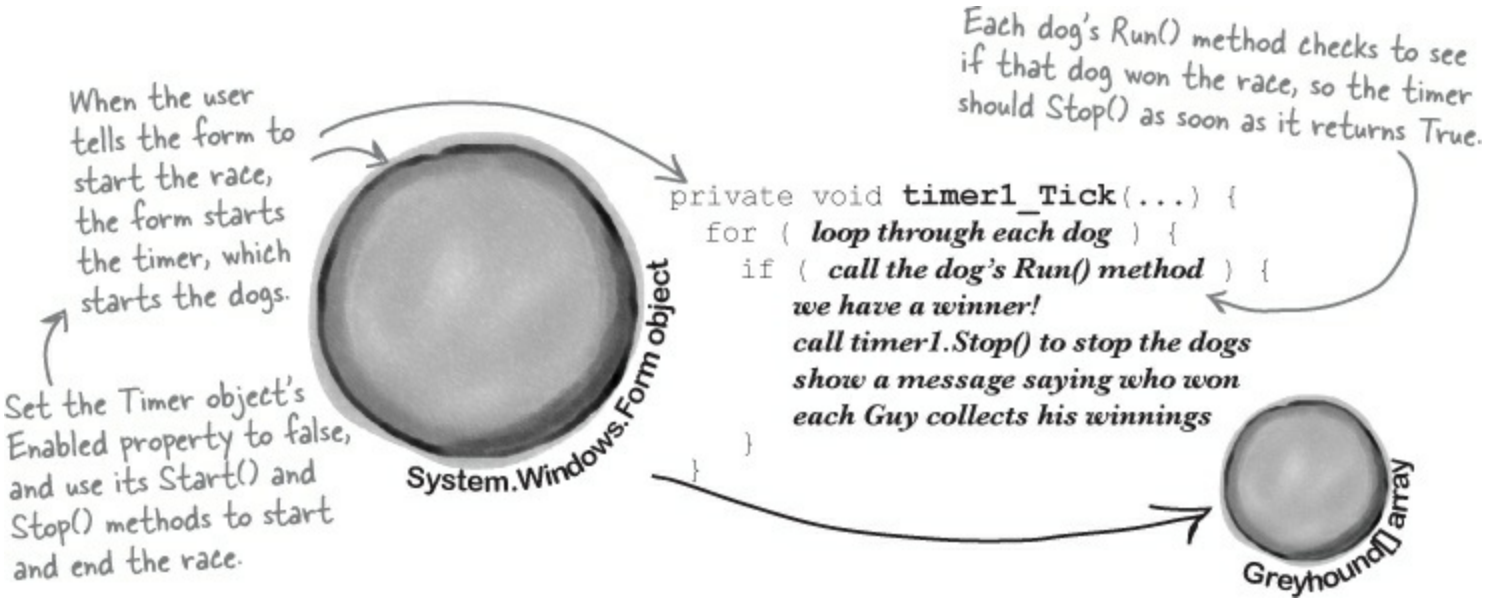


If your code won't build because of an error message about "inconsistent accessibility," make sure you added `public` to the beginning of the three class declarations. (You'll learn more about this later on in the book.)

When a Guy places a bet, he creates a new Bet object



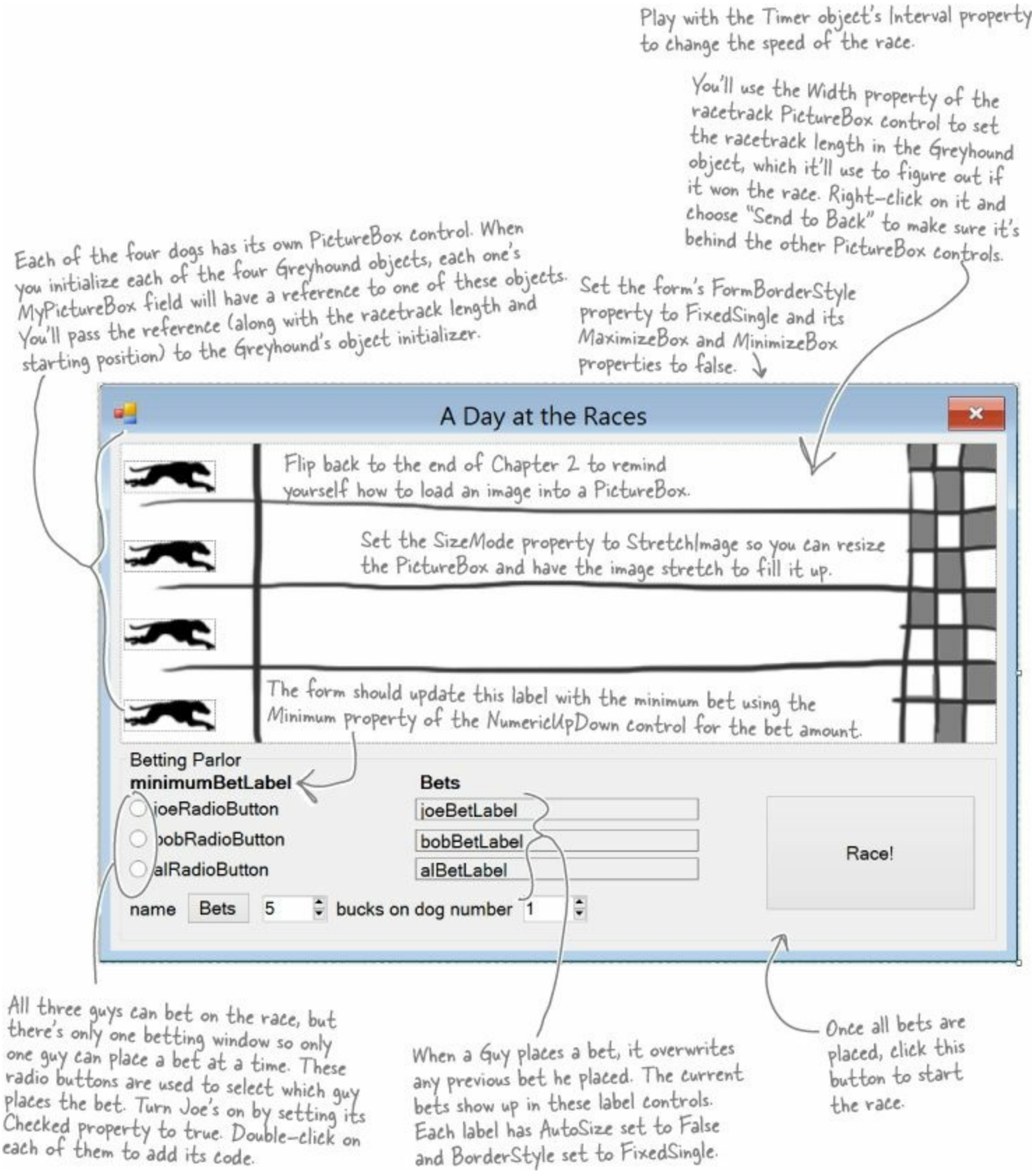
The form uses a Timer to keep the dogs running until there's a winner



The Bet object figures out if it should pay out

Here's what your GUI should look like

The graphical user interface for the "Day at the Races" application consists of a form that's divided into two sections. The top is the racetrack: a `PictureBox` control for the track, and four more for the dogs. The bottom half of the form shows the betting parlor, where three guys (Joe, Bob, and Al) can bet on the outcome of the race.



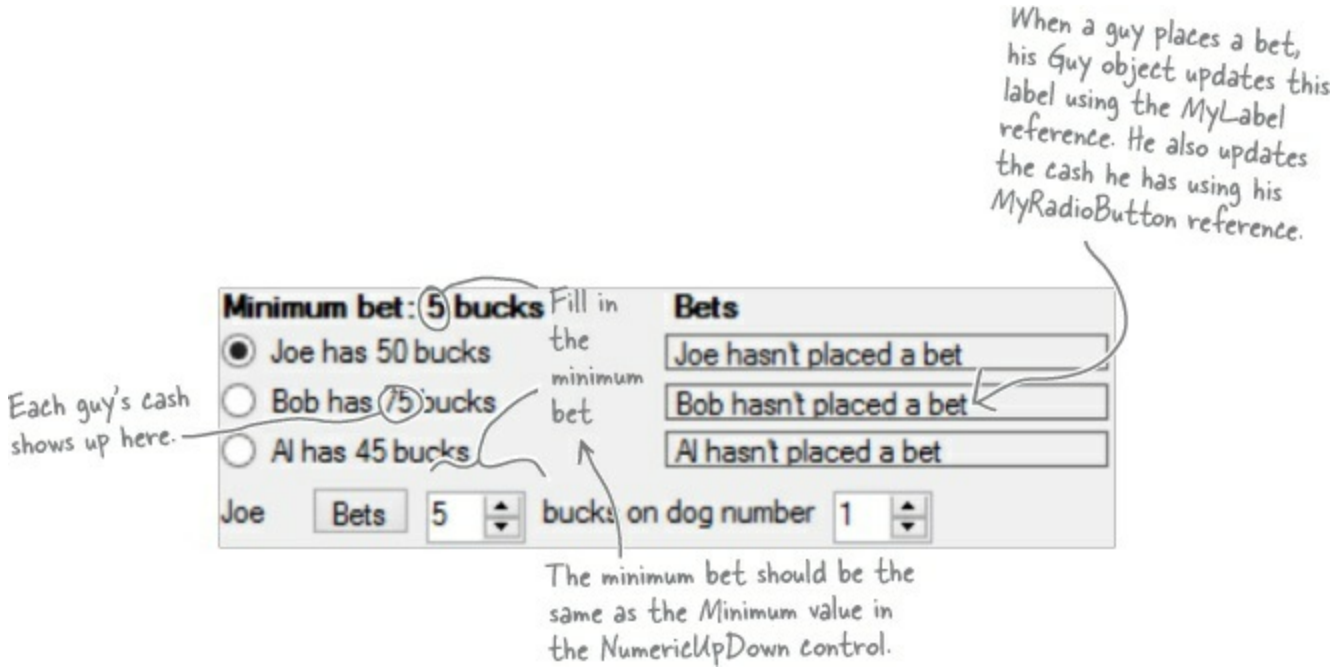
You can download the graphics files from www.headfirstlabs.com/books/hfcsharp/.

Placing bets

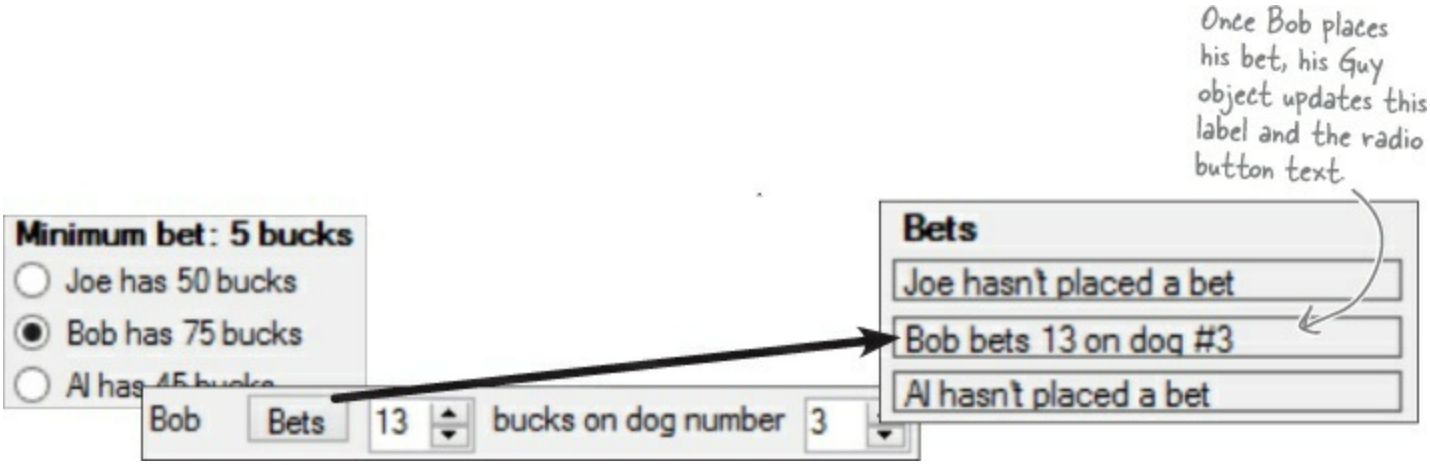
Use the controls in the Betting Parlor GroupBox to place each guy's bet. There are three distinct stages here:

- **❶ No bets have been placed yet.**
  When the program first starts up, or if a race has just finished, no bets have been placed in the betting parlor. You'll see each guy's total cash next to his name on the left.

> **NOTE**
>
> You'll need a loop to initialize each Guy object by calling his ClearBet() method (which has him place a bet with zero bucks) and then calling his UpdateLabels() method.



- **❷ Each guy places his bets.**
  To place a bet, select the guy's radio button, select an amount and a dog, and click the Bets button. His `PlaceBet()` method will update the label and radio button.

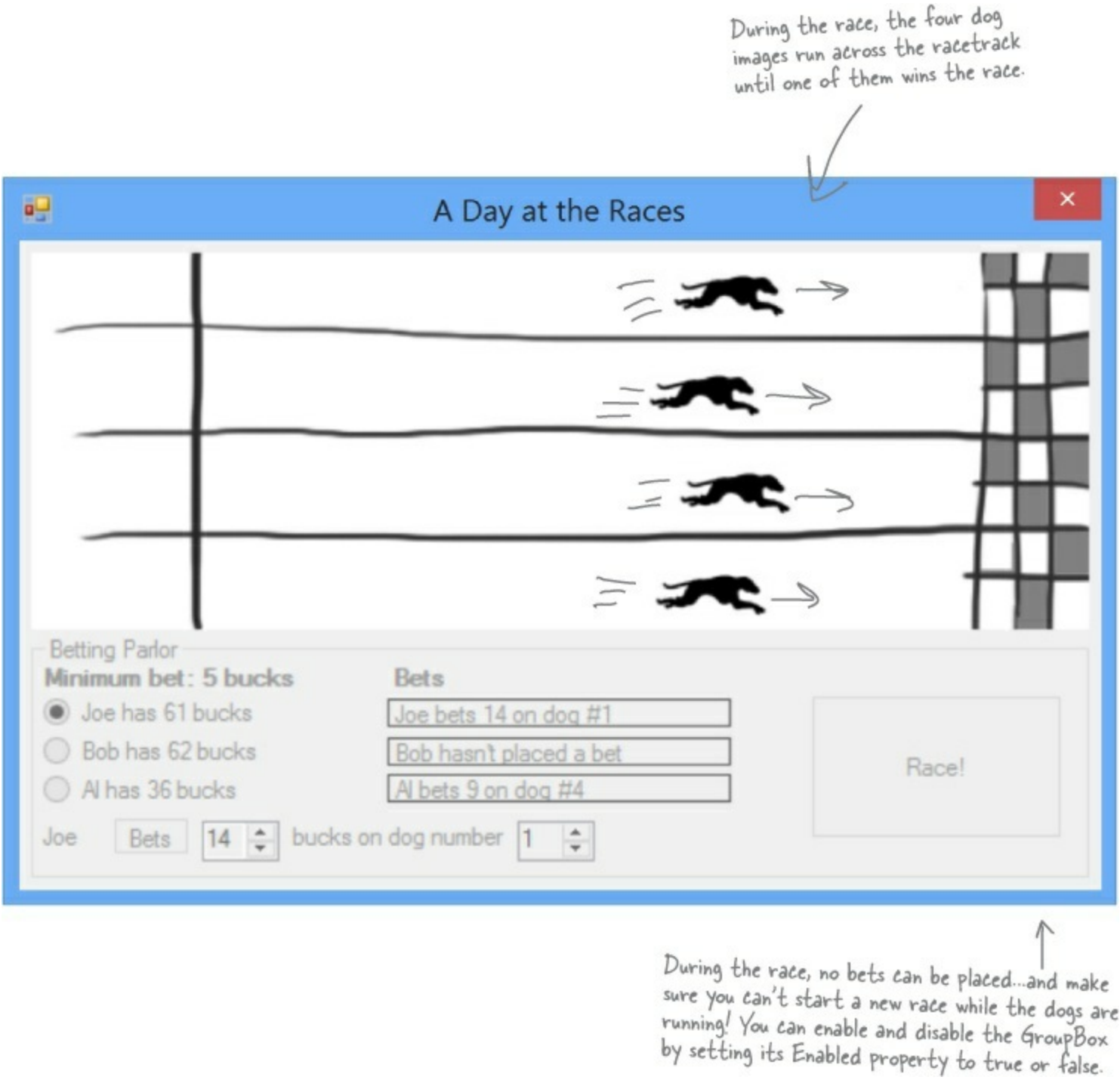


- **❸ After the race, each guy collects his winnings (or pays up!).**
  Once the race is complete and there's a winner, each `Guy` object calls his `Collect()` method and adds his winnings or losses to his cash.

Make sure all the Greyhound objects share <u>the same</u> Random object! If each dog creates its own new instance of Random, you might see a bug where all of the dogs generate the same sequence of random numbers.

The Finished Product

You'll know your "Day at the Races" application is done when your guys can place their bets and watch the dogs race.



You can download a finished executable, as well as the graphics files for the four dogs and the racetrack, from the Head First labs website:
www.headfirstlabs.com/books/hfcsharp

We didn't give solutions for this lab because when programs get large enough, there are too many ways to build them for us to say there's one "right" solution. But if you need a hint, plenty of people have claimed their bragging rights by publishing their own code on CodePlex.com and other collaborative source code hosting sites.

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**. You'll **make your object's data private**, and add methods to **protect how that data is accessed**.

Kathleen is an event planner

She's been planning dinner parties for her clients and she's doing really well. But lately she's been having a hard time responding to clients fast enough with an estimate for her services.

When a new client calls Kathleen to do a party, she needs to find out the number of guests, what kind of drinks to serve, and what decorations she should buy. Then she uses a pretty complicated calculation to figure out the total cost, based on a flow chart she's been using for years. The bad news is that it takes her a long time to work through her chart, and while she's estimating, her potential clients are checking out other event planners.

It's up to you to build her a C#-driven event estimator and save her business. Imagine the party she'll throw you when you succeed!

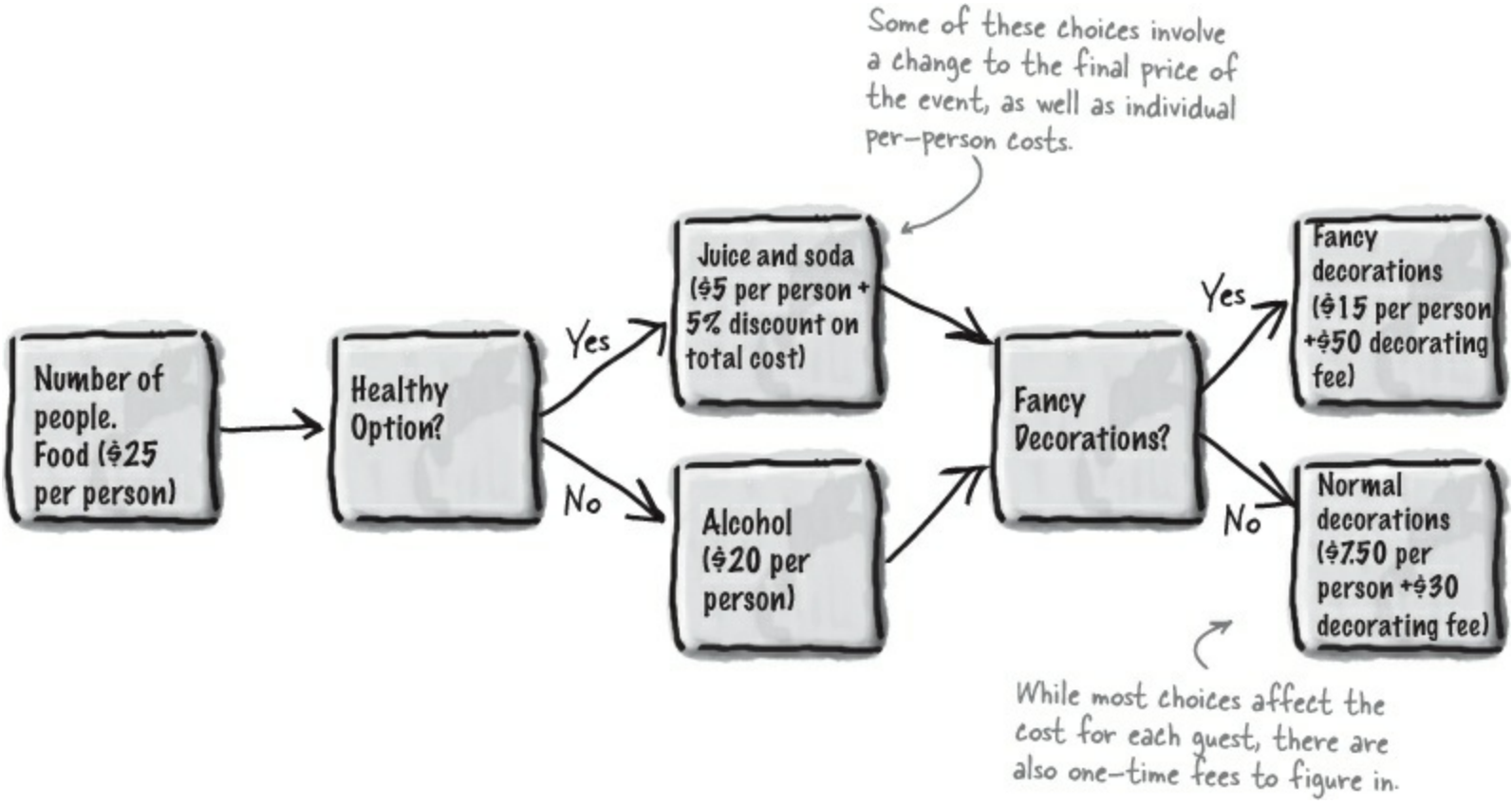
What does the estimator do?

Kathleen runs down some of the basics of her system for figuring out the costs of an event. Here's part of what she came up with:

KATHLEEN'S PARTY PLANNING PROGRAM — COST ESTIMATE FOR A DINNER PARTY

- For each person on the guest list there's a $25 food charge.
- Clients have a choice when it comes to drinks. Most parties serve alcohol, which costs $20 per person. But they can also choose to have a party without alcohol. Kathleen calls that the "Healthy Option," and it only costs $5 per person to have soda and juice instead of alcohol. Choosing the Healthy Option is a lot easier for her, so she gives the client a 5% discount on the entire party, too.
- There are two options for the cost of decorations. If a client goes with the normal decorations, it's $7.50 per person with a $30 decorating fee. A client can also upgrade the party decorations to the "Fancy Decorations" — that costs $15 per person with a $50 one-time decorating fee.

Here's another look at this same set of costs, broken down into a little flow chart to help you see how it works:

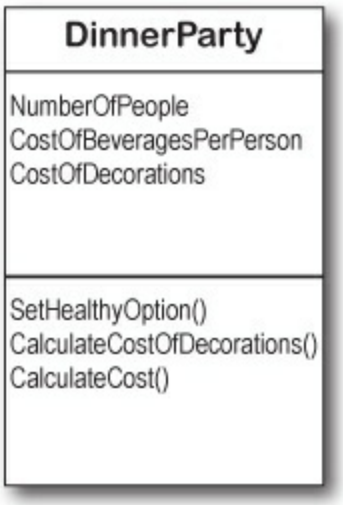


You're going to build a program for Kathleen

When you flip the page, you'll see an exercise to build a dinner party–planning program for Kathleen. Here's a sneak preview of what you'll build.

You'll build this form, which Kathleen will use to set the options for her party. She'll set the number of people and check or uncheck the boxes for fancy decorations or a healthy option. As she does, the cost at the bottom will change based on her selections.

The logic for the program will be built into a class called DinnerParty. The form will create a DinnerParty object, store a reference to that object in a field, and use its fields and methods to perform the calculation.

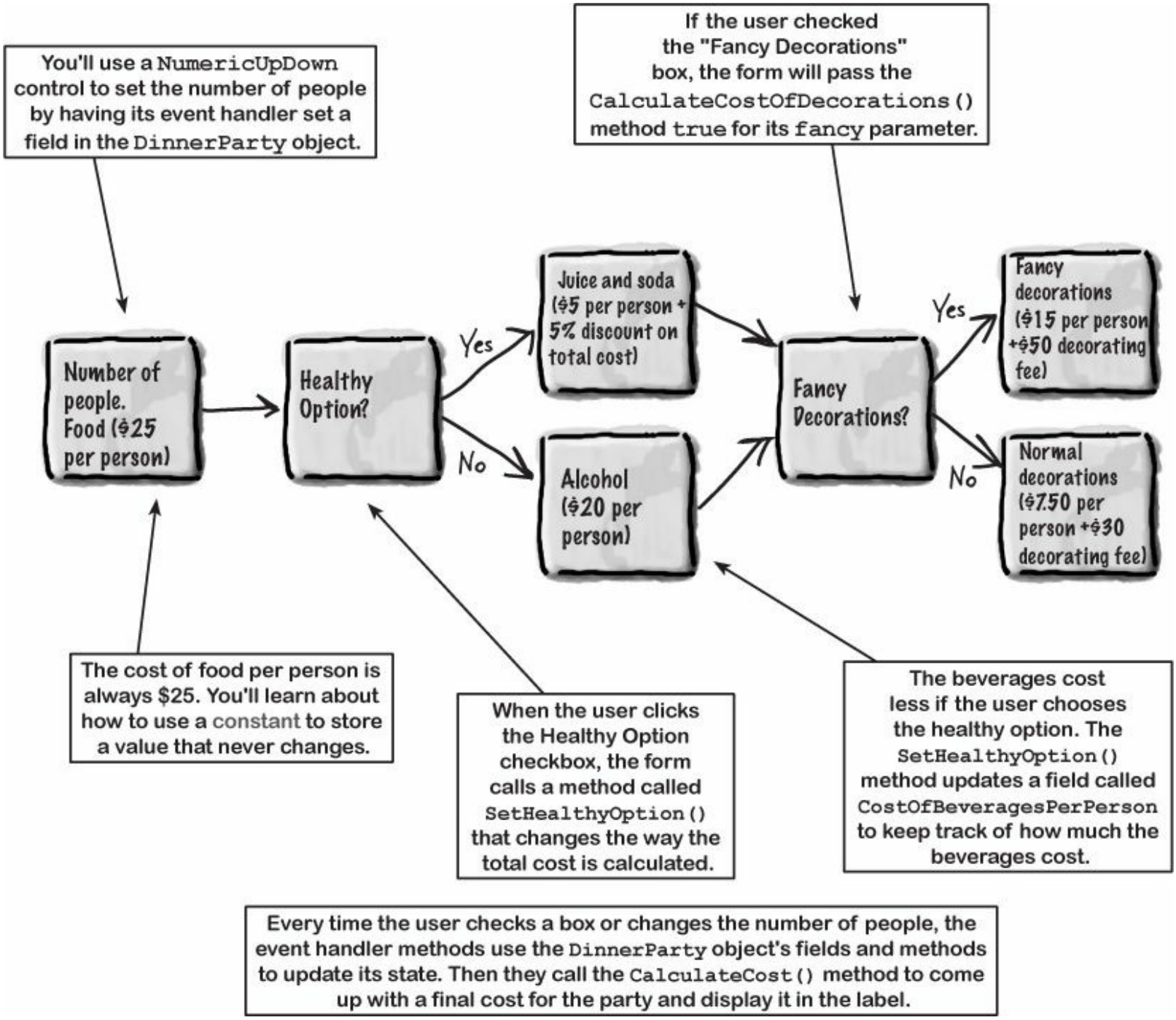


Here's what the top of the form will look like. It will have a field called `dinnerParty` to do the cost calculation. The first thing the form will do is set it up with default values, and then calculate the cost using a method called `DisplayDinnerPartyCost()`. The form will call that method every time the user changes an option.

```
public partial class Form1 : Form
{
    DinnerParty dinnerParty;

    public Form1()
    {
        InitializeComponent();
        dinnerParty = new DinnerParty() { NumberOfPeople = 5 };
        dinnerParty.SetHealthyOption(false);
        dinnerParty.CalculateCostOfDecorations(true);
        DisplayDinnerPartyCost();
    }
    ...
```

Here's how the `DinnerParty` class will work. The current state of the `DinnerParty` object —

the values stored in its fields — determines how it does its cost calculation. Setting the healthy option, choosing fancy decorations, and adding or removing people changes the state of the object, which causes the `CalculateCost()` method to return a different number.



Got all that? Let's start building!

We're going to start asking you to solve longer and tougher problems because we know you're up to the challenge!

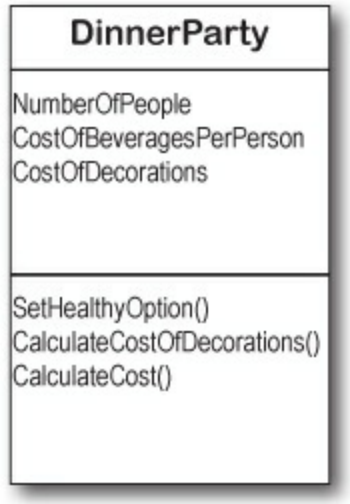
NOTE

The goal is to help you become a great C# programmer, and the quickest way to that goal is solving problems like this one.

EXERCISE

Build a program to solve Kathleen's party estimating problem.

- ❶ Create a new Windows Forms Application project, add a class file to it called *DinnerParty. cs*, and build the `DinnerParty` class using the class diagram to the left. It's got three methods: `CalculateCostOfDecorations()`, `SetHealthyOption()`, and `CalculateCost()`. For the fields, use `decimal` for the two costs, and an `int` for the number of people. Make sure you **add an M after every literal** you assign to a decimal value (`10.0M`).



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

- ❷ Here's a useful C# tool. Since the cost of food won't be changed by the program, you can declare it as a ***constant***, which is like a variable except that its value can never be changed. Here's the declaration to use:

```
public const int CostOfFoodPerPerson = 25;
```

- ❸ Flip back to the previous page to be sure you've got the calculations right for the methods. Only one of them returns a value (a `decimal`) — the other two are `void`. The `CalculateCostOfDecorations()` method figures out the cost of decorations for the number of people attending the party. Use the `CalculateCost()` method to figure out the total cost by adding the cost of the decorations to the cost of drinks and food per person. If the client wants the healthy option, you can apply the discount inside the `CalculateCost()` method after you've figured out the total cost.
- ❹ Add this code to your form:

You'll declare the dinnerParty field in the form, and then add these four lines below InitializeComponent().

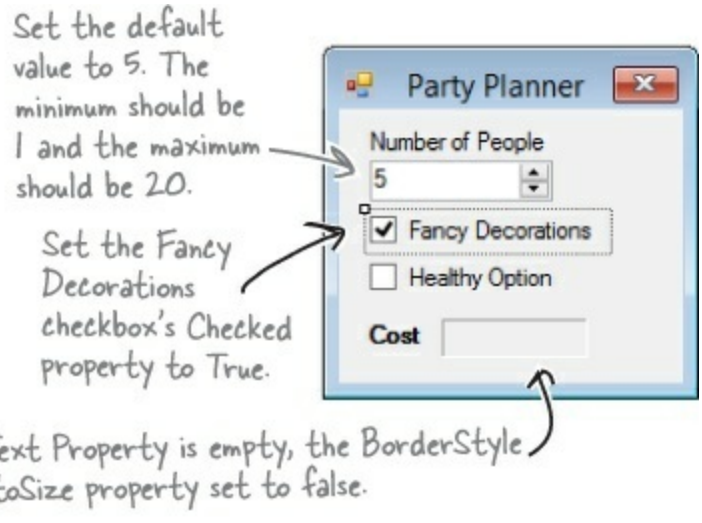
```
DinnerParty dinnerParty;
public Form1() {
  InitializeComponent();
  dinnerParty = new DinnerParty() { NumberOfPeople = 5 };
  dinnerParty.SetHealthyOption(false);
  dinnerParty.CalculateCostOfDecorations(true);
  DisplayDinnerPartyCost();
}
```

The SetHealthyOption() method uses a bool parameter (healthyOption) to update the CostOfBeveragesPerPerson field based on whether or not the client wants the healthy option.

- ❺ Here's what the form should look like. Use the `NumericUpDown` control's properties to set the maximum number of people to 20, the minimum to 1, and the default to 5. Get rid of the maximize and minimize buttons, too.

NOTE

The checkboxes are named fancyBox and healthyBox. You can keep the default name for the NumericUpDown control.



- ❻ Instead of using a button to calculate the costs, this form will update the cost label automatically as soon as you use a checkbox or the `NumericUpDown` control. The first thing you need to do is create a method in the form that displays the cost. Add this method to the Form1 class. It'll get called when the `NumericUpDown` control is clicked:

This method will get called by all of the other methods you create on the form. It's how you update the cost label with the right value whenever anything changes.

Add this method to the form—it'll recalculate the cost of the party and put it in the Cost label.

```
private void DisplayDinnerPartyCost()
{
    decimal Cost = dinnerParty.CalculateCost(checkBox2.Checked);
    costLabel.Text = Cost.ToString("c");
}
```

Change the name of the label that displays the cost to costLabel.

Passing "c" to ToString() tells it to format the cost as a currency value. If you're in a country that uses dollars, it'll add a dollar sign.

This is true if the checkbox for the Healthy Option is checked.

- ❼ Now hook up the `NumericUpDown` field to the `NumberOfPeople` variable you created in the `DinnerParty` class and display the cost in the form. Double-click on the `NumericUpDown` control — the IDE will ***add an event handler method*** that gets run every time the value in the control is changed. Use this method to reset the number of people in the party. Here's the code for the method:

> **NOTE**
>
> You've been using event handlers all along — when you double-click on a button, the IDE adds a Click event handler. Now you know what it's called.

```
private void numericUpDown1_ValueChanged(
                                object sender, EventArgs e)
{
    dinnerParty.NumberOfPeople = (int) numericUpDown1.Value;
    DisplayDinnerPartyCost();
}
```

You need to cast numericUpDown.Value to an int because it's a Decimal property.

Uh oh — there's a problem with this code. Can you spot it? Don't worry if you don't see it just yet.

NOTE

The value you send from the form to the method will be fancyBox.Checked. That will be passed as a boolean parameter to the method in the class.

- ❽ Double-click on the Fancy Decorations checkbox on the form and make sure that it first calls `CalculateCostOfDecorations()` and then `DisplayDinnerPartyCost()`. Next, double-click the Healthy Option checkbox and make sure that it calls the `SetHealthyOption()` method in the `DinnerParty` class and then calls the `DisplayDinnerPartyCost()` method.

NOTE

These are just two-line methods. The first line will call the method you created in the class to figure out the costs, and the second will display the total cost on the form.

EXERCISE SOLUTION

Here's the code that goes into *DinnerParty.cs*.

```
class DinnerParty {
    public const int CostOfFoodPerPerson = 25;
    public int NumberOfPeople;
    public decimal CostOfBeveragesPerPerson;
    public decimal CostOfDecorations = 0;

    public void SetHealthyOption(bool healthyOption) {
        if (healthyOption) {
            CostOfBeveragesPerPerson = 5.00M;
        } else {
            CostOfBeveragesPerPerson = 20.00M;
        }
    }

    public void CalculateCostOfDecorations(bool fancy) {
        if (fancy)
        {
            CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        } else {
            CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;
        }
    }
    public decimal CalculateCost(bool healthyOption) {
        decimal totalCost = CostOfDecorations +
                ((CostOfBeveragesPerPerson + CostOfFoodPerPerson)
                    * NumberOfPeople);

        if (healthyOption) {
            return totalCost * .95M;
        } else {
            return totalCost;
        }
    }
}
```

Using a constant for CostOfFoodPerPerson ensures the value can't be changed. It also makes the code easier to read—it's clear that this value never changes.

When the form first creates the object, it uses the initializer to set NumberOfPeople. Then it calls SetHealthyOption() and CalculateCostOfDecorations() to set the other fields.

We used "if (Fancy)" instead of typing "if (Fancy == true)" because the if statement always checks if the condition is true.

We used parentheses to make sure the math works out properly.

This applies the 5% discount to the overall event cost if the nonalcoholic option was chosen.

NOTE

You don't need to add "using System.Windows.Forms;" to your DinnerParty class, because it doesn't use MessageBox.Show() or anything else from that .NET Framework namespace.

We had you use a `decimal` for the prices because it's designed for monetary values. Just make sure you always put an "M" after every literal — so if you want to store $35.26, make sure you write `35.26M`. You can remember this because the M stands for *Money*!

```
public partial class Form1 : Form {
    DinnerParty dinnerParty;
    public Form1() {
        InitializeComponent();
        dinnerParty = new DinnerParty() { NumberOfPeople = 5 };
        dinnerParty.CalculateCostOfDecorations(fancyBox.Checked);
        dinnerParty.SetHealthyOption(healthyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void fancyBox_CheckedChanged(object sender, EventArgs e) {
        dinnerParty.CalculateCostOfDecorations(fancyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void healthyBox_CheckedChanged(object sender, EventArgs e) {
        dinnerParty.SetHealthyOption(healthyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void numericUpDown1_ValueChanged(object sender, EventArgs e) {
        dinnerParty.NumberOfPeople = (int)numericUpDown1.Value;
        DisplayDinnerPartyCost();
    }

    private void DisplayDinnerPartyCost() {
        decimal Cost = dinnerParty.CalculateCost(healthyBox.Checked);
        costLabel.Text = Cost.ToString("c");
    }
}
```

We call DisplayDinnerPartyCost to initialize the label that shows the cost as soon as the form's loaded.

Changes to the checkboxes on the form set the healthyOption and Fancy booleans to true or false in the SetHealthyOption() and CalculateCostOfDecorations() methods.

We **named our checkboxes** "healthyBox" and "fancyBox" so you could see what's going on in their event handler methods.

The new dinner party cost needs to be recalculated and displayed any time the number changes or the checkboxes are checked.

STRING FORMATTING

You've already seen how you can convert any object to a string using its ToString() method. If you pass "c" to ToString(), it converts it to the local currency. You can also pass it "f3" to format it as a decimal number with three decimal places, "0" (that's a zero) to convert it to a whole number, "0%" for a whole number percentage, and "n" to display it as a number with a comma separator for thousands. Take a minute and see how each of these looks in your program!

Kathleen's test drive

Rob (on phone): Hi, Kathleen. How are the arrangements for my dinner party going?

> **NOTE**
>
> Rob's one of Kathleen's favorite clients. She did his wedding last year, and now she's planning an important dinner party for him.

Kathleen: Just great. We were out looking at decorations this morning and I think you'll love the way the party's going to look.

Rob: That's awesome. Listen, we just got a call from my wife's aunt. She and her husband are going to be visiting for the next couple of weeks. Can you tell me what it does to the estimate to move from 10 to 12 people on the guest list?

> **NOTE**
>
> When you start the program, the Fancy Decorations box should already be checked because you set its Checked property to true. Setting the number of people to 10 gives a cost of $575.

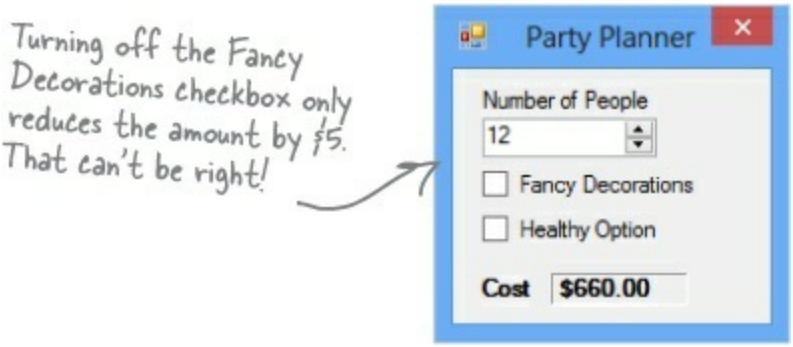
Kathleen: Sure! I'll have that for you in just one minute.

We took this screenshot in the United States, so we saw a dollar sign. If you're in the United Kingdom, France, or Japan, you'll see a sign for the pound, euro, or yen because you're using ToString("c") to convert the decimal cost to a currency string.

Changing the Number of People value from 10 to 12 and hitting enter shows $665 as the total cost. Hmm, that seems a little low....

Kathleen: OK. It looks like the total cost for the dinner will go from $575 to $665.

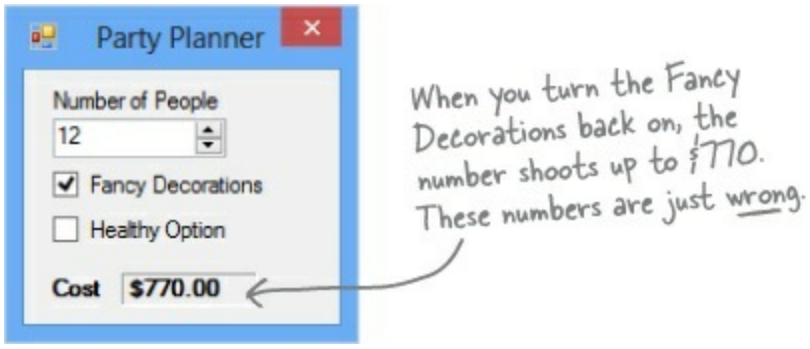
Rob: Only $90 difference? That sounds like a great deal! What if we decide to cut the fancy decorations? What's the cost then?



Kathleen: Um, it looks like...um, $660.

Rob: $660? I thought the decorations were $15 per person. Did you change your pricing or something? If it's only $5 difference, we might as well go with the fancy decorations. I've gotta tell you though, this pricing is confusing.

Kathleen: We just had this new program written to do the estimation for us. But it looks like there might be a problem. Just one second while I add the fancy decorations back to the bill.



Kathleen: Rob, I think there's been a mistake. It looks like the cost with the fancy decorations just shot up to $770. That does seem to make more sense. But I am beginning not to trust this application. I'm going to send it back for some bug fixes and work up your estimate by hand. Can I get back to you tomorrow?

Rob: I am not paying $770 just to add two people to the party. The price you quoted me before was a lot more reasonable. I'll pay you the $665 you quoted me in the first place, but I just can't go higher than that!

BRAIN POWER

Why do you think the numbers are coming out wrong every time Kathleen makes a change?

Each option should be calculated individually

Even though we made sure to calculate all of the amounts according to what Kathleen said, we didn't think about what would happen when people made changes to just one of the options on the form.

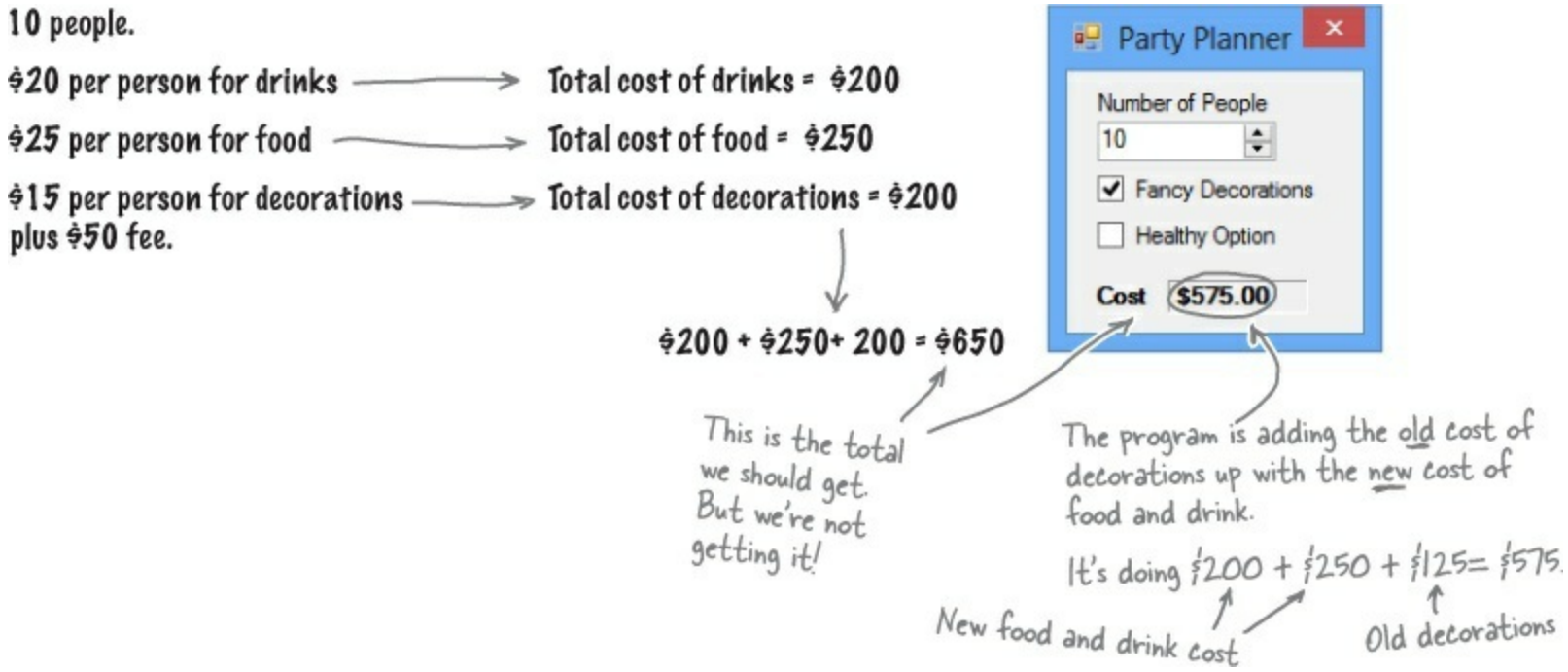
RELAX

Don't worry! This one wasn't your fault.

We built a nasty little bug into the code we gave you to show you just how easy it is to have problems with how objects use one another's fields...and just how hard those problems are to spot.

When you launch the program, the form sets the number of people to 5 and Fancy Decorations to true. It leaves Healthy Option unchecked and it calculates the cost of the dinner party as $350. Here's how it comes up with the initial total cost:

When you change the number of guests, the application should recalculate the total estimate the same way. But it doesn't:



Uncheck the Fancy Decorations checkbox and then check it again.

This will cause the `DinnerParty` object's `CostOfDecorations` field to be updated, and then the correct cost of $650 will show up.

THE PROBLEM UP CLOSE

Take a look at the method that handles changes to the value in the `numericUpDown` control. It sets the value from the field to the `NumberofPeople` variable and then calls the `DisplayDinnerPartyCost()` method. Then it counts on that method to handle recalculating all the individual new costs.

```
private void numericUpDown1_ValueChanged(
                            object sender, EventArgs e) {
    dinnerParty.NumberOfPeople = (int)numericUpDown1.Value;
    DisplayDinnerPartyCost();
}
```

This line sets the value of NumberofPeople in this instance of DinnerParty to the value in the form.

This method calls the CalculateCost() method, but not the CalculateCostofDecorations() method.

So, when you make a change to the value in the `NumberofPeople` field, this method never gets called:

```
public void CalculateCostOfDecorations(bool Fancy) {
    if (Fancy) {
        CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;
    } else {
        CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;
    }
}
```

This variable is set to $125 from when the form first called it, and since this method doesn't get called again, it doesn't change.

That's why the number corrects itself when you turn Fancy Decorations back on. Clicking the checkbox makes the program run CalculateCostOf Decorations() again.

This isn't the only part of the program that has problems, either. The two checkboxes are **inconsistent** in how they behave: one calls a method to set the object's state, and the other is passed as an argument to a method. A programmer trying to figure out how this program works will find it ***totally counterintuitive***!

NOTE

Did you have a bit of trouble figuring out how this exercise works? Don't be hard on yourself if you did. It could be because we asked you to build a program that had these conceptual problems! You'll build a much better, simpler version at the end of this chapter.

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

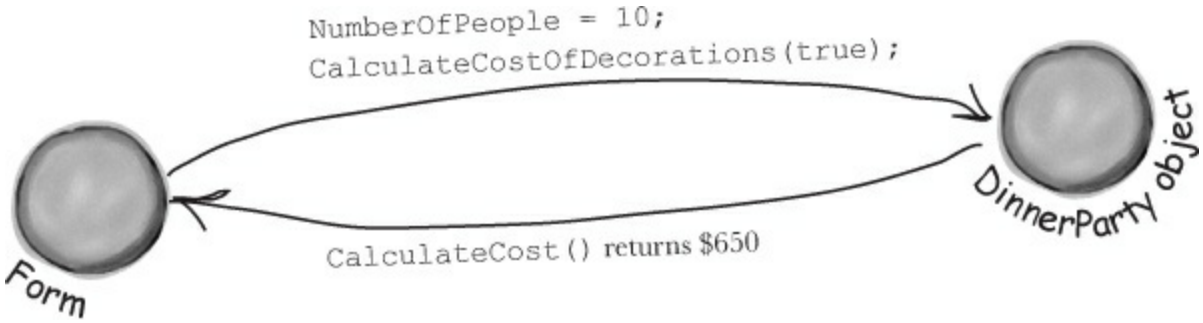
> **NOTE**
>
> ...and sometimes those “people” who are using your classes are you! You might be writing a class today that you’ll be using tomorrow.

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.

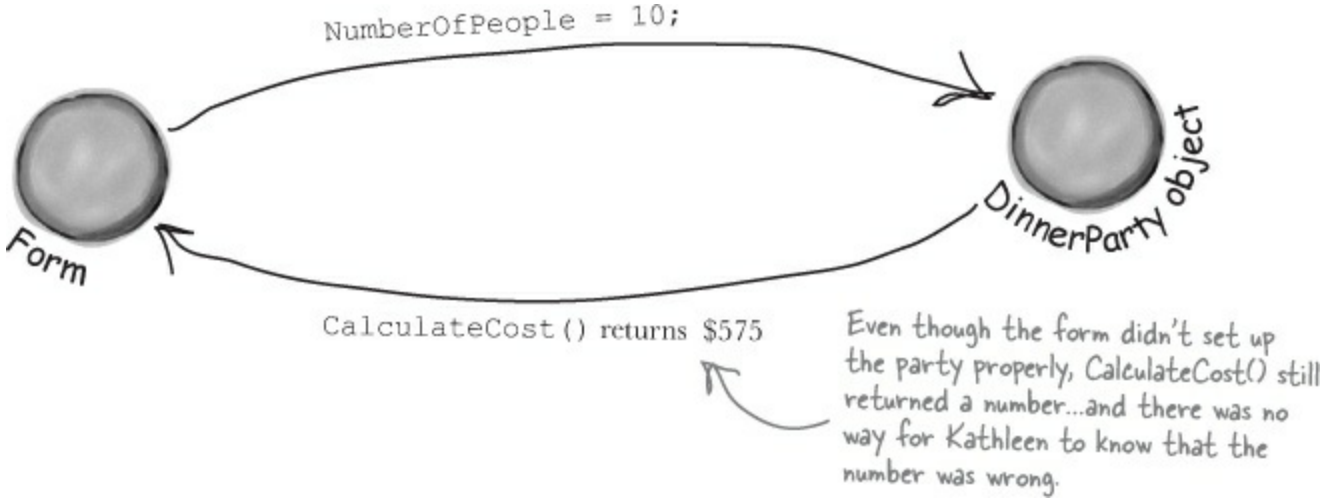
It's easy to accidentally misuse your objects

Kathleen ran into problems because her form ignored the convenient `CalculateCostOfDecorations()` method that you set up and instead went directly to the fields in the `DinnerParty` class. So even though your `DinnerParty` class worked just fine, the form called it in an unexpected way... and that caused problems.

- **❶ HOW THE DINNERPARTY CLASS EXPECTED TO BE CALLED**
  The `DinnerParty` class gave the form a perfectly good method to calculate the total cost of decorations. All it had to do was set the number of people and then call `CalculateCostOfDecorations()`, and then `CalculateCost()` will return the correct cost.



- **❷ HOW THE DINNERPARTY CLASS WAS ACTUALLY CALLED**
  The form set the number of people, but just called the `CalculateCost()` method without first recalculating the cost of the decorations. That threw off the whole calculation, and Kathleen ended up giving Rob the wrong price.



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 object *of the same class*).

NOTE

Use your laziness to your own benefit — if you leave off the "private" or "public" declaration, then C# will just assume that your field is private.

NOTE

Also, a class's static methods can access the private field in any instance of that class.

```
class DinnerParty {
   private int numberOfPeople;

   ...

   public void SetPartyOptions(int people, bool fancy) {
      numberOfPeople = people;
      CalculateCostOfDecorations(fancy);
   }

   public int GetNumberOfPeople() {
      return numberOfPeople;
   }
}
```

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 DinnerParty, its numberOfPeople field can only be read and written by that instance—or another instance of DinnerParty. Other objects won't even know it's there.

Other objects still need a way to set the number of people for the dinner party. One good way to give them access to it is to add methods to set or get the number of people. That way you can make sure that the CalculateCostOfDecorations() method gets run every time the number of people is changed. That'll take care of that pesky bug.

By making the field that holds the number of party guests *private*, we only give the form one way to tell the `DinnerParty` class how many people are at the party — and we can make sure the cost of decorations is recalculated properly. When you make some data private and then write code to use that data, it's called *encapsulation*.

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

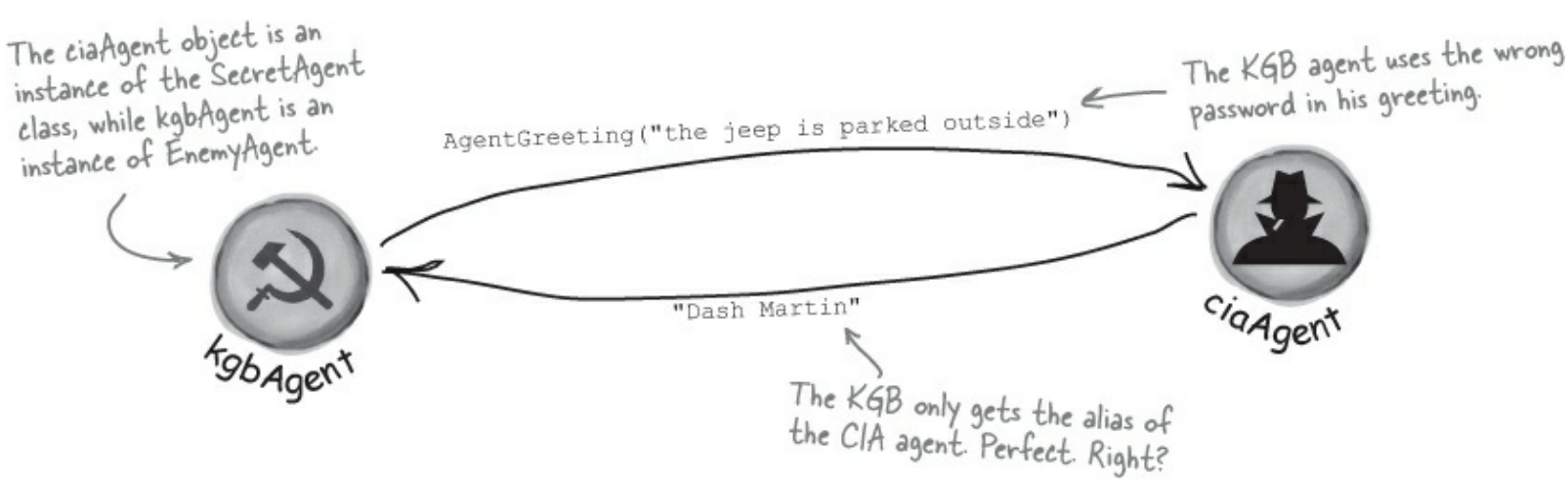
When you make all of your fields and methods public, any other class can access them. Everything your class does and knows about becomes an open book for every other class in your program...and you just saw how that can cause your program to behave in ways you never expected. Encapsulation lets you control what you share and what you keep private inside your class. Let's see how this works:

- ❶ Super-spy Herb Jones is 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.

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

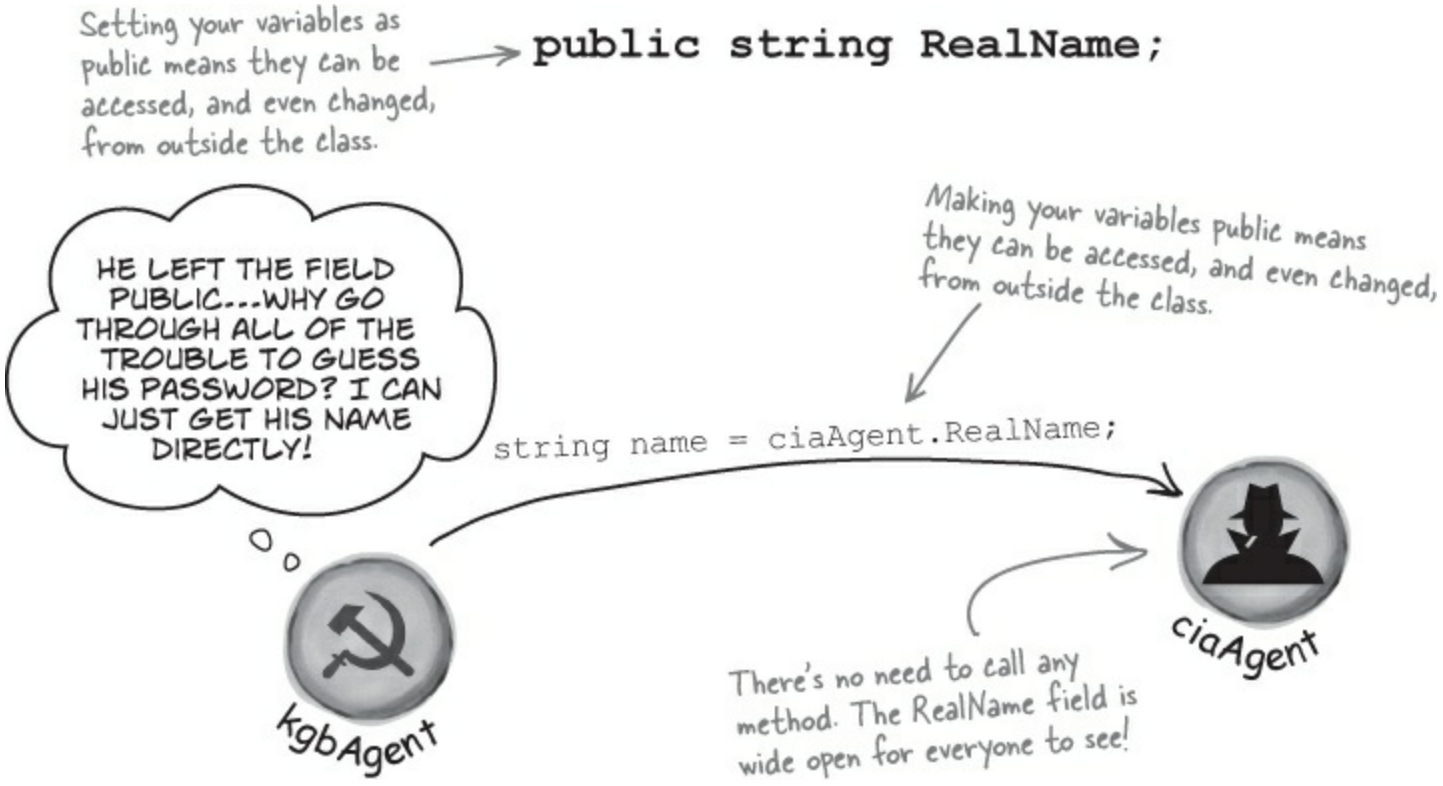
- ❷ Agent Jones has a plan to help him evade the enemy KGB agents. He added an `AgentGreeting()` method that takes a password as its parameter. If he doesn't get the right password, he'll only reveal his alias, Dash Martin.

- ❸ 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.


The ciaAgent object is an instance of the SecretAgent class, while kgbAgent is an instance of EnemyAgent.
The KGB agent uses the wrong password in his greeting.
AgentGreeting("the jeep is parked outside")
"Dash Martin"
kgbAgent
ciaAgent
The KGB only gets the alias of the CIA agent. Perfect. Right?

But is the RealName field REALLY protected?

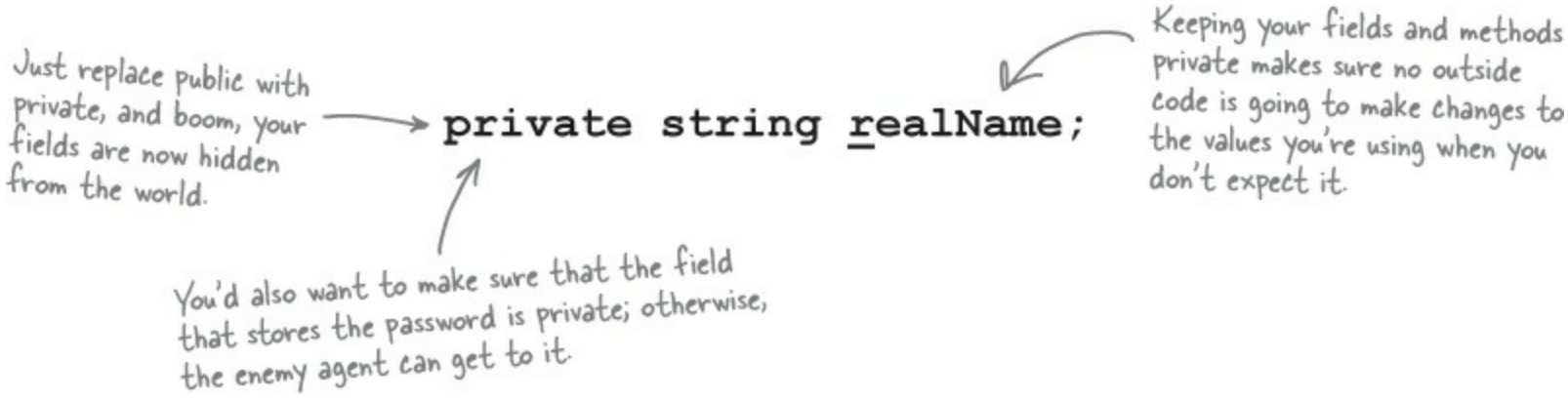
So as long as the KGB doesn't know any CIA agent passwords, the CIA's real names are safe. Right? But what about the field declaration for the `realName` field:



Agent Jones 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

The kgbAgent object can't access the ciaAgent's private fields because they're instances of **different classes**.

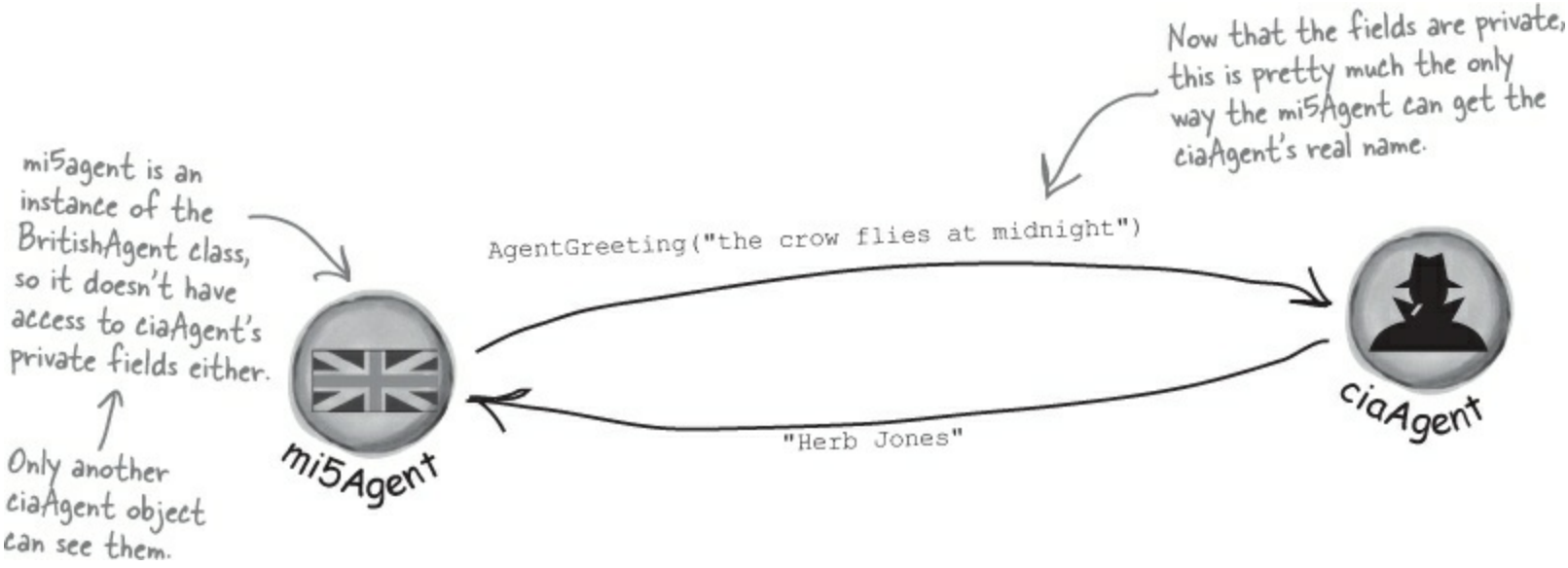


BRAIN POWER

Why do you think we used an uppercase ***R*** for the public field, but switched to a lowercase ***r*** for the private one?

Private fields and methods can only be accessed from inside the 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 can see everything — any class can **see private fields in other instances of the same class**.



THERE ARE NO DUMB QUESTIONS

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

A: **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. Here's an example. When computers generate random numbers, they use special values called *seeds*. You don't need to know how they work, but every instance of `Random` actually contains an array of several dozen numbers that it uses to make sure that `Next()` always gives you a random number. If you create an instance of `Random`, you won't be able to see that array. That's because you don't need it — but 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: **Q: Hey, I just noticed that all of the event handlers I've been using have the `private` keyword. Why are they private?**

A: **A:** Because C# forms are set up so that only the controls on the forms can trigger event handlers. When you put the `private` keyword in front of any method, then that method can only be used from inside your class. When the IDE adds an event handler method to your program, it declares it as private so other forms or objects can't get to it. But there's no rule that says that an event handler must be private. In fact, you can check this out for yourself — double-click on a button, then change its event handler declaration to `public`. The code will still compile and run.

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.

SHARPEN YOUR PENCIL

Here's a class with some private fields. Circle the statements below that **won't compile** if they're run from outside the class using **an instance of the object called `mySuperChef`**.

```
class SuperChef
{
      public string cookieRecipe;
      private string secretIngredient;
      private const int loyalCustomerOrderAmount = 60;
      public int Temperature;
      private string ingredientSupplier;

      public string GetRecipe (int orderAmount)
      {
            if (orderAmount >= loyalCustomerOrderAmount)
            {
                  return cookieRecipe + " " + secretIngredient;
            }
            else
            {
                  return cookieRecipe;
            }
      }
}
```

1. `string ovenTemp = mySuperChef.Temperature;`
2. `string supplier = mySuperChef.ingredientSupplier;`
3. `int loyalCustomerOrderAmount = 54;`
4. `mySuperChef.secretIngredient = "cardamom";`
5.
```
mySuperChef.cookieRecipe = "get 3 eggs, 2 1/2 cup flour, 1 tsp salt,
1 tsp vanilla and 1.5 cups sugar and mix them together. Bake for 10
minutes at 375. Yum!";
```
6. `string recipe = mySuperChef.GetRecipe(56);`
7. After running all of the lines that will compile above, what's the value of `recipe`?

_____

_____

SHARPEN YOUR PENCIL SOLUTION

Here's a class with some private fields. Circle the statements below that **won't compile** if they're run from outside the class using **an instance of the object called `mySuperChef`**.

```
class SuperChef
{
      public string cookieRecipe;
      private string secretIngredient;
      private const int loyalCustomerOrderAmount = 60;
      public int Temperature;
      private string ingredientSupplier;

      public string GetRecipe (int orderAmount)
      {
            if (orderAmount >= loyalCustomerOrderAmount)
            {
                  return cookieRecipe + " " + secretIngredient;
            }
            else
            {
                  return cookieRecipe;
            }
      }
}
```

The only way to get the secret ingredient is to order a whole lot of cookies. Outside code can't access this field directly.

1. `string ovenTemp = mySuperChef.Temperature;`

#1 doesn't compile because you can't just assign an int to a string.

2. `string supplier = mySuperChef.ingredientSupplier;`

3. `int loyalCustomerOrderAmount = 54;`

#2 and #4 don't compile because ingredientSupplier and secretIngredient are private.

4. `mySuperChef.secretIngredient = "cardamom";`

5. `mySuperChef.cookieRecipe = "Get 3 eggs, 2 1/2 cup flour, 1 tsp salt, 1 tsp vanilla and 1.5 cups sugar and mix them together. Bake for 10 minutes at 375. Yum!";`

Even though you created a local variable called loyalCustomerAmount and set it to 54, that didn't change the object's loyalCustomerAmount value, which is still 60—so it won't print the secret ingredient.

6. `string recipe = mySuperChef.GetRecipe(56);`

7. After running all of the lines that will compile above, what's the value of recipe?

"Get 3 eggs, 2 1/2 cup flour, 1 tsp salt, 1 tsp vanilla and 1.5 cups sugar and mix them together.
Bake for 10 minutes at 375. Yum!"

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.

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 pop up in the IntelliSense window later when you need to use that class.
- **Easy to maintain**
  Remember that bug in Kathleen's program? It happened because the form accessed a field directly rather than using a method to set it. If that field had been private, you would 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.

Encapsulation means having one class hide information from another. It helps you prevent bugs in your programs.

BRAIN POWER

How could building a poorly encapsulated class now make your programs harder to modify later?

Mike's navigator program could use better encapsulation

NOTE

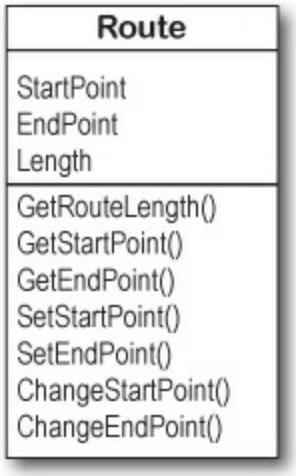
Geocaching is a sport where people use their GPS navigators to hide and seek containers that can be hidden anywhere in the world. Mike is really into GPS stuff, so you can see why he likes it so much.

Remember Mike's street navigation program from Chapter 3? Mike joined a geocaching group, and he thinks his navigator will give him an edge. But it's been a while since he's worked on it, and now he's run into a little trouble. Mike's navigator program has a `Route` class that stores a single route between two points. But he's running into all sorts of bugs because he can't seem to figure out how it's supposed to be used! Here's what happened when Mike tried to go back to his navigator and modify the code:

- Mike set the `StartPoint` property to the GPS coordinates of his home and the `EndPoint` property to the coordinates of his office, and checked the `Length` property. It said the length was 15.3. When he called the `GetRouteLength()` method, it returned 0.
- He uses the `SetStartPoint()` property to set the start point to the coordinates of his home and the `SetEndPoint()` property to set the end point to his office. The `GetRouteLength()` method returned 9.51, and the `Length` property contained 5.91.
- When he tried using the `StartPoint` property to set the starting point and the `SetEndPoint()` method to set the ending point, `GetRouteLength()` always returned 0 and the `Length` property always contained 0.
- When he tried using the `SetStartPoint()` method to set the starting point and the `EndPoint` property to set the ending point, the `Length` property contained 0, and the `GetRouteLength()` method caused the program to crash with an error that said something about not being able to divide by zero.

SHARPEN YOUR PENCIL

Here's the `Route` object from Mike's navigator program. Which properties or methods would ***you*** make `private` in order to make it easier to use?



_____

_____

_____

_____

_____

_____

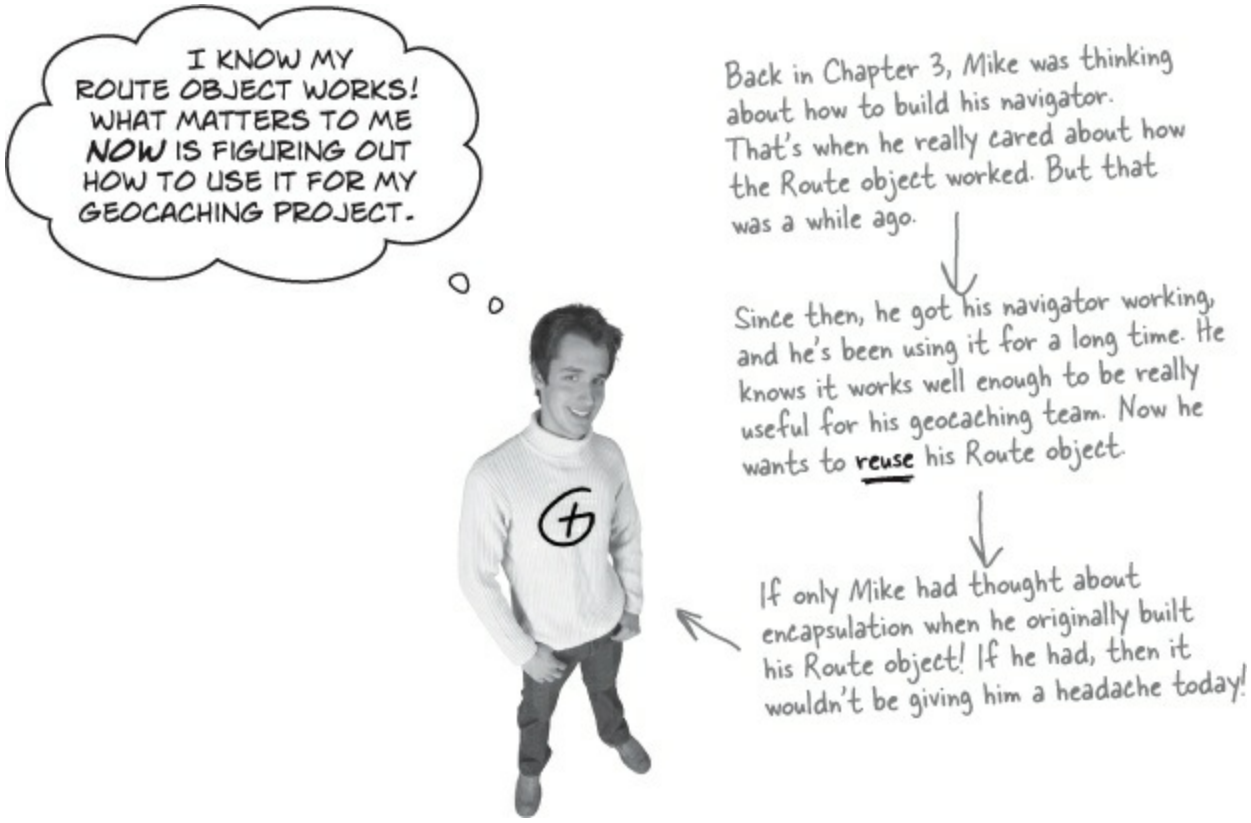
NOTE

There are lots of ways to solve this problem, all potentially correct! Write down the one you think is best.

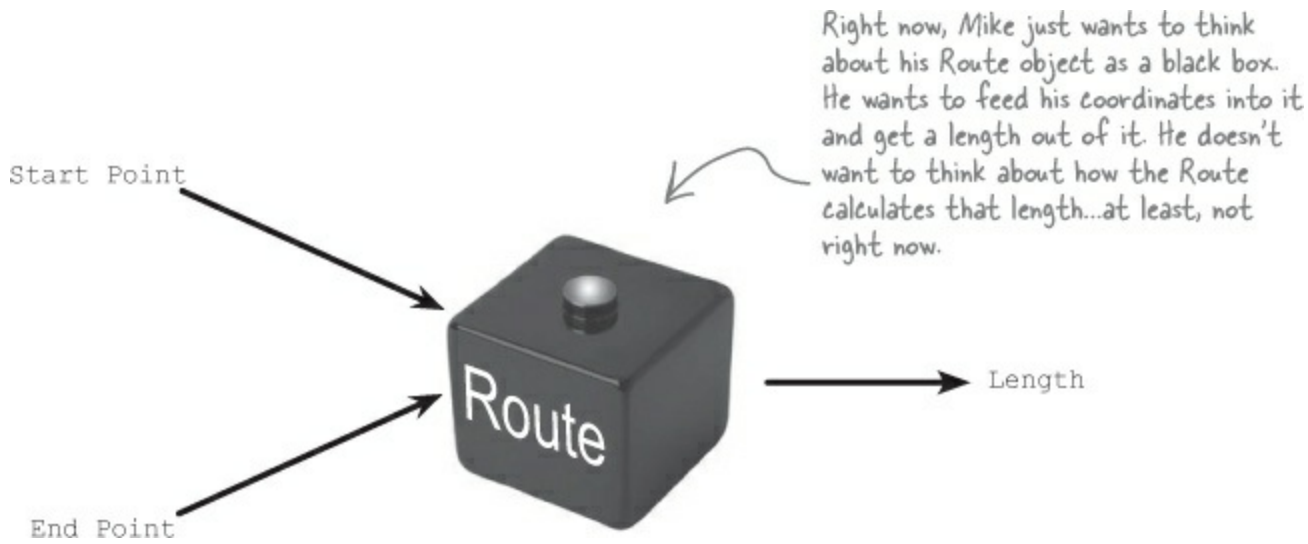
Think of an object as a black box

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 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 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!



If you encapsulate your classes well today, that makes them a lot easier to reuse tomorrow.



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 is difficult for some programmers to understand.

Until now, everything you've learned 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!

A few ideas for encapsulating classes

- **Think about ways the fields can be misused.**
  What can go wrong if they're not set properly?
- **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.
- **What fields require some processing or calculation to happen when they're set?**
  Those are prime candidates for encapsulation. If someone writes a method later that changes the value in any one of them, it could cause problems for the work your program is trying to do.

- **Only make fields and methods public if you need to.**
  If you don't have a reason to declare something public, don't. You could make things really messy for yourself by making all of the fields in your program public — but don't just go making everything private, either. Spending a little time up front thinking about which fields really need to be public and which don't can save you a lot of time later.

Encapsulation keeps your data pristine

Sometimes the value in a field changes as your program does what it's supposed to do. If you don't explicitly tell your program to reset the value, you can do your calculations using the old one. When this is the case, you want to have your program execute some statements any time a field is changed — like having Kathleen's program recalculate the cost every time you change the number of people. We can avoid the problem by encapsulating the data using private fields. We'll provide a method to get the value of the field, and another method to set the field and do all the necessary calculations.

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. That makes the uppercase letters look like "humps" of a camel.

NOTE

Your code is easier to read when you use consistent case when choosing names for fields, properties, variables, and methods This is a convention that a lot of programmers follow.

A quick example of encapsulation

A `Farmer` class uses a field to store the number of cows, and multiplies it by a number to figure out how many bags of cattle feed are needed to feed the cows:

```
class Farmer
{
        private int numberOfCows;
}
```

We'd better make this field private so nobody can change it without also changing bagsOfFeed—if they get out of sync, that'll create bugs!

When you create a form to let a user enter the number of cows into a numeric field, you need to be able to change the value in the `numberOfCows` field. To do that, you can create a method that returns the value of the field to the form object:

```
public const int FeedMultiplier = 30;
public int GetNumberOfCows()
{
        return numberOfCows;
}
public void SetNumberOfCows(int newNumberOfCows)
{
        numberOfCows = newNumberOfCows;
        BagsOfFeed = numberOfCows * FeedMultiplier;
}
```

The farmer needs 30 bags of feed for each cow.

We'll add a method to give other classes a way to get the number of cows.

numberOfCows is a private field, so we used camelCase when we named it.

And here's a method to set the number of cows that makes sure the BagsOfFeed field is changed too. Now there's no way for the two to get out of sync.

These accomplish the same thing!

```
private int numberOfCows;
public int NumberOfCows
{
    get
    {
        return numberOfCows;
    }
    set
    {
        numberOfCows = value;
        BagsOfFeed = numberOfCows * FeedMultiplier;
    }
}
```

We'll rename the private field to numberOfCows (notice the lowercase "n"). This will become the **backing field** for the NumberOfCows property.

You'll often use properties by combining them with a normal field declaration. Here's the declaration for NumberOfCows.

This is a **get accessor**. It's a method that's run any time the NumberOfCows property is **read**. It has a return value that matches the type of the variable—in this case it returns the value of the private numberOfCows property.

This is a **set accessor** that's called every time the NumberOfCows property is **set**. Even though the method doesn't look like it has any parameters, it actually has one called **value** that contains whatever value the field was set to.

Properties make encapsulation easier

You can use **properties**, which are methods that look just like fields to other objects. A property can be used to get or set a **backing field**, which is just a name for a field set by a property.

You **use** get and set accessors exactly like fields. Here's code for a button that sets the number of cows and then gets the bags of feed:

```
private void button1_Click(object sender, EventArgs e) {
      Farmer myFarmer = new Farmer();
      myFarmer.NumberOfCows = 10;

      int howManyBags = myFarmer.BagsOfFeed;

      myFarmer.NumberOfCows = 20;
      howManyBags = myFarmer.BagsOfFeed;
}
```

When this line sets NumberOfCows to 10, the set accessor sets the private numberOfCows field and then updates the public BagsOfFeed field.

Since the NumberOfCows set accessor updated BagsOfFeed, now you can get its value.

Even though the code treats NumberOfCows like a field, it runs the set accessor, passing it 20. And when it queries the BagsOfFeed field it runs the get accessor, which returns 20*30=600.

Build an application to test the Farmer class

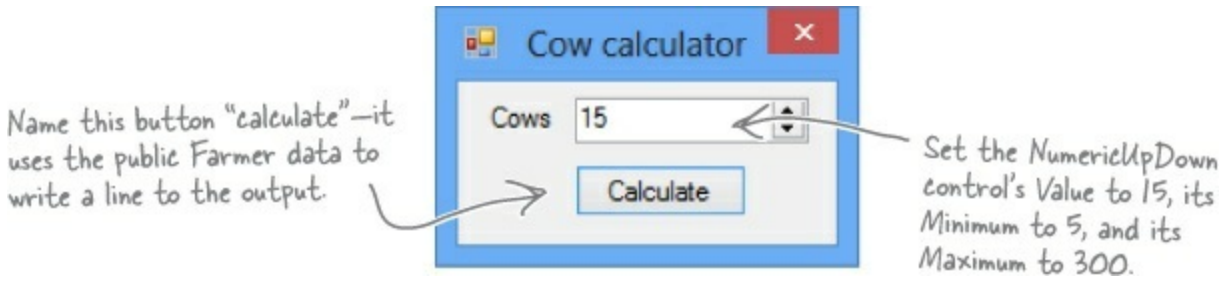
Create a **new Windows Forms application** that we can use to test the `Farmer` class and see properties in action. The **`Console.WriteLine()`** method will write the results to the *Output Window* in the IDE.

DO THIS

- ❶ Add the `Farmer` class to your project:

```
class Farmer {
    public int BagsOfFeed;
    public const int FeedMultiplier = 30;

    private int numberOfCows;
    public int NumberOfCows {
      // (add the get and set accessors from the
      // previous page)
    }
}
```

- ❷ Build this form:



- ❸ Here's the code for the form. It uses `Console.WriteLine()` to send its output to the **Output window** (which you can bring up by selecting "Output" from the Debug→Windows menu). You can pass several parameters to `WriteLine()` — the first one is the string to write. If you include "`{0}`" inside the string, then `WriteLine()` replaces it with the first parameter. It replaces "`{1}`" with the second parameter, "`{2}`" with the third, etc.

```
public partial class Form1 : Form {
    Farmer farmer;
    public Form1() {
        InitializeComponent();
        farmer = new Farmer() { NumberOfCows = 15 };
    }
    private void numericUpDown1_ValueChanged(object sender, EventArgs e) {
        farmer.NumberOfCows = (int)numericUpDown1.Value;
    }
    private void calculate_Click(object sender, EventArgs e) {
        Console.WriteLine("I need {0} bags of feed for {1} cows",
            farmer.BagsOfFeed, farmer.NumberOfCows);
    }
}
```

Use the Console.WriteLine() method to send a line of text to the IDE's Output window.

WriteLine() replaces "{0}" with the value in the first parameter, and "{1}" with the second parameter.

WATCH IT!

Console output is displayed in the Output window.

When a Windows Forms application uses the `Console.WriteLine()` *method to write output, the ouptut is displayed in the Output window in the IDE. WinForms apps don't typically use console output, but we will use it extensively as a learning tool.*

Don't forget that controls need to be "hooked up" to their event handlers! Double-click on Button and NumericUpDown in the designer to make the IDE create their event handler method stubs.

Use automatic properties to finish the class

It looks like the Cow Calculator works really well. Give it a shot — run it and click the button. Then change the number of cows to 30 and click it again. Do the same for 5 cows and then 20 cows. Here's what your Output window should look like:

But there's a problem with the class. Add a button to the form that executes this statement:

```
farmer.BagsOfFeed = 5;
```

Now run your program again. It works fine until you press the new button. But press that button and then press the Calculate button again. Now your ouput tells you that you need 5 bags of feed — *no matter how many cows you have*! As soon as you change the `NumericUpDown`, the Calculate button should work again.

NOTE

Can you see how this could lead you to accidentally add a really irritating bug in your program?

Fully encapsulate the Farmer class

The problem is that your class **isn't fully encapsulated**. You used properties to encapsulate `NumberOfCows`, but `BagsOfFeed` is still public. This is a common problem. In fact, it's so common that C# has a way of automatically fixing it. Just change the public `BagsOfFeed` field to an ***automatic property***. And the IDE makes it really easy for you to add automatic properties. Here's how:

- ❶ Remove the `BagsOfFeed` field from the `Farmer` class. Put your cursor where the field used to be, and then type **`prop`** and press the Tab key twice. The IDE will add this line to your code:

> **NOTE**
>
> The prop-tab-tab code snippet adds an automatic property to your code.

```
public int MyProperty { get; set; }
```

- ❷ Press the Tab key — the cursor jumps to `MyProperty`. Change its name to `BagsOfFeed`:

```
public int BagsOfFeed { get; set; }
```

  Now you've got a property instead of a field. When C# sees this, it works exactly the same as if you had used a backing field (like the private `numberOfCows` behind the public `NumberOfCows` property).

- ❸ That hasn't fixed our problem yet. But there's an easy fix — just make it a **read-only property**:

```
public int BagsOfFeed { get; private set; }
```

  Try to rebuild your code — you'll get an error on the line in the button that sets `BagsOfFeed` telling you that the **set accessor is inaccessible**. You can't modify `BagsOfFeed` from outside the `Farmer` class — you'll need to remove that line in order to get your code to compile, so **remove the button *and* its event handler** from the form. Now your `Farmer` class is better encapsulated!

A property is read-only if it can't be set by another class. You can make its set accessor `private`. You can also leave out the set accessor for normal properties, but not automatic properties (which must have both get and set accessors).

What if we want to change the feed multiplier?

We built the Cow Calculator to use a `const` for the feed multiplier. But what if we want to use the same `Farmer` class in different programs that need different feed multipliers? You've seen how poor encapsulation can cause problems when you make fields in one class too accessible to other classes. That's why you should **only make fields and methods public if you need to**. Since the Cow Calculator never updates `FeedMultiplier`, there's no need to allow any other class to set it. So let's change it to a read-only property that uses a backing field.

DO THIS!

- ❶ Remove this line from your `Farmer` class:

```
public const int FeedMultiplier = 30;
```

Use prop-tab-tab to add a read-only property. But instead of adding an automatic property, use a backing field:

This property acts just like an int field, except instead of storing a value it just returns the backing field, feedMultiplier. And since there's no set accessor, it's read-only. It has a public get, which means any other class can read the value of FeedMultiplier. But since its set is private, that makes it read-only — it can only be set by an instance of Farmer.

```
private int feedMultiplier;
public int FeedMultiplier { get { return feedMultiplier; } }
```

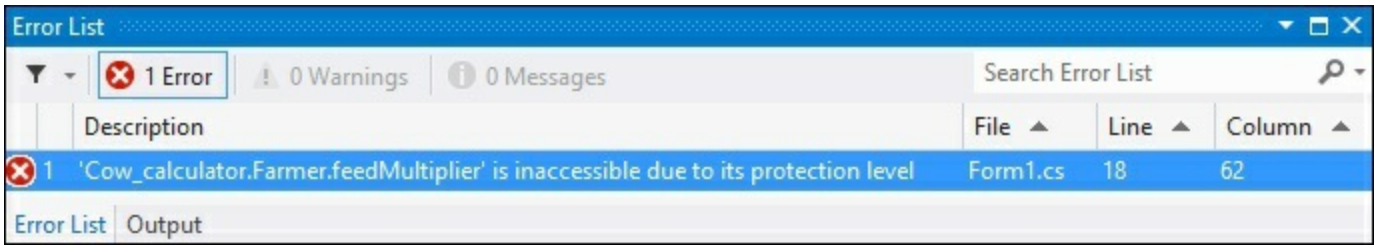
Since we changed FeedMultiplier from a public const to a private int field, we changed its name, so it starts with a lowercase "f." That's a pretty standard naming convention you'll see throughout the book.

- ❷ Go ahead and make that change to your code. Then run it. Uh oh — something's wrong! `BagsOfFeed` **always returns 0 bags**.
Wait, that makes sense. `FeedMultiplier` never got initialized. It starts out with the default value of zero and never changes. When it's multiplied by the number of cows, it still gives you zero. So add an object initializer:

```
public Form1() {
   InitializeComponent();
   farmer = new Farmer() { NumberOfCows = 15, feedMultiplier = 30 };
```

Check the Error List window for helpful warnings from the IDE about things like forgetting to initialize a variable before using it.

Uh oh — the **program won't compile!** You should get this error:



You can only initialize <u>public</u> fields and properties inside an object initializer. So how can you make sure your object gets initialized properly if some of the fields that need to be initialized are <u>private</u>?

Use a constructor to initialize private fields

If you need to initialize your object, but some of the fields that need to be initialized are private, then an object initializer just won't do. Luckily, there's a special method that you can add to any class called a **constructor**. If a class has a constructor, then that constructor is the **very first thing that gets executed** when the class is created with the `new` statement. You can pass parameters to the constructor to give it values that need to be initialized. But the constructor **does not have a return value**, because you don't actually call it directly. You pass its parameters to the `new` statement. And you already know that `new` returns the object — so there's no way for a constructor to return anything.

All you have to do to add a constructor to a class is add a method that has the same name as the class and no return value.

- **❶ ADD A CONSTRUCTOR TO YOUR FARMER CLASS.**
  This constructor only has two lines, but there's a lot going on here. So let's take it step by step. We already know that we need the number of cows and a feed multiplier for the class, so we'll add them as parameters to the constructor. Since we changed `feedMultiplier` from a `const` to an `int`, now we need an initial value for it. So let's make sure it gets passed into the constructor. We'll use the constructor to set the number of cows, too.

Notice how there's no "void" or "int" or another type after "public". That's because constructors don't have a return value.

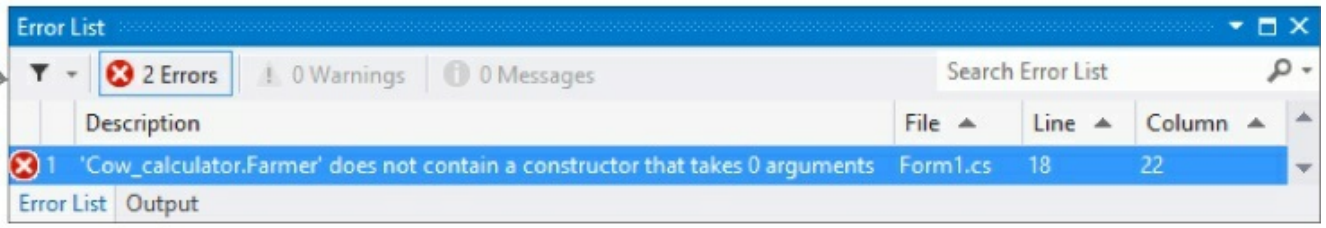
The "this" keyword in this.feedMultiplier tells C# that you're talking about the field, not the parameter with the same name.

```
public Farmer(int numberOfCows, int feedMultiplier) {
    this.feedMultiplier = feedMultiplier;
    NumberOfCows = numberOfCows;
}
```

The first thing we'll do is set the feed multiplier, because it needs to be set before we can call the NumberOfCows set accessor.

If we just set the private numberOfCows field, the NumberOfCows set accessor would never be called. Setting NumberOfCows makes sure it's called.

This is the error you'll get if your constructor takes parameters but your "new" statement doesn't have any.



- **❷ NOW CHANGE THE FORM SO THAT IT USES THE CONSTRUCTOR.**
  The only thing you need to do now is change the form so that the `new` statement that creates the `Farmer` object uses the constructor instead of an object initializer. Once you replace the `new` statement, both errors will go away, and your code will work!

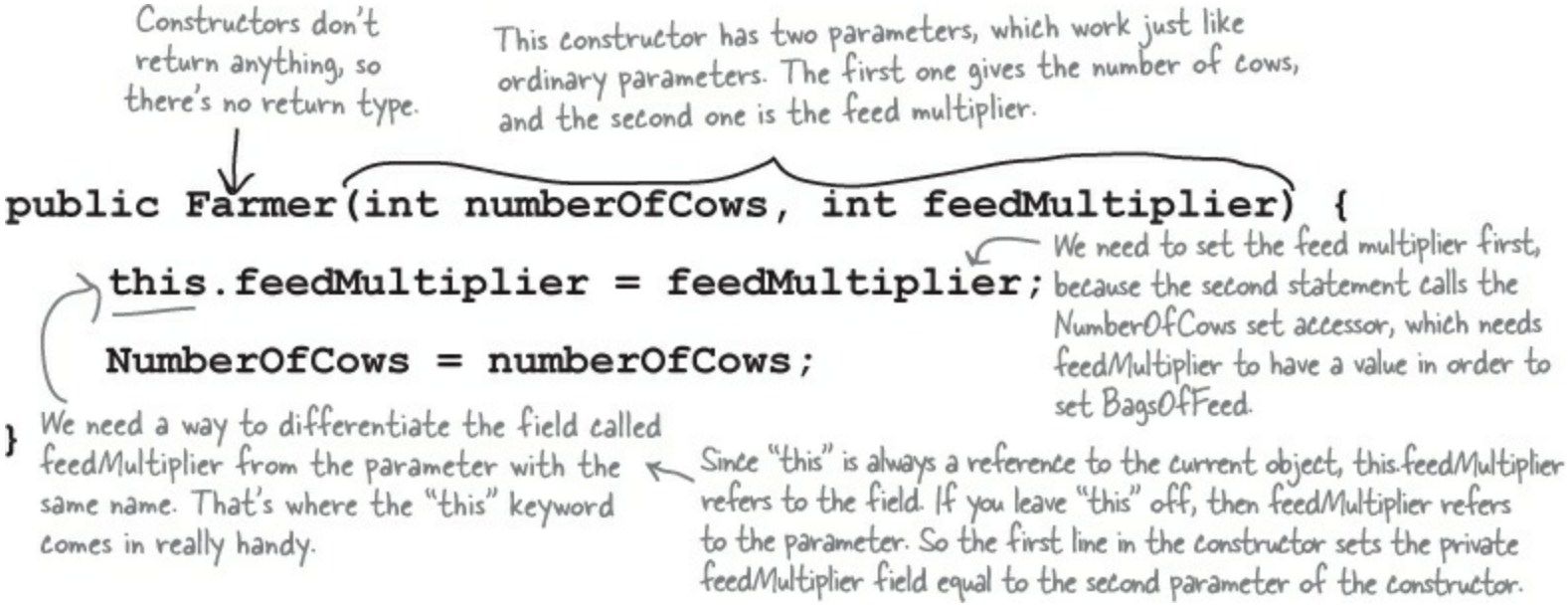
You already know that the form is an object. Well, it's got a constructor too! That's what this method is—notice how it's named Form1 (like the class) and it doesn't have a return value.

```
public Form1() {
    InitializeComponent();
    farmer = new Farmer(15, 30);
}
```

Here's where the new statement calls the constructor. It looks **just like any other new statement**, except that it has parameters that it passes into the constructor method. When you type it in, watch for the IntelliSense pop up—it looks just like any other method.

CONSTRUCTORS WAY UP CLOSE

Let's take a closer look at the `Farmer` constructor so we can get a good sense of what's really going on.



THERE ARE NO DUMB QUESTIONS

Q: **Q: Is it possible to have a constructor without any parameters?**

A: **A:** Yes. It's actually very common for a class to have a constructor without a parameter. In fact, you've already seen an example of it — **your form's constructor**. Look inside a newly added Windows form and find its constructor's declaration:

```
public Form1() {
  InitializeComponent();
}
```

That's the constructor for your form object. It doesn't take any parameters, but it does have to do a lot. Take a minute and open up *Form1.Designer.cs*. Find the `InitializeComponent()` method by clicking on the plus sign next to "Windows Form Designer generated code."
That method initializes all of the controls on the form and sets all of their properties. If you drag a new control onto your form in the IDE's form designer and set some of its properties in the Properties window, you'll see those changes reflected inside the `InitializeComponent()` method.
The `InitializeComponent()` method is called inside the form's constructor so that the controls all get initialized as soon as the form object is created. (Remember, every form that gets displayed is just another object that happens to use methods that the .NET Framework provides in the `System.Windows.Forms` namespace to display windows, buttons, and other controls.)

WATCH IT!

When a method's parameter has the same name as a field, then it masks the field.

The constructor's `feedMultiplier` parameter masks the backing field behind the `FeedMultiplier` property because they have the same name, so the parameter takes precedence inside the body of the constructor. If you wanted to use the backing field inside the constructor, you'd use the `this` keyword: `feedMultiplier` refers to the parameter, and `this.feedMultiplier` refers to the private field.

NOTE

Here's a helpful way to remember what "this" does: think of it as short for "this instance."

THERE ARE NO DUMB QUESTIONS

Q: **Q: Why would I need complicated logic in a get or set accessor? Isn't it just a way of creating a field?**

A: **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 Kathleen's problem — she ran into trouble because the form didn't run the method to recalculate the cost of the decorations after setting the number of people in the `DinnerParty` class. If we replaced the field with a set accessor, then we could make sure that the set accessor recalculates the cost of the decorations. (In fact, you're about to do exactly that in just a couple of pages!)

Q: **Q: Wait a minute — so what's the difference between a method and a get or set accessor?**

A: **A:** There is none! Get and set accessors are a special kind of method — one that looks just like a field to other objects, and is called whenever that "field" is set. Get accessors always return a value that's the same type as the field, and set accessors always take exactly one parameter called `value` whose type is the same as the field. Oh, and by the way, you can just say "property" instead of "get and set accessor."

Q: **Q: So you can have ANY kind of statement in a property?**

A: **A:** Absolutely. Anything you can do in a method, you can do in a property. They can call other methods, access other fields, even create objects and instances. But they only get called when a property gets accessed, so it doesn't make sense to have any statements in them that don't have to do with getting or setting the property.

Q: **Q: If a set accessor always takes a parameter called `value`, why doesn't its declaration have parentheses with the "`int value`" parameter in them, like you'd have with any other method that takes a parameter called `value`?**

A: **A:** Because C# was built to keep you from having to type in extra information that the compiler doesn't need. The parameter gets declared without you having to explicitly type it in, which doesn't sound like much when you're only typing one or two — but when you have to type a few hundred, it can be a real time saver (not to mention a bug preventer).
Every set accessor ***always*** has exactly one parameter called `value`, and the type of that parameter ***always*** matches the type of the property. C# has all the information it needs about the type and parameter as soon as you type `set {`. So there's no need for you to type any more, and the C# compiler isn't going to make you type more than you have to.

Q: **Q: Wait a sec — is that why I don't add a return value to my constructor?**

A: **A:** Exactly! Your constructor doesn't have a return value because ***every*** constructor is always `void`. It would be redundant to make you type `void` at the beginning of each constructor, so you don't have to.

Q: **Q: Can I have a get without a set or a set without a get?**

A: **A:** Yes! When you have a get accessor but no set, you create a read-only property. For example, the `SecretAgent` class might have public read-only field with a backing field for the name:

```
string name = "Dash Martin";
public string RealName {
   get { return name; }
}
```

And if you create a property with a set accessor but no get, then your backing field can only be written, not read. The `SecretAgent` class could use that for a `Password` property that other spies could write to, but not see:

```
public string Password {
  set {
   if (value == secretCode) {
     name = "Herb Jones";
   }
}
```

Both of those techniques can come in really handy when you're doing encapsulation.

Q: **Q: I've been using objects for a while, but I haven't written a constructor. Does that mean some classes don't need one?**

A: **A:** No, it just means that C# automatically makes a zero-parameter constructor if there's none defined. If you define a constructor, then it doesn't do that. That's a valuable tool for encapsulation, because it means that you have the option — but not the requirement — to force anyone instantiating your class to use your constructor.

Properties (get and set accessors) are just another kind of C# method that's only run when the property value is read or written.

Here's something useful: the first line of a method that contains the access modifier, return value, name, and parameters i called the method's signature. Properties have signatures, too.

SHARPEN YOUR PENCIL

Take a look at the `get` and `set` accessors here. The form that is using this class has a new instance of `CableBill` called `thisMonth` and calls the `GetThisMonthsBill()` method with a button click. Write down the value of the `amountOwed` variable after the code below executed.

```
class CableBill {
    private int rentalFee;
    public CableBill(int rentalFee) {
        this.rentalFee = rentalFee;
        discount = false;
    }

    private int payPerViewDiscount;
    private bool discount;
    public bool Discount {
        set {
            discount = value;
            if (discount)
            payPerViewDiscount = 2;
            else
            payPerViewDiscount = 0;
        }
    }

    public int CalculateAmount(int payPerViewMoviesOrdered) {
        return (rentalFee - payPerViewDiscount) * payPerViewMoviesOrdered;
    }
}
```

1.
```
CableBill january = new CableBill(4);
MessageBox.Show(january.CalculateAmount(7).ToString());
```

What's the value of `amountOwed`?

2.
```
CableBill february = new CableBill(7);
february.payPerViewDiscount = 1;
MessageBox.Show(february.CalculateAmount(3).ToString());
```

What's the value of `amountOwed`?

3.
```
CableBill march = new CableBill(9);
march.Discount = true;
MessageBox.Show(march.CalculateAmount(6).ToString());
```

What's the value of `amountOwed`?

THERE ARE NO DUMB QUESTIONS

Q: **Q: I noticed that you used uppercase names for some fields but lowercase ones for others. Does that matter?**

A: **A:** Yes — it matters to you. But it doesn't matter to the compiler. C# doesn't care what you name your variables, but if you choose weird names then it makes your code hard to read. Sometimes it can get confusing when you have variables that are named the same, except one starts with an uppercase letter and the other starts with a lowercase one.

Case matters in C#. You can have two different variables called `Party` and `party` in the same method. It'll be confusing to read, but your code will compile just fine. Here are a few tips about variable names to help you keep it straight. They're not hard-and-fast rules — the compiler doesn't care whether a variable is uppercase or lowercase — but they're good suggestions to help make your code easier to read.

1. When you declare a private field, it should be in camelCase and start with a lowercase letter. (It's called camelCase because it starts with a lowercase letter and additional words are uppercase, so they resemble humps on a camel.)
2. Properties and methods are in PascalCase (they start with an uppercase letter), whether or not they're public.
3. Parameters to methods should be in camelCase.
4. Some methods, especially constructors, will have parameters with the same names as fields. When this happens, the parameter **masks** the field, which means statements in the method that use the name end up referring to the parameter, not the field. Use the `this` keyword to fix the problem — add it to the variable to tell the compiler you're talking about the field, not the parameter.

SHARPEN YOUR PENCIL

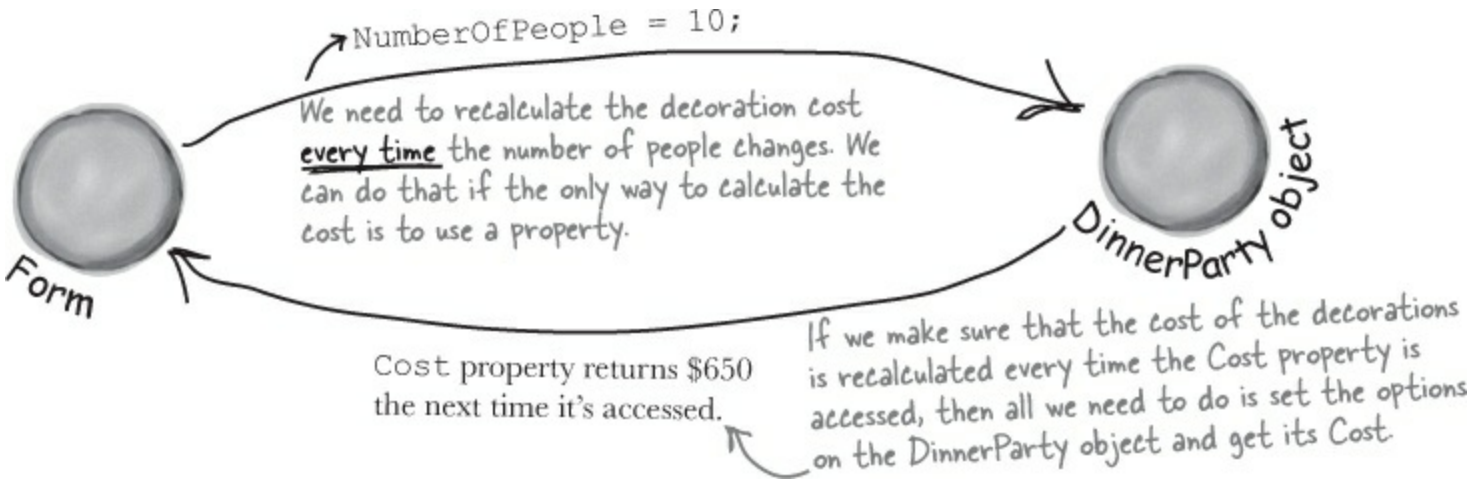
This code has problems. Write down what you think is wrong with the code, and what you'd change.

```
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)
      {
            if (this.coinsInserted >= price) { // check the field
                gumballs -= 1;
                return "Here's your gumball";
            } else {
                return "Please insert more coins";
            }
      }
}
```

EXERCISE

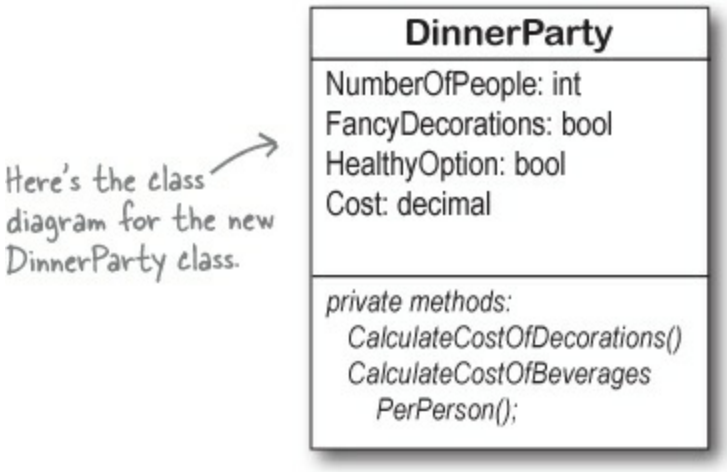
Use what you've learned about properties and constructors to fix Kathleen's Party Planner program. This new program will be much simpler and more consistent than the first version.

- **❶ Fix the Dinner Party calculator.**
To fix the `DinnerParty` class, we'll need to make sure the `CalculateCostOfDecorations()` method is called every time `NumberOfPeople` changes. We'll do it by adding a property called `Cost`.



- **❷ Use properties to set the number of people and the party options.**
You may want to create a new project, because you're going to overhaul the `DinnerParty` class. Start by creating these three automatic properties:

```
public int NumberOfPeople { get; set; }
public bool FancyDecorations { get; set; }
public bool HealthyOption { get; set; }
```



You'll also need a **constructor** with this signature to set the properties:

```
public DinnerParty(int numberOfPeople, bool healthyOption,
                   bool fancyDecorations)
```

- **❸ Create private methods to calculate the intermediate costs.**
Here are signatures for the methods that help calculate the cost. Fill in their calculations:

```
private decimal CalculateCostOfDecorations() { ... }

private decimal CalculateCostOfBeveragesPerPerson() { ... }
```

These should be very similar to the methods you already wrote at the start of the chapter.

- **❹ Add the read-only `Cost` property to calculate the cost.**
Add a property called `Cost` that calculates the cost of the dinner party:

```
public decimal Cost {
    get {
        // Fill in the code to calculate the cost
    }
}
```

Here's a hint. Start with a decimal variable called totalCost, then use the compound operators += and *= to modify its value before returning the final cost.

- **❺ Update the form to use the properties.**
  Here's the complete code for the form. It uses the constructor and the three properties (`NumberOfPeople`, `FancyDecoration`, and `HealthyOption`) to pass information into the object, and it uses the `Cost` property to calculate the cost.

```
public partial class Form1 : Form
{
    DinnerParty dinnerParty;
    public Form1()
    {
        InitializeComponent();
        dinnerParty = new DinnerParty((int)numericUpDown1.Value,
                                      healthyBox.Checked, fancyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void fancyBox_CheckedChanged(object sender, EventArgs e)
    {
        dinnerParty.FancyDecorations = fancyBox.Checked;
        DisplayDinnerPartyCost();
    }

    private void healthyBox_CheckedChanged(object sender, EventArgs e)
    {
        dinnerParty.HealthyOption = healthyBox.Checked;
        DisplayDinnerPartyCost();
    }

    private void numericUpDown1_ValueChanged(object sender, EventArgs e)
    {
        dinnerParty.NumberOfPeople = (int)numericUpDown1.Value;
        DisplayDinnerPartyCost();
    }

    private void DisplayDinnerPartyCost()
    {
        decimal Cost = dinnerParty.Cost;
        costLabel.Text = Cost.ToString("c");
    }
}
```

The form stores an instance of DinnerParty and updates its properties every time the number of people or party options change.

The form uses the `DinnerParty` object's constructor to initialize it with the right values. You'll need to make sure your `DinnerParty` class has this constructor.

This method updates the dinner party cost on the form by accessing the Cost property every time it updates the form.

The form is simpler now because it doesn't need to access the methods that do the calculations. Those calculations are encapsulated behind the `Cost` property.

EXERCISE SOLUTION

This idea is called "separation of concerns," and it's a good way to think about your programs. The form concerns itself with the user interface, while the DinnerParty object concerns itself with the cost calculation.

Did you notice how your new form doesn't need to do very much? All it does is set properties on objects based on user input, and change its ouput based on those properties. Think about how the code for user input and output is separated from the code that does the calculation.

```
class DinnerParty {
    public const int CostOfFoodPerPerson = 25;

    public int NumberOfPeople { get; set; }

    public bool FancyDecorations { get; set; }

    public bool HealthyOption { get; set; }

    public DinnerParty(int numberOfPeople, bool healthyOption, bool fancyDecorations) {
        NumberOfPeople = numberOfPeople;
        FancyDecorations = fancyDecorations;
        HealthyOption = healthyOption;
    }

    private decimal CalculateCostOfDecorations() {
        decimal costOfDecorations;
        if (FancyDecorations)
        {
            costOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        }
        else
        {
            costOfDecorations = (NumberOfPeople * 7.50M) + 30M;
        }
        return costOfDecorations;
    }

    private decimal CalculateCostOfBeveragesPerPerson() {
        decimal costOfBeveragesPerPerson;
        if (HealthyOption)
        {
            costOfBeveragesPerPerson = 5.00M;
        }
        else
        {
            costOfBeveragesPerPerson = 20.00M;
        }
        return costOfBeveragesPerPerson;
    }

    public decimal Cost {
        get {
            decimal totalCost = CalculateCostOfDecorations();
            totalCost += ((CalculateCostOfBeveragesPerPerson()
                                + CostOfFoodPerPerson) * NumberOfPeople);
            if (HealthyOption)
            {
                totalCost *= .95M;
            }
            return totalCost;
        }
    }
}
```

These properties are set in the constructor and updated by the form, and they're used when calculating the cost.

Here's the DinnerParty constructor. It sets the three properties based on the values passed into it by the form.

By making this method private, you made sure that it can't be accessed from outside of the class, which will keep it from being misused.

The private methods used in the cost calculation access the properties so that they have the latest information from the form.

You had a `SetHealthyOption()` method in the first version of this program. Now it's changed to a property called `HealthyOption`.

If you have a method that starts with "Set" that sets a field and then updates the state of the object, changing it to a property could make it more obvious how you expect it to be used.

That's one way encapsulation makes your classes easier to understand and reuse later.

Now that the calculations are private and encapsulated behind the Cost property, there's no way for the form to recalculate the cost of the decorations that doesn't use the current options. That'll fix the bug that almost cost Kathleen one of her best clients!

SHARPEN YOUR PENCIL SOLUTION

Write down the value of the `amountOwed` variable after the code below executed.

1.
```
CableBill january = new CableBill(4);
MessageBox.Show(january.CalculateAmount(7).ToString());
```

What's the value of `amountOwed`?

28

2.
```
CableBill february = new CableBill(7);
february.payPerViewDiscount = 1;
MessageBox.Show(february.CalculateAmount(3).ToString());
```

What's the value of `amountOwed`?

won't compile

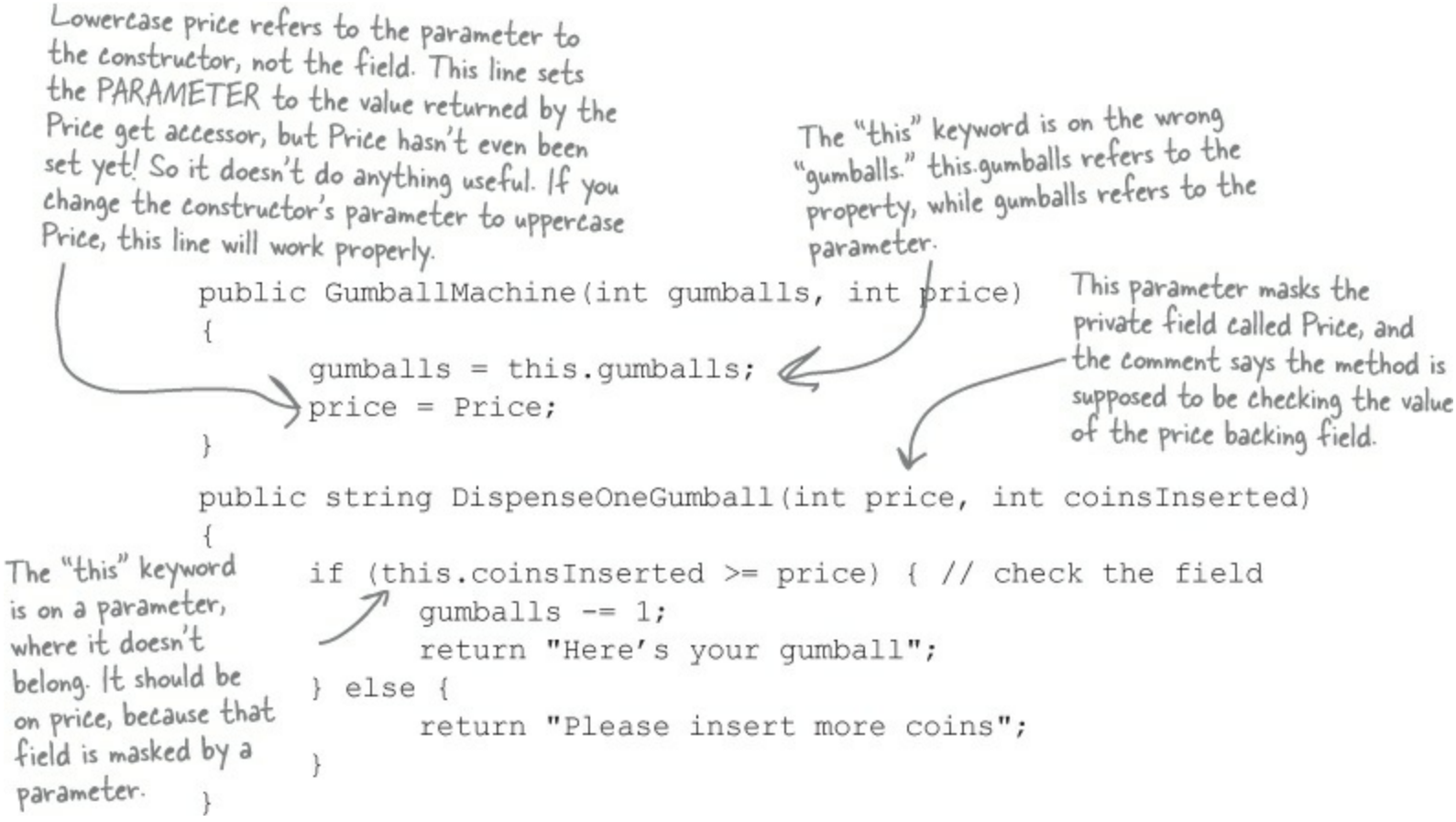
3.
```
CableBill march = new CableBill(9);
march.Discount = true;
MessageBox.Show(march.CalculateAmount(6).ToString());
```

What's the value of `amountOwed`?

42

SHARPEN YOUR PENCIL SOLUTION

This code has problems. Write down what you think is wrong with the code, and what you'd change.




```
public GumballMachine(int gumballs, int price)
{
        gumballs = this.gumballs;
        price = Price;
}
public string DispenseOneGumball(int price, int coinsInserted)
{
    if (this.coinsInserted >= price) { // check the field
        gumballs -= 1;
        return "Here's your gumball";
    } else {
        return "Please insert more coins";
    }
}
```

Take an extra minute or two and really look at this code. These are some of the most common mistakes that new programmers make when working with objects, and avoiding them makes it much more satisfying to write code.

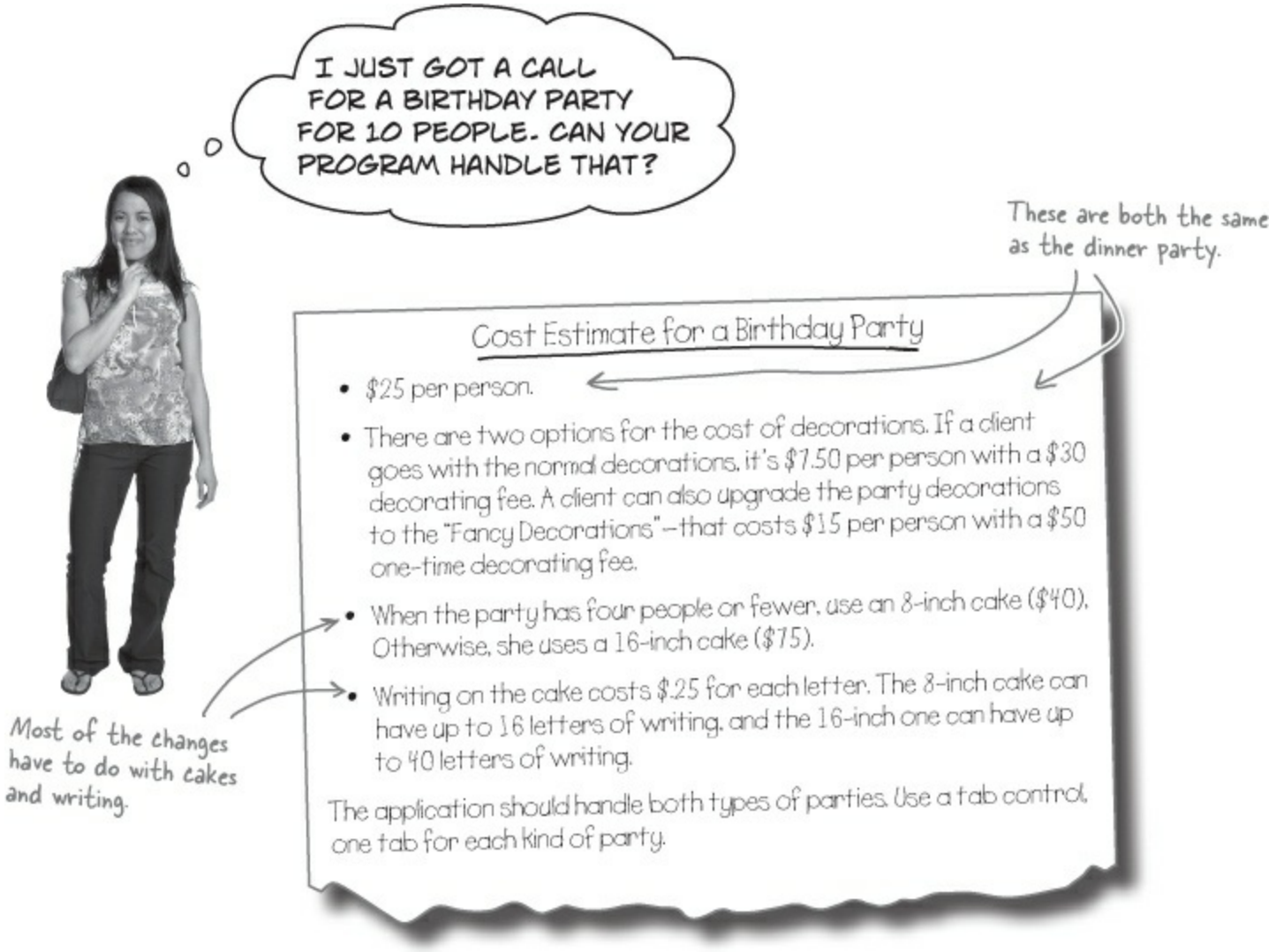
Chapter 6. Inheritance: Your object's family tree

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**.

Kathleen does birthday parties, too

Now that you got your program working, Kathleen is using it all the time. But she doesn't just handle dinner parties — she does birthdays too, and they're priced a little differently. She'll need you to add birthdays to her program.



BRAIN POWER

There's no healthy option for birthday parties. Can you think of how this could lead to bugs if you start out a project by copying and pasting code from the `DinnerParty` class from the last chapter?

We need a BirthdayParty class

Modifying your program to calculate the cost of Kathleen's birthday parties means adding a new class and changing the form to let you handle both kinds of parties.

Here's what we're going to do:

- **❶ CREATE A NEW CLASS FOR BIRTHDAY PARTIES.**

> **NOTE**
>
> You'll do all this in a minute — but first you'll need to get a sense of what the job involves.

Your new class will need to calculate the costs, deal with decorations, and check the size of the writing on the cake.

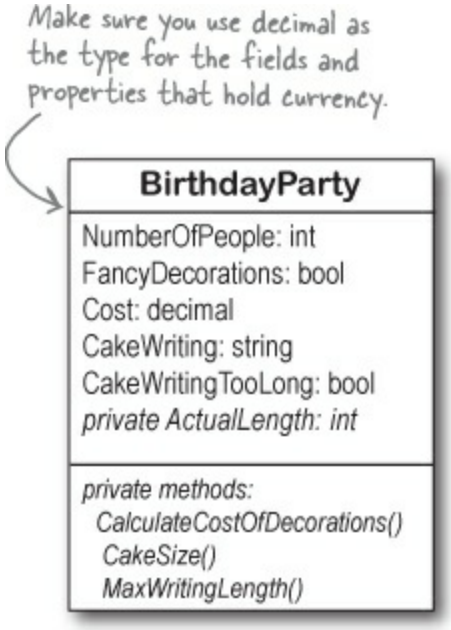
- **❷ ADD A TAB CONTROL TO YOUR FORM.**
  Each tab on the form is a lot like the `GroupBox` control you used to choose which guy placed the bet in the Betting Parlor lab. Just click on the tab you want to display, and drag controls into it.

- **❸ LABEL THE FIRST TAB AND MOVE THE DINNER PARTY CONTROLS INTO IT.**
  You'll drag each of the controls that handle the dinner party into the new tab. They'll work exactly like before, but they'll only be displayed when the dinner party tab is selected.
- **❹ LABEL THE SECOND TAB AND ADD NEW BIRTHDAY PARTY CONTROLS TO IT.**
  You'll design the interface for handling birthday parties just like you did for the dinner parties.
- **❺ WIRE YOUR BIRTHDAY PARTY CLASS UP TO THE CONTROLS.**
  Now all you need to do is add a `BirthdayParty` reference to the form's fields, and add the code to each of your new controls so that it uses its methods and properties.

> **THERE ARE NO DUMB QUESTIONS**
>
> Q: **Q: Why can't we just create a new instance of `DinnerParty`, like Mike did when he wanted to compare three routes in his navigation program?**
>
> A: **A:** Because if you created another instance of the `DinnerParty` class, you'd only be able to use it to plan extra dinner parties. Two instances of the same class can be really useful if you need to manage two different pieces of the same kind of data. But if you need to store **different kinds of data**, you'll need **different classes** to do it.
>
> Q: **Q: How do I know what to put in the new class?**
>
> A: **A:** Before you can start building a class, you need to know what problem it's supposed to solve. That's why you had to talk to Kathleen — she's going to be using the program. Good thing you took a lot of notes! You can come up with your class's methods, fields, and properties by thinking about its behavior (what it ***needs to do***) and its state (what it ***needs to know***).

Build the Part y Planner version 2.0

Start a new project — we're going to build Kathleen a new version of her program that handles birthdays *and* dinner parties. We'll start by creating a well-encapsulated `BirthdayParty` class to do the actual calculation.



DO THIS!

- **❶ ADD THE NEW BIRTHDAYPARTY CLASS TO YOUR PROGRAM.**
  You already know how you'll handle the `NumberOfPeople` and `FancyDecorations` properties — they're just like their counterparts in `DinnerParty`. We'll start by creating your new class and adding those, and then we'll add the rest of the behavior.

  Add the `CostOfFoodPerPerson` constant, and the `NumberOfPeople` and `FancyDecorations` properties. You'll also need a **private `int` property** called `ActualLength`. (Yes, properties can be private, too!)

```
class BirthdayParty
{
    public const int CostOfFoodPerPerson = 25;

    public int NumberOfPeople { get; set; }

    public bool FancyDecorations { get; set; }

    public string CakeWriting { get; set; }

    public BirthdayParty(int numberOfPeople,
                         bool fancyDecorations, string cakeWriting)
    {
        NumberOfPeople = numberOfPeople;
        FancyDecorations = fancyDecorations;
        CakeWriting = cakeWriting;
    }
```

We made this const public because we might want to display it to the user in the future. Was that a good choice? Would you make it private?

When the BirthdayParty object is initialized, it needs to know the number of people, the kind of decorations, and the writing on the cake, so it can start out with the right cake cost when the Cost property is accessed.

The constructor sets up the object's state by setting the properties so that it can calculate the cost later.

You'll add this CakeWriting property on the next page.

  The `ActualLength` property calculates the actual length of the field by comparing the `CakeWriting` property against `MaxWritingLength()`. If the cake writing is too long, it returns the maximum allowable length for the cake.
  The program uses the size of the cake (which varies based on the number of people) and the maximum number of letters that will

fit on the cake (based on the cake size). You'll add two methods to calculate these things, `CakeSize()` and `MaxWritingLength()`.

Properties can be private, too. This property only has a get accessor, which calculates the actual length of the writing to use for the calculation. Private property names typically start with a capital letter even though they're private.

If the writing is too long for the cake, the private ActualLength property calculates the actual number of letters that will fit on the cake.

```
private int ActualLength
{
    get
    {
        if (CakeWriting.Length > MaxWritingLength())
            return MaxWritingLength();
        else
            return CakeWriting.Length;
    }
}

private int CakeSize() {
    if (NumberOfPeople <= 4)
        return 8;
    else
        return 16;
}

private int MaxWritingLength()
{
    if (CakeSize() == 8)
        return 16;
    else
        return 40;
}
```

This if/else block checks the length of the writing and updates the actualLength field with the number of letters that will fit on the cake.

Did you notice how we **left out** some of the brackets? When you only have one statement in a code block, you don't need to add curly brackets around it.

We made ActualLength a private read-only property because we wanted to include an example of how that would look (including the PascalCase naming convention). Would it make more sense as a private method? That wouldn't change the way the program functions. But would it make the code easier to understand? Would someone reading the code wonder why you used a property for ActualLength but a method for CakeSize()? As your code gets more complex, you'll find that there are an almost unlimited number of ways that you can solve a problem. Your job is to make good decisions about your code.

CURLY BRACKETS ARE OPTIONAL FOR SINGLE-LINE BLOCKS

A lot of times you'll have an if statement or while loop that's just got a single statement inside its block. When that happens a lot, you can end up with a whole lot of curly brackets — and that can be a real eyesore! C# helps you avoid that problem by letting you drop the curly brackets if there's just one statement. So this is perfectly valid syntax for a loop and an if statement:

for (int i = 0; i < 10; i++)	if (myValue == 36)
DoTheJob(i);	myValue *= 5;

Keep on going with the `BirthdayParty` class...

- Finish off the `BirthdayParty` class by adding the `Cost` property. But instead of taking the decoration cost and adding the cost of beverages (which is what happens in `DinnerParty`), it'll add the cost of the cake.

```
    public bool CakeWritingTooLong
    {
        get
        {
            if (CakeWriting.Length > MaxWritingLength())
                return true;
            else
                return false;
        }
    }

    private decimal CalculateCostOfDecorations()
    {
        decimal costOfDecorations;
        if (FancyDecorations)
            costOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        else
            costOfDecorations = (NumberOfPeople * 7.50M) + 30M;
        return costOfDecorations;
    }

    public decimal Cost
    {
        get
        {
            decimal totalCost = CalculateCostOfDecorations();
            totalCost += CostOfFoodPerPerson * NumberOfPeople;
            decimal cakeCost;
            if (CakeSize() == 8)
                cakeCost = 40M + ActualLength * .25M;
            else
                cakeCost = 75M + ActualLength * .25M;
            return totalCost + cakeCost;
        }
    }
}
```

This property returns true if the writing is too long for the cake. We'll use it to display a "TOO LONG" message to Kathleen.

This property only has a get accessor, because it doesn't change the state of the object at all. It just uses the fields and methods to calculate a bool value.

This method is just like the one in the DinnerParty class.

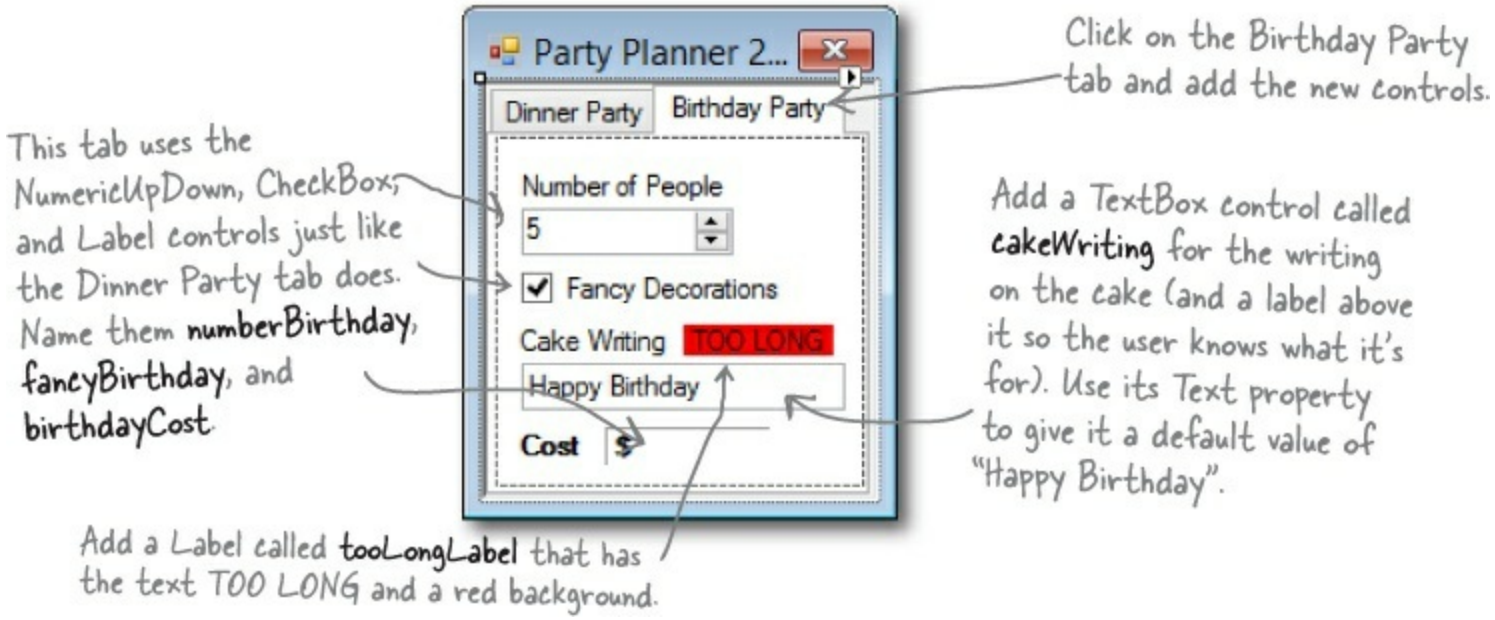
The BirthdayParty class has a decimal Cost property, just like DinnerParty. But it does a different calculation that uses the CakeSize() method and actualLength field (which is set by the CakeWriting property).

Flip back a page and take a closer look at how the CakeWriting property sets the actualLength field. If the writing is too long for the cake, it sets actualLength to the number of letters that actually fit on the cake. Once the writing hits its maximum length, the cost stops going up.

- **❷ USE A TABCONTROL TO ADD TABS TO THE FORM.**
  Drag a `TabControl` out of the toolbox and onto your form, and resize it so it takes up the entire form. Change the text of each tab using the `TabPages` property: a "..." button shows up in the Properties window next to the property. When you click it, the IDE pops up a window that lets you edit the properties of each tab. Set the `Text` property of the tabs to "Dinner Party" and "Birthday Party".
- **❸ PASTE THE DINNER PARTY CONTROLS ONTO THEIR TAB.**
  Open up the Party Planner program from Chapter 5 in another IDE window. Select the controls on the form, copy them, and **paste them into the new Dinner Party tab**. You'll need to click **inside** the tab to make sure they get pasted into the right place (otherwise you'll get an error about not being able to add a component to a container of type `TabControl`).

One thing to keep in mind here: when you copy and paste a control into a form, you're only adding the control itself, **not the event handlers for the control**. And you'll need to check to make sure that `(Name)` is set correctly in the Properties window for each of them. Make sure that each control has the same name as it did in your Chapter 5 project, and then double-click on each control after you add it to add a new empty event handler.

- **❹ BUILD THE BIRTHDAY PARTY USER INTERFACE.**
  The Birthday Party GUI has a `NumericUpDown` control for the number of people, a `CheckBox` control for fancy decorations, and a `Label` control with a 3D border for the cost. Then you'll add a `TextBox` control for the cake writing.



Keep on going with the code for the form...

The form creates a DinnerParty object, so you'll need to copy the DinnerParty class into your project. You can use Add→Existing Item... in the Solution Explorer to add the DinnerParty.cs file — but make sure you change the namespace to match your new project.

- **❺ PUT IT ALL TOGETHER.**

All the pieces are there — now it's just a matter of writing a little code to make the controls work.

You'll need fields in your form that have references to a `BirthdayParty` object and a `DinnerParty` object, and you'll need to instantiate them in the constructor.

You already have code for the dinner party controls' event handlers — they're in your Chapter 5 project. If you haven't double-clicked on the `NumericUpDown` and `CheckBox` controls in the Dinner Party tab to add the event handlers, do it now. Then copy the contents of each event handler from the Chapter 5 program and paste them in here. Here's the code for the form:

```
public partial class Form1 : Form {
    DinnerParty dinnerParty;
    BirthdayParty birthdayParty;
    public Form1() {
        InitializeComponent();
        dinnerParty = new DinnerParty((int)numericUpDown1.Value,
                                      healthyBox.Checked, fancyBox.Checked);
        DisplayDinnerPartyCost();

        birthdayParty = new BirthdayParty((int)numberBirthday.Value,
                                    fancyBirthday.Checked, cakeWriting.Text);
        DisplayBirthdayPartyCost();
    }

    // The fancyBox, healthyBox, and numericUpDown1 event handlers and
    // the DisplayPartyDinnerCost() method are identical to the ones in
    // the Dinner Party exercise at the end of Chapter 5.
```

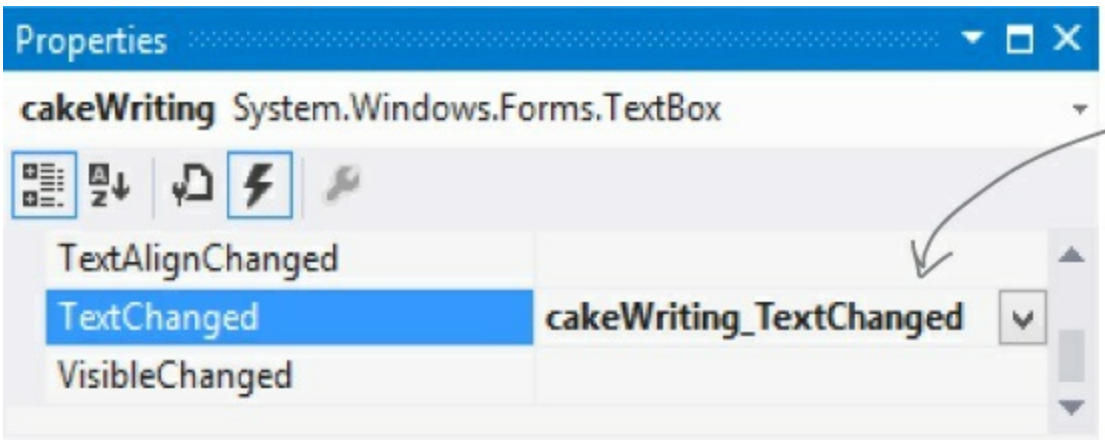
The BirthdayParty instance is initialized in the form's constructor, just like the instance of DinnerParty.

Add code to the `NumericUpDown` control's event handler method to set the object's `NumberOfPeople` property, and make the Fancy Decorations checkbox work.

```
private void numberBirthday_ValueChanged(object sender, EventArgs e)
{
    birthdayParty.NumberOfPeople = (int)numberBirthday.Value;
    DisplayBirthdayPartyCost();
}

private void fancyBirthday_CheckedChanged(object sender, EventArgs e)
{
    birthdayParty.FancyDecorations = fancyBirthday.Checked;
    DisplayBirthdayPartyCost();
}
```

The CheckBox and NumericUpDown controls' event handlers are just like the ones for the dinner party.

Use the Events page in the Properties window to add a new `TextChanged` event handler to the `cakeWriting` TextBox. Click on the lightning bolt button in the Properties window to switch to the Events page. Then select the TextBox and scroll down until you find the `TextChanged` event. Double-click on it to add a new event handler for it.



When you select the cakeWriting TextBox and double-click on the TextChanged row in the Events page of the Properties window, the IDE will add a new event handler that gets fired every time the text in the box changes.

```
private void cakeWriting_TextChanged(object sender, EventArgs e)
{
    birthdayParty.CakeWriting = cakeWriting.Text;
```

```
        DisplayBirthdayPartyCost();
    }
```

Add a `DisplayBirthdayPartyCost()` method and add it to all of the event handlers so the cost label is updated automatically any time there's a change.

Controls have a Visible property that causes them to appear on or disappear from the form.

The BirthdayParty class exposes this property so the form can display a warning.

The way that the form handles the cake writing can be really simple because the BirthdayParty class is **well encapsulated**. All the form has to do is use its controls to set the properties on the object, and the object takes care of the rest.

```
    private void DisplayBirthdayPartyCost() {
        tooLongLabel.Visible = birthdayParty.CakeWritingTooLong;
        decimal cost = birthdayParty.Cost;
        birthdayCost.Text = cost.ToString("c");
    }
}
```

All the intelligence for dealing with the writing, the number of people, and the cake size is built into the NumberOfPeople and CakeWriting set accessors, so the form just has to set and display the values.

...and you're done with the form!

- **❻ YOUR PROGRAM'S DONE...TIME TO RUN IT!**
  Make sure the program works the way it's supposed to. Check that it pops up a message box if the writing is too long for the cake. Make sure the price is always right. If it's working, you're done!

Start up the program and go to the Dinner Party tab. Make sure that it works just like your old Party Planner program.

Does the calculation work correctly? In this case, 10 people means £25 per person (£250) plus £75 for a 16" cake plus £7.50 per person (£75) for the non-fancy decorations plus a £30 decorating fee plus £.25 per letter for 21 letters on the cake (£5.25).

Click on the Birthday Party tab. Make sure the cost changes when you change the number of people or click the Fancy Decorations checkbox.

So £250 + £75 + £75 + £30 + £5.25 = £435.25. It works!

When you type in the Cake Writing text box, the TextChanged event handler should update the cost every time you add or remove a letter.

If the cake writing is too long for the cake, the BirthdayParty class sets its CakeWritingTooLong property to true and calculates the cost with the maximum length. The form doesn't need to do any calculation at all.

One more thing...can you add a $100 fee for parties over 12?

Kathleen's gotten so much business using your program that she can afford to charge a little more for some of her larger clients. So what would it take to change your program to add in the extra charge?

- Change the `DinnerParty.Cost` property to check `NumberOfPeople` and add $100 to the return value if it's over 12.
- Do the exact same thing for the `BirthdayParty.Cost` property.

Take a minute and think about how you'd add a fee to both the `DinnerParty` and `BirthdayParty` classes. What code would you write? Where would it have to go?

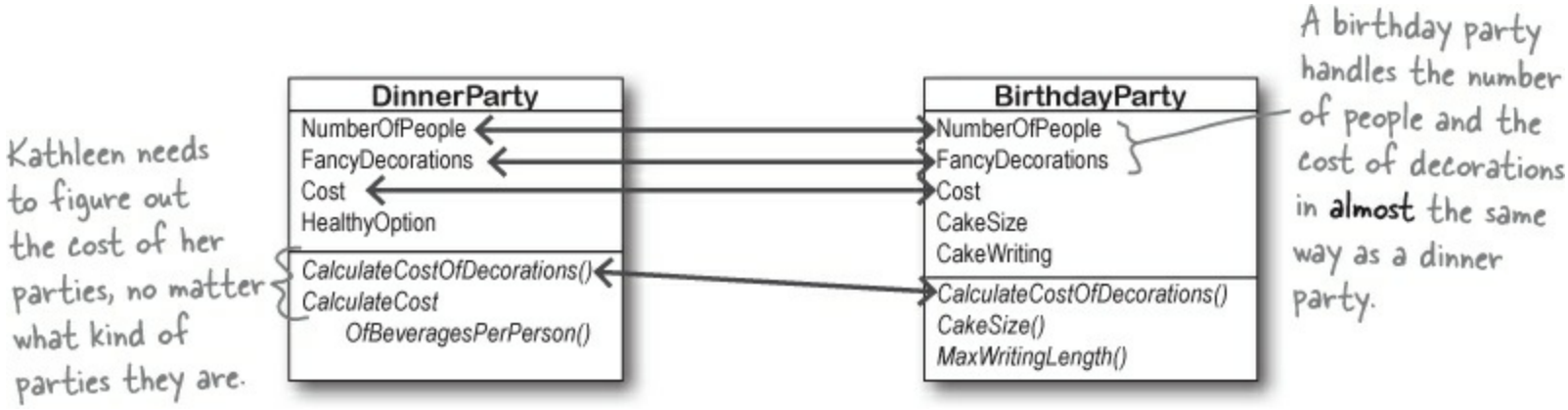
Easy enough...but what happens if there are three similar classes? Or four? Or twelve? And what if you had to maintain that code and make more changes later? What if you had to make the ***same exact change*** to five or six ***closely related*** classes?

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***.

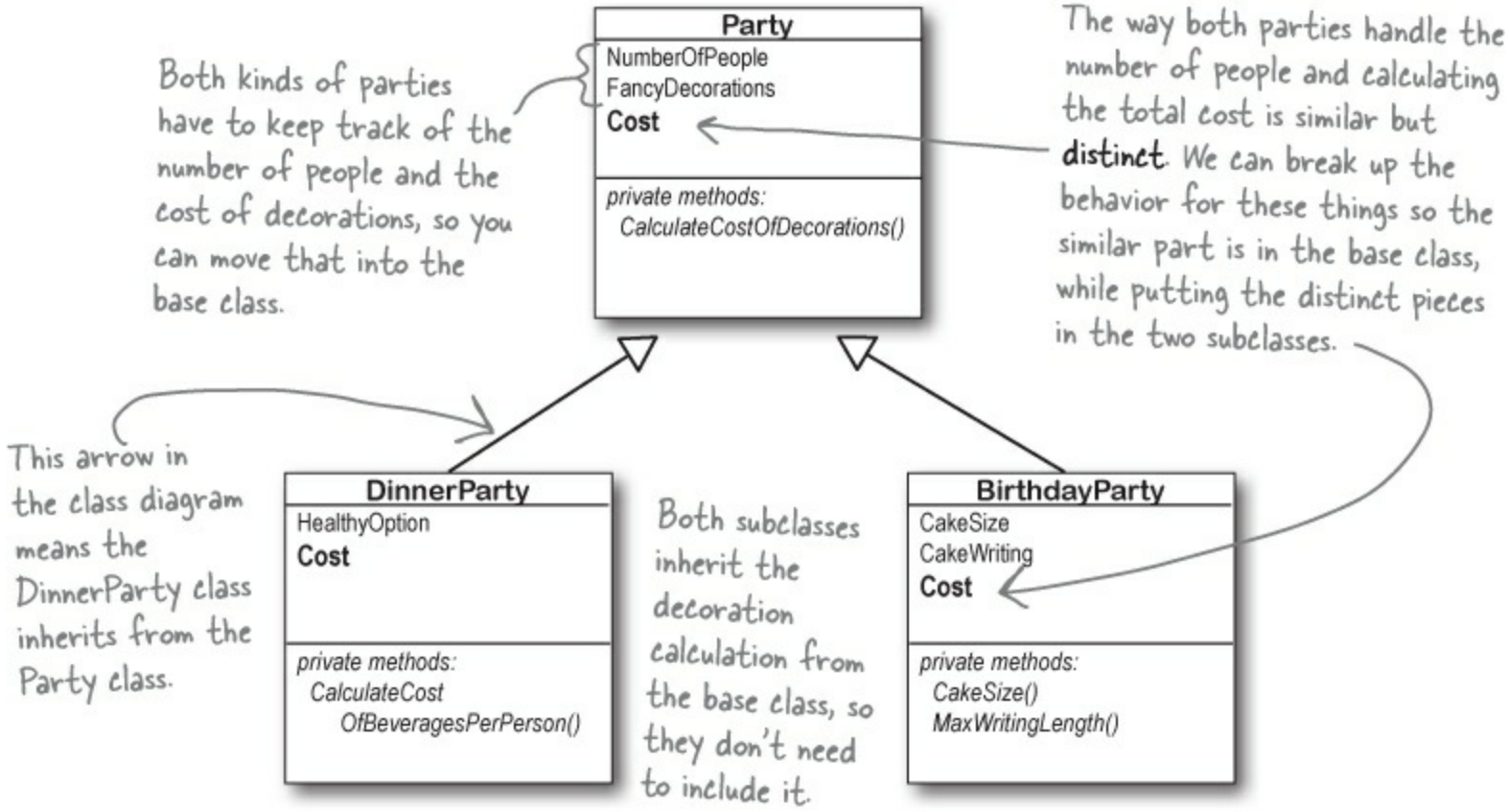
When your classes use inheritance, you only need to write your code once

It's no coincidence that your `DinnerParty` and `BirthdayParty` 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 — a birthday party and a dinner party — have **similar behaviors**.



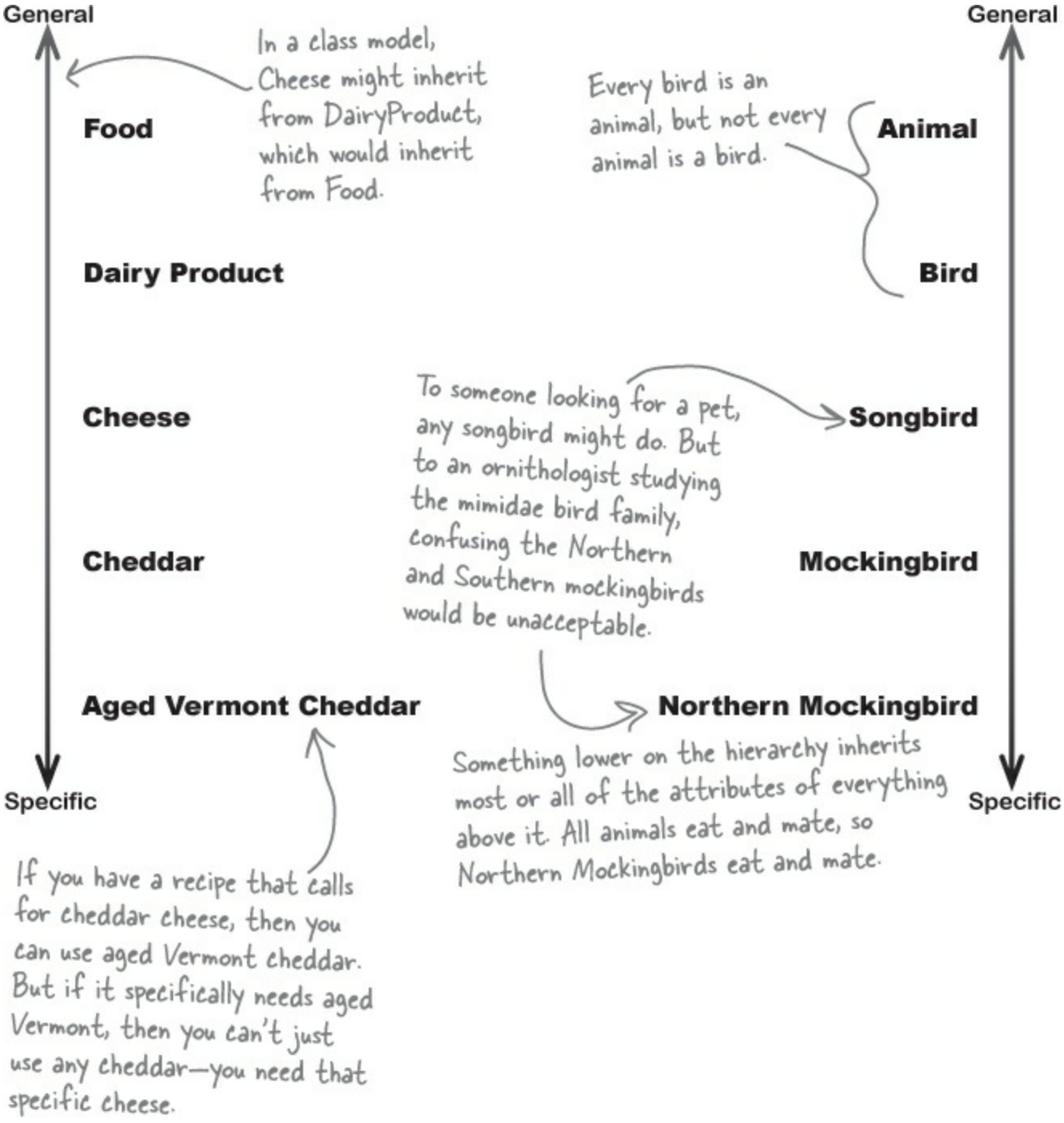
Dinner parties and birthday parties are both parties

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

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.



NOTE

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 cat. Your job is to design a program that simulates a zoo. (Don't get too excited — we're not going to actually build the code, just design the classes to represent the animals.)

We'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.

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.

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.

- **❶ LOOK FOR THINGS THE ANIMALS HAVE IN COMMON.**
  Take a look at these six animals. What do a lion, a hippo, a tiger, a cat, 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.

- **❷ 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`.

Use inheritance to avoid duplicate code in subclasses

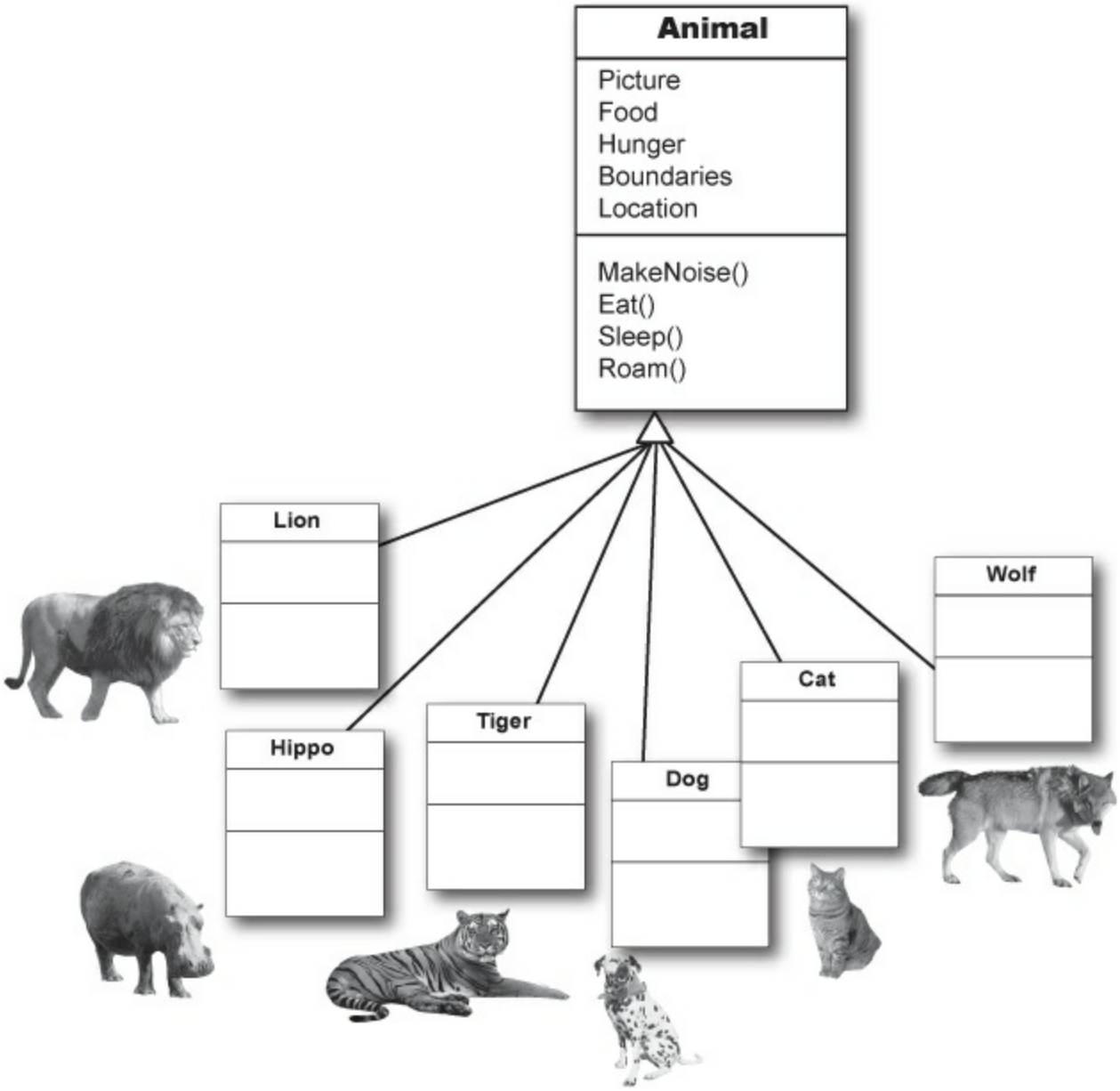
You already know that duplicate code sucks. It's hard to maintain, and always leads to headaches down the road. So let's choose fields and methods for an `Animal` base class that you **only have to write once**, and each of the animal subclasses can inherit from them. Let's start with the public fields:

- `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.

> **NOTE**
>
> Choosing a base class is about making choices. You could have decided to use a ZooOccupant class that defines the feed and maintenance costs, or an Attraction class with methods for how the animals entertain the zoo visitors. But we think Animal makes the most sense here. Do you agree?


Animal
Picture
Food
Hunger
Boundaries
Location
MakeNoise()
Eat()
Sleep()
Roam()
Lion
Hippo
Tiger
Dog
Cat
Wolf

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!

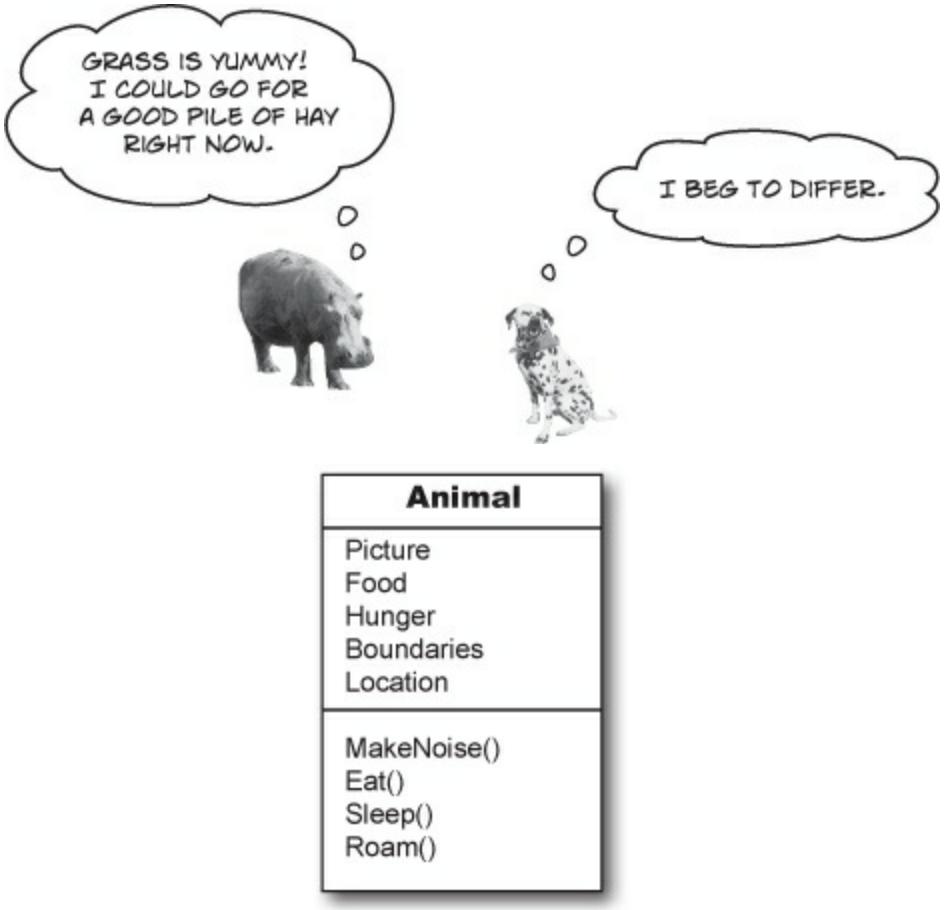
- **❸ FIGURE OUT WHAT EACH ANIMAL DOES THAT THE ANIMAL CLASS DOES DIFFERENTLY-OR NOT AT ALL.**
  What does each type of animal do that all the other animals don't? Dogs eat dog food, so the dog's `Eat()` method will need to override the `Animal.Eat()` method. Hippos swim, so a hippo will have a `Swim()` 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 little 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.



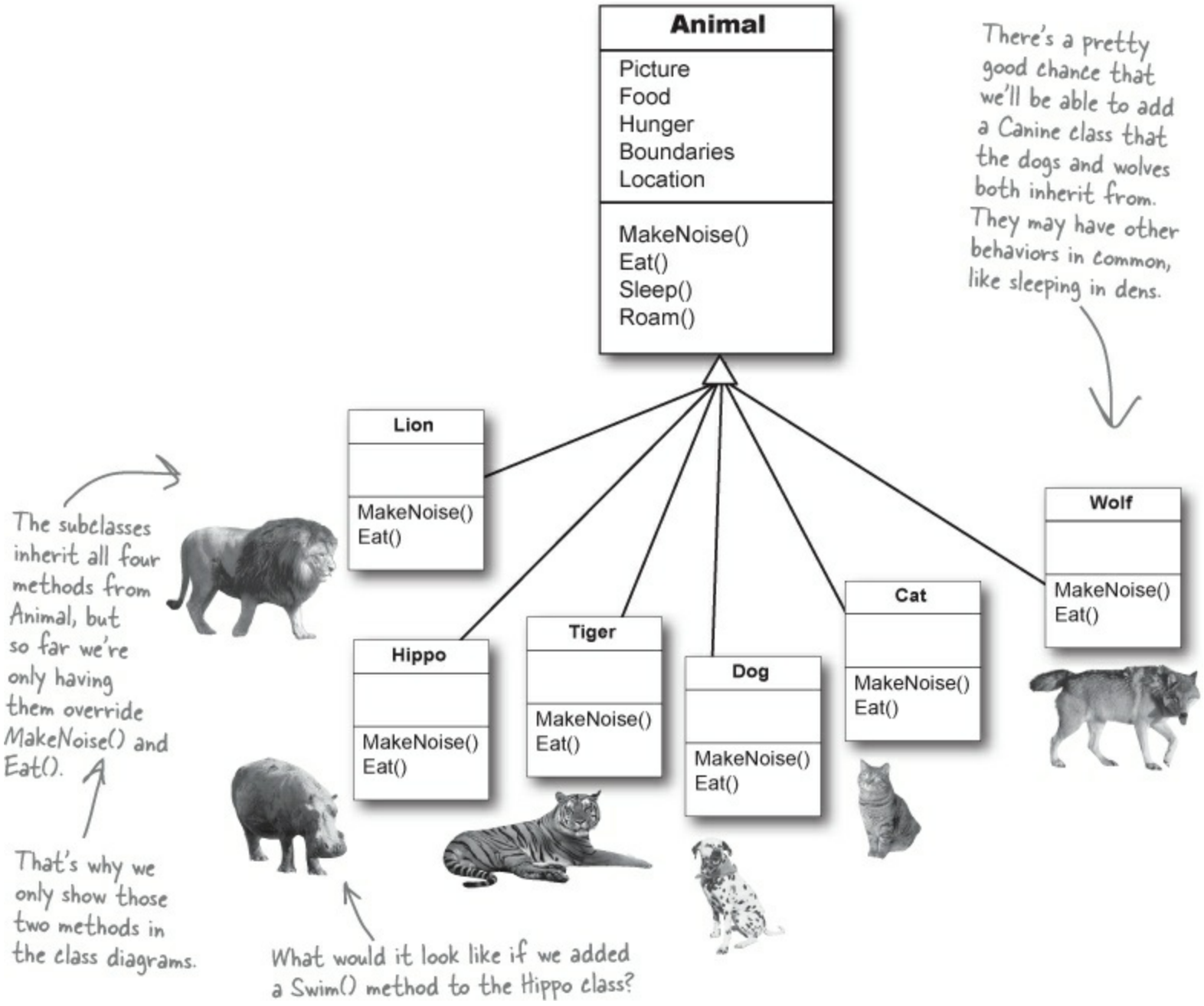
BRAIN POWER

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.

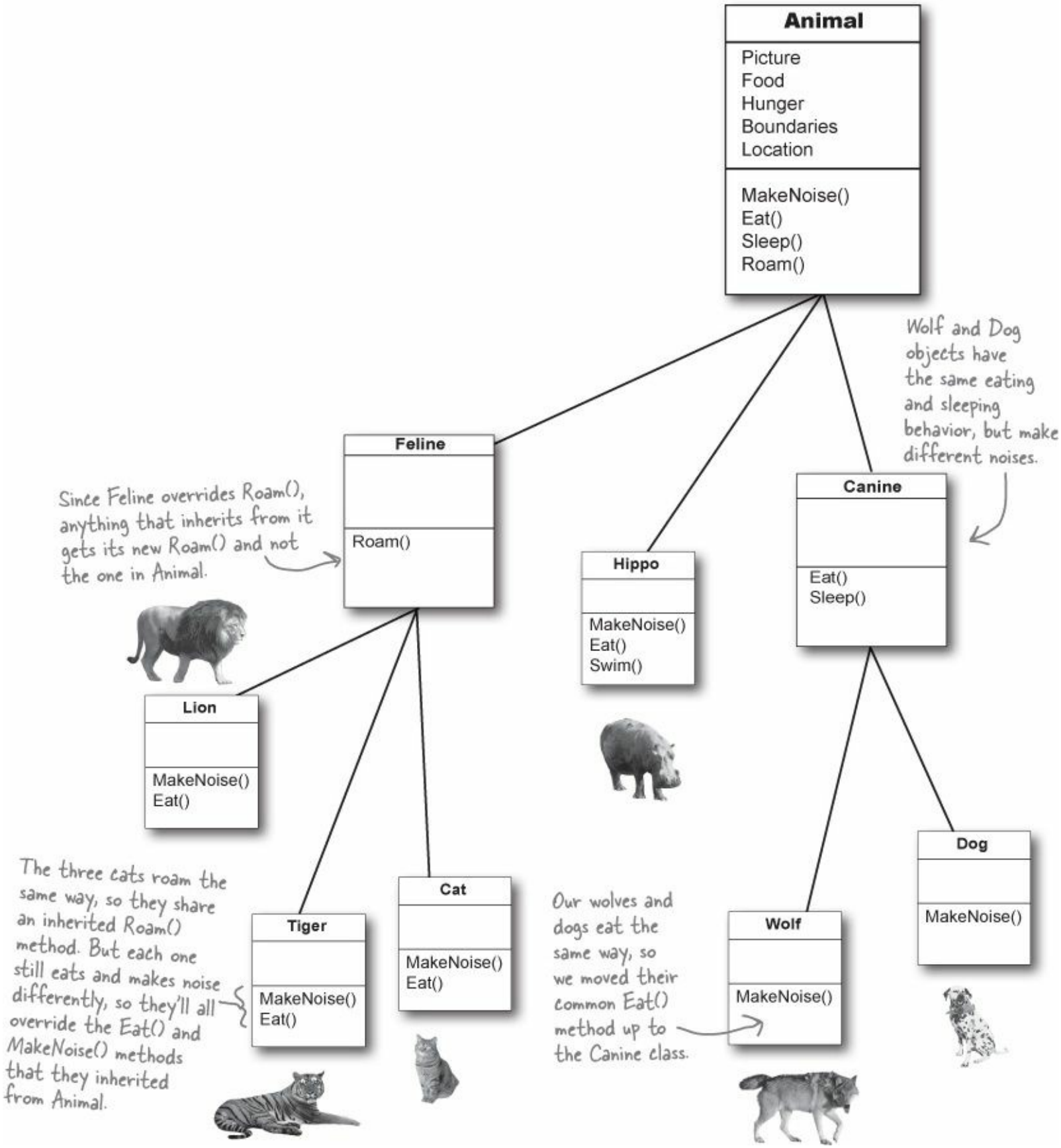
- **❹ LOOK FOR CLASSES THAT HAVE A LOT IN COMMON.**
  Don't dogs and wolves seem pretty similar? They're both canines, and it's a good bet that if you look at their behavior they have a lot in common. They probably eat the same food and sleep the same way. What about domestic cats, tigers, and lions? 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 cat 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.

- **❺ FINISH YOUR CLASS HIERARCHY.**
  Now that you know how you'll organize the animals, you can add the `Feline` and `Canine` classes.



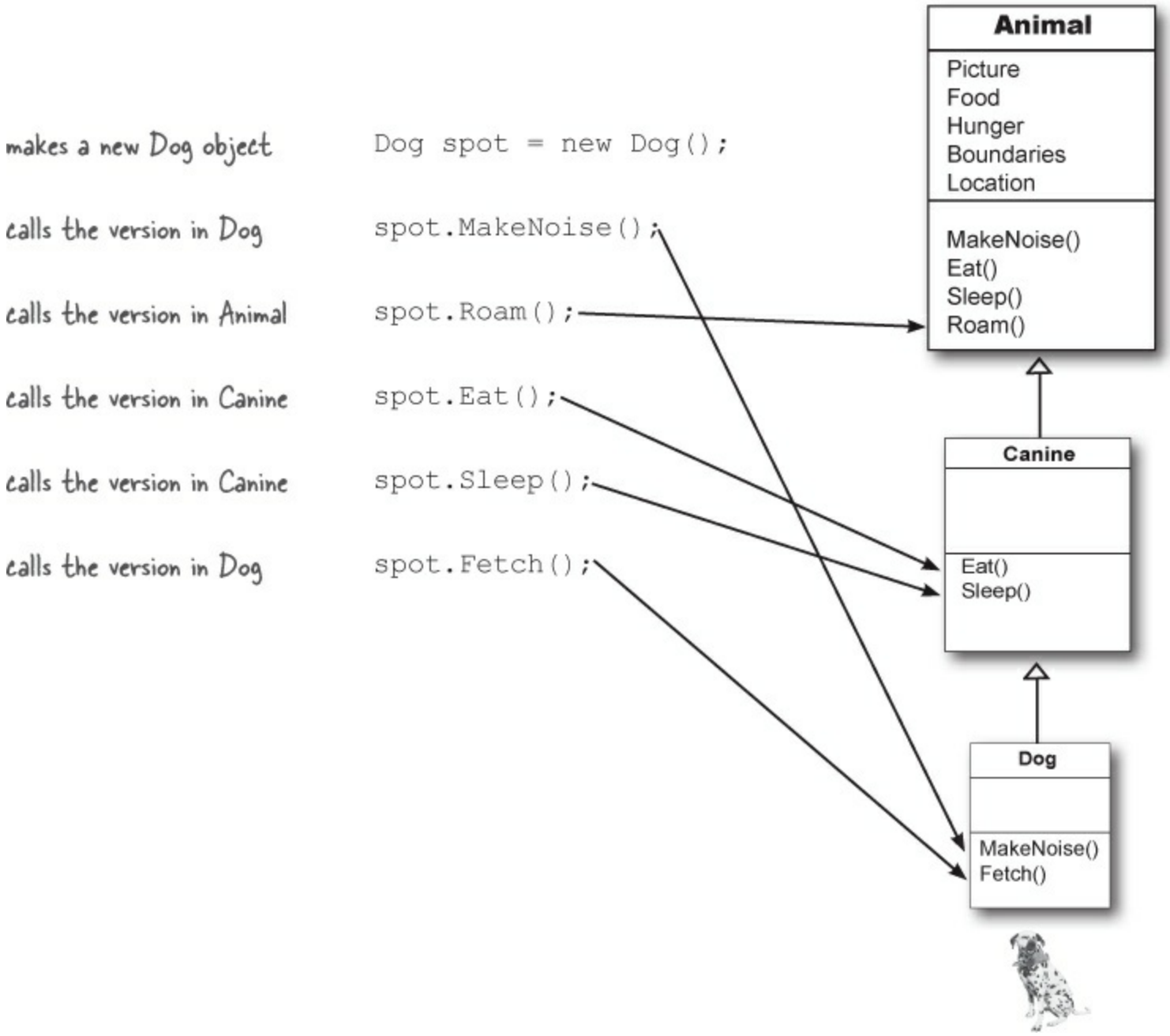
NOTE

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 his way up from the mailroom to the top of the corporate* ***hierarchy***.

Every subclass extends its base class

You're not limited to the methods that a subclass inherits from its base class...but you already know that! After all, you've been building your own classes all along. When you 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.



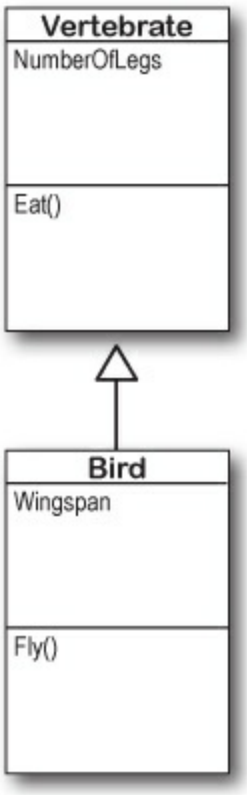
C# always calls the most specific method

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`.

Use a colon to inherit from 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.




```
class Vertebrate
{
    public int NumberOfLegs;
    public void Eat() {
        // code to make it eat
    }
}
```

The Bird class uses a colon to inherit from the Vertebrate class. This means that it inherits all of the fields, properties, and methods from Vertebrate.

```
class Bird : Vertebrate
{
    public double Wingspan;
    public void Fly() {
        // code to make the bird fly
    }
}
```

You inherit a class by adding a colon to the end of the class declaration, followed by the base class to inherit from.

tweety is an instance of Bird, so it's got the Bird methods and fields as usual.

```
public button1_Click(object sender, EventArgs e) {
    Bird tweety = new Bird();
    tweety.Wingspan = 7.5;
    tweety.Fly();
    tweety.NumberOfLegs = 2;
    tweety.Eat();
}
```

Since the Bird class inherits from Vertebrate, every instance of Bird also has the fields and methods defined in the Vertebrate class.

When a subclass inherits from a base class, all of the fields, properties, and methods in the base class are automatically added to the subclass.

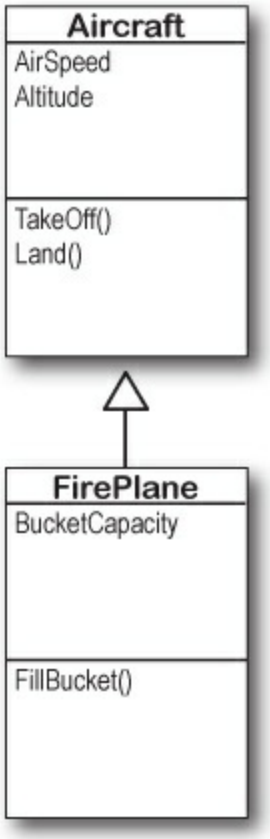
THERE ARE NO DUMB QUESTIONS

Q: **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?**

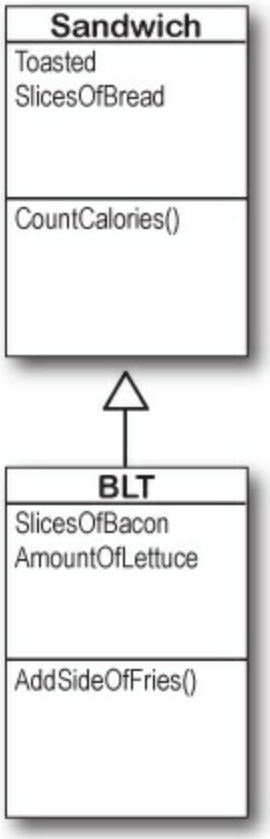
A: **A:** It might look better, but it wouldn't be as accurate. When you set up a class to inherit from another one, you build that relationship into the subclass — the base class remains the same. And that makes sense when you think about it from the perspective of the base class. Its behavior is completely 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.

SHARPEN YOUR PENCIL

Take a look at these class models and declarations, and then circle the statements that won't work.



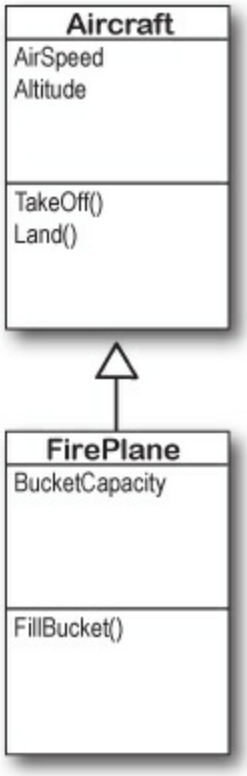

```
class Aircraft {
   public double AirSpeed;
   public double Altitude;
   public void TakeOff() { ... };
   public void Land() { ... };
}

class FirePlane : Aircraft {
   public double BucketCapacity;
   public void FillBucket() { ... };
}

public void FireFightingMission() {
   FirePlane myFirePlane = new FirePlane();
   new FirePlane.BucketCapacity = 500;
   Aircraft.Altitude = 0;
   myFirePlane.TakeOff();
   myFirePlane.AirSpeed = 192.5;
   myFirePlane.FillBucket();
   Aircraft.Land();
}
```




```
class Sandwich {
   public boolean Toasted;
   public int SlicesOfBread;
   public int CountCalories() { ... }
}

class BLT : Sandwich {
   public int SlicesOfBacon;
   public int AmountOfLettuce;
   public int AddSideOfFries() { ... }
}

public BLT OrderMyBLT() {
   BLT mySandwich = new BLT();
   BLT.Toasted = true;
   Sandwich.SlicesOfBread = 3;
   mySandwich.AddSideOfFries();
   mySandwich.SlicesOfBacon += 5;
   MessageBox.Show("My sandwich has "
      + mySandwich.CountCalories + "calories.");
   return mySandwich;
}
```

SHARPEN YOUR PENCIL SOLUTION

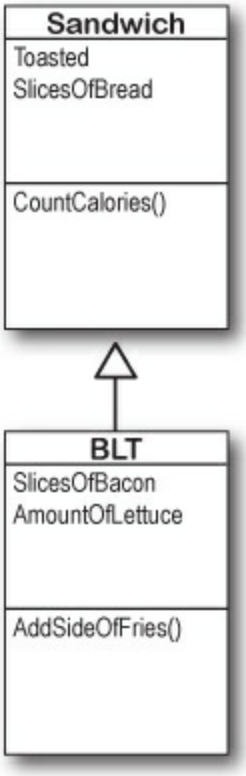
Take a look at these class models and declarations, and then circle the statements that won't work.




```
class Aircraft {
    public double AirSpeed;
    public double Altitude;
    public void TakeOff() { ... };
    public void Land() { ... };
}

class FirePlane : Aircraft {
    public double BucketCapacity;
    public void FillBucket() { ... };
}

public void FireFightingMission() {
    FirePlane myFirePlane = new FirePlane();
    new FirePlane.BucketCapacity = 500;
    Aircraft.Altitude = 0;
    myFirePlane.TakeOff();
    myFirePlane.AirSpeed = 192.5;
    myFirePlane.FillBucket();
    Aircraft.Land();
}
```

That's not how you use the "new" keyword.

These statements all use the class names instead of the name of the instance, myFirePlane.



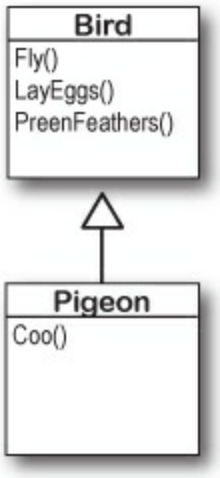

```
class Sandwich {
    public boolean Toasted;
    public int SlicesOfBread;
    public int CountCalories() { ... }
}

class BLT : Sandwich {
    public int SlicesOfBacon;
    public int AmountOfLettuce;
    public int AddSideOfFries() { ... }
}

public BLT OrderMyBLT() {
    BLT mySandwich = new BLT();
    BLT.Toasted = true;
    Sandwich.SlicesOfBread = 3;
    mySandwich.AddSideOfFries();
    mySandwich.SlicesOfBacon += 5;
    MessageBox.Show("My sandwich has "
        + mySandwich.CountCalories + "calories.");
    return mySandwich;
}
```

These properties are part of the instance, but the statements are trying to call them incorrectly using the class names.

CountCalories is a method, but this statement doesn't include the parentheses () after the call to the method.

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.



NOTE

Pigeon is a subclass of Bird, so any fields and methods in Bird are automatically part of Pigeon, too.

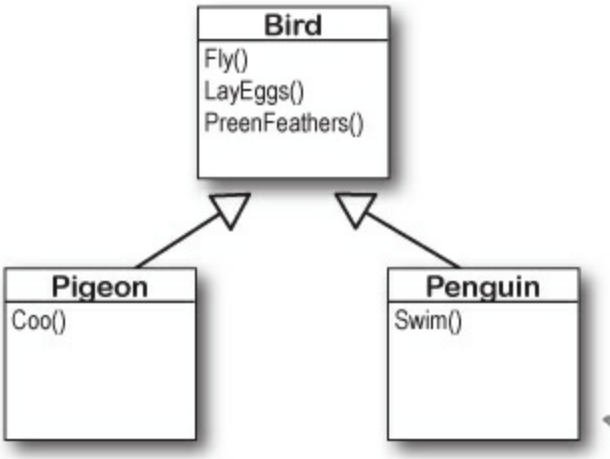
...but some birds don't fly!

What do you do if your base class has a method that your subclass needs to ***modify***?

```
class Bird {
   public void Fly() {
    // here's the code to make the bird fly
   }

   public void LayEggs() { ... };

   public void PreenFeathers() { ... };
}


class Pigeon : Bird {
   public void Coo() { ... }
}


class Penguin : Bird {
   public void Swim() { ... }
}


public void BirdSimulator() {
      Pigeon Harriet = new Pigeon();
      Penguin Izzy = new Penguin();
      Harriet.Fly();
      Harriet.Coo();
      Izzy.Fly();
}
```

Izzy is an instance of Penguin. Since it inherited the Fly() method, there's nothing stopping it from flying.

Both Pigeon and Penguin inherit from Bird, so they both get the Fly(), LayEggs(), and PreenFeathers() methods.



Penguin objects shouldn't be able to fly! But if the Penguin class inherits from Bird, then you'll have penguins flying all over the place. So what do we do?

Pigeons fly, lay eggs, and preen their feathers, so there's no problem with the Pigeon class inheriting from Bird.

BRAIN POWER

If this were your Bird Simulator code, what would you do to keep the penguins from flying?

A subclass can override methods to change or replace methods it inherited

Sometimes you've got a subclass that you'd like to inherit *most* of the behaviors from the base class, but *not all of them*. When you want to change the behaviors that a class has inherited, you can **override** the methods.

- **❶ 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, which tells C# to allow the subclass to override methods.

```
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.

- **❷ ADD A METHOD WITH THE SAME NAME TO THE DERIVED CLASS.**
  You'll need to have exactly the same signature — meaning the same return value and parameters — and you'll need to use the `override` keyword in the declaration.

```
class Penguin : Bird {

    public override void Fly() {

        MessageBox.Show("Penguins can't fly!")

    }
}
```

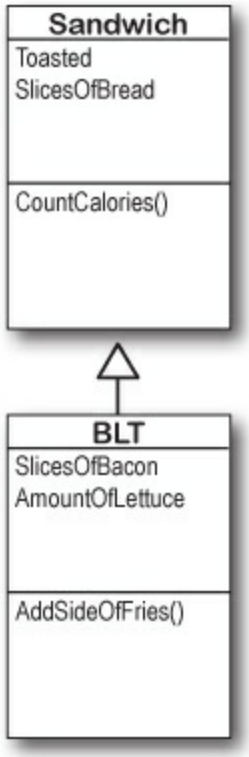
To override the Fly() method, add an identical method to the subclass and use the override keyword.

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.

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.

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 use a subclass in place of the base class it inherits from. So if your `Recipe()` method takes a `Cheese` object and you've got an `AgedVermontCheddar` class that inherits from `Cheese`, then you can pass an instance of `AgedVermontCheddar` to the `Recipe()` method. `Recipe()` only has access to the fields, properties, and methods that are part of the `Cheese` class, though — it doesn't have access to anything specific to `AgedVermontCheddar`.



- ❶ Let's say we have a method to analyze `Sandwich` objects:

```
public void SandwichAnalyzer(Sandwich specimen) {
   int calories = specimen.CountCalories();
   UpdateDietPlan(calories);
   PerformBreadCalculations(specimen.SlicesOfBread, specimen.Toasted);
}
```

- ❷ You could pass a sandwich to the method — but you could also pass a BLT. Since a BLT is a *kind* of sandwich, we set it up so that it inherits from the `Sandwich` class:

```
public button1_Click(object sender, EventArgs e) {
   BLT myBLT = new BLT();
   SandwichAnalyzer(myBLT);
}
```

- ❸ 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.

We'll talk about this more in the next chapter!

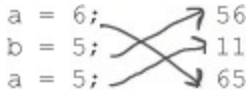
```
public button2_Click(object sender, EventArgs e) {
    Sandwich mySandwich = new Sandwich();
    BLT myBLT = new BLT();
    Sandwich someRandomSandwich = myBLT;
    BLT anotherBLT = mySandwich;     // <--- THIS WON'T COMPILE!!!
}
```

You can assign myBLT to any Sandwich variable because a BLT is a kind of sandwich.

But you can't assign mySandwich to a BLT variable, because not every sandwich is a BLT! That's why this last line will cause an error.

EXERCISE

Mixed Messages



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, ";
  }
}
```

```
class C : B {
  public ___________ string m3() {
    return "C's m3, " + (ivar + 6);
  }
}
```

Here's the entry point for the program—it doesn't show a form, it just pops up a message box.

```
class Mixed5 {
  public static void Main(string[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    A a2 = new C();
    string q = "";
```

Hint: think really hard about what this line really means.

Candidate code goes here (three lines)

```
    System.Windows.Forms.MessageBox.Show(q);
  }
}
```

Code candidates:

```
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();
```

Output:

```
A's m1, A's m2, C's m3, 6
B's m1, A's m2, A's m3,
A's m1, B's m2, C's m3, 6
B's m1, A's m2, C's m3, 13
B's m1, C's m2, A's m3,
A's m1, B's m2, A's m3,
B's m1, A's m2, C's m3, 6
A's m1, A's m2, C's m3, 13
```

(Don't just type this into the IDE — you'll learn a lot more if you figure this out on paper!)

POOL PUZZLE

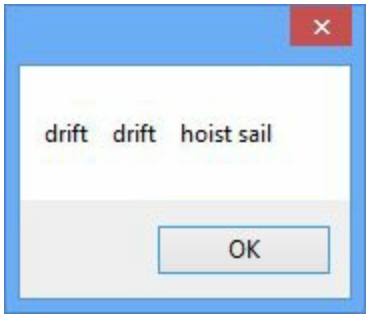
Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code. You may use the same snippet more than once, and you might not need to use all the snippets. Your ***goal*** is to make a set of classes that will compile and run together as a program. Don't be fooled — this one's harder than it looks.

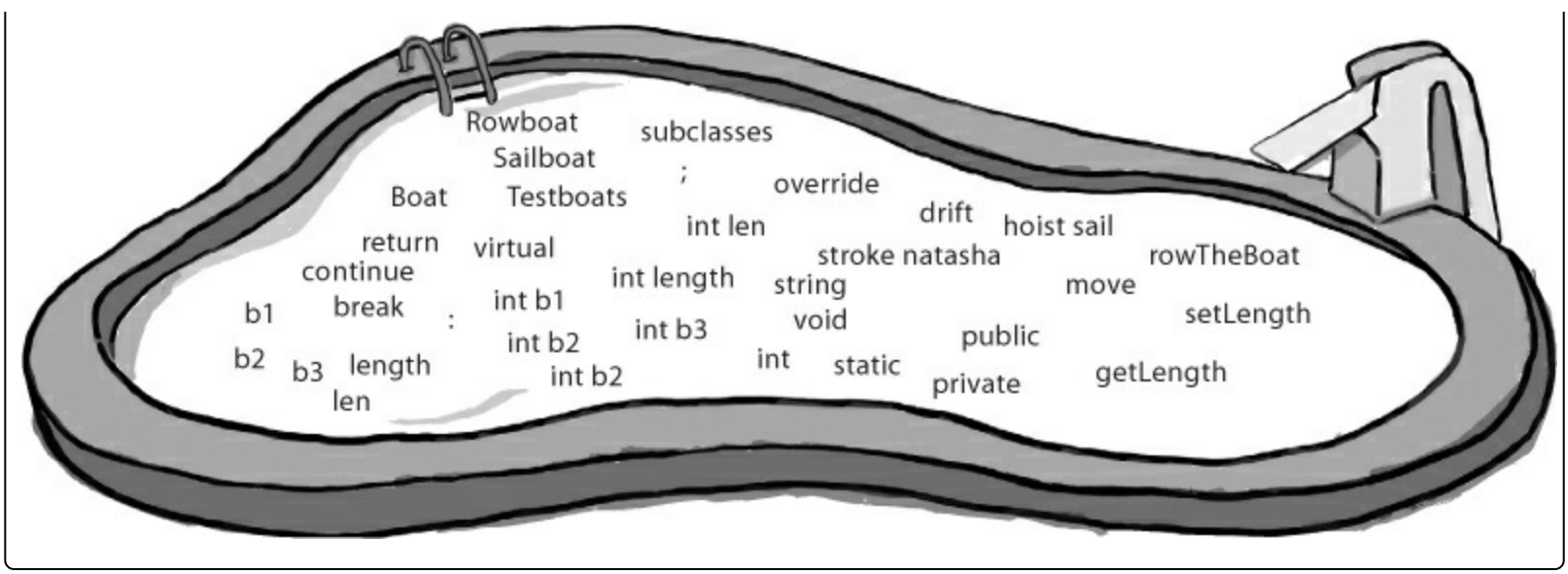
```
class Rowboat ______________ {
   public ________ rowTheBoat() {
      return "stroke natasha";
   }
}
_____________________________________
class _________ {
   private int ___________;
  _______ void _____________ (_______) {
      length = len;
   }
   public int getLength() {
     ________________;
   }
   public _____________ move() {
      return "_________";
   }
}
```

```
class TestBoats {
   ......................... Main(){
     ............ xyz = "";
     ............ b1 = new Boat();
       Sailboat b2 = new ................ ();
       Rowboat ........... = new Rowboat();
       b2.setLength(32);
       xyz = b1. ........... ();
       xyz += b3. ........... ();
       xyz += ..........move();
       System.Windows.Forms.MessageBox.Show(xyz);
   }
}
_____________________________________________
class ................ : Boat {
   public ............................ () {
      return "................ ";
   }
}
```

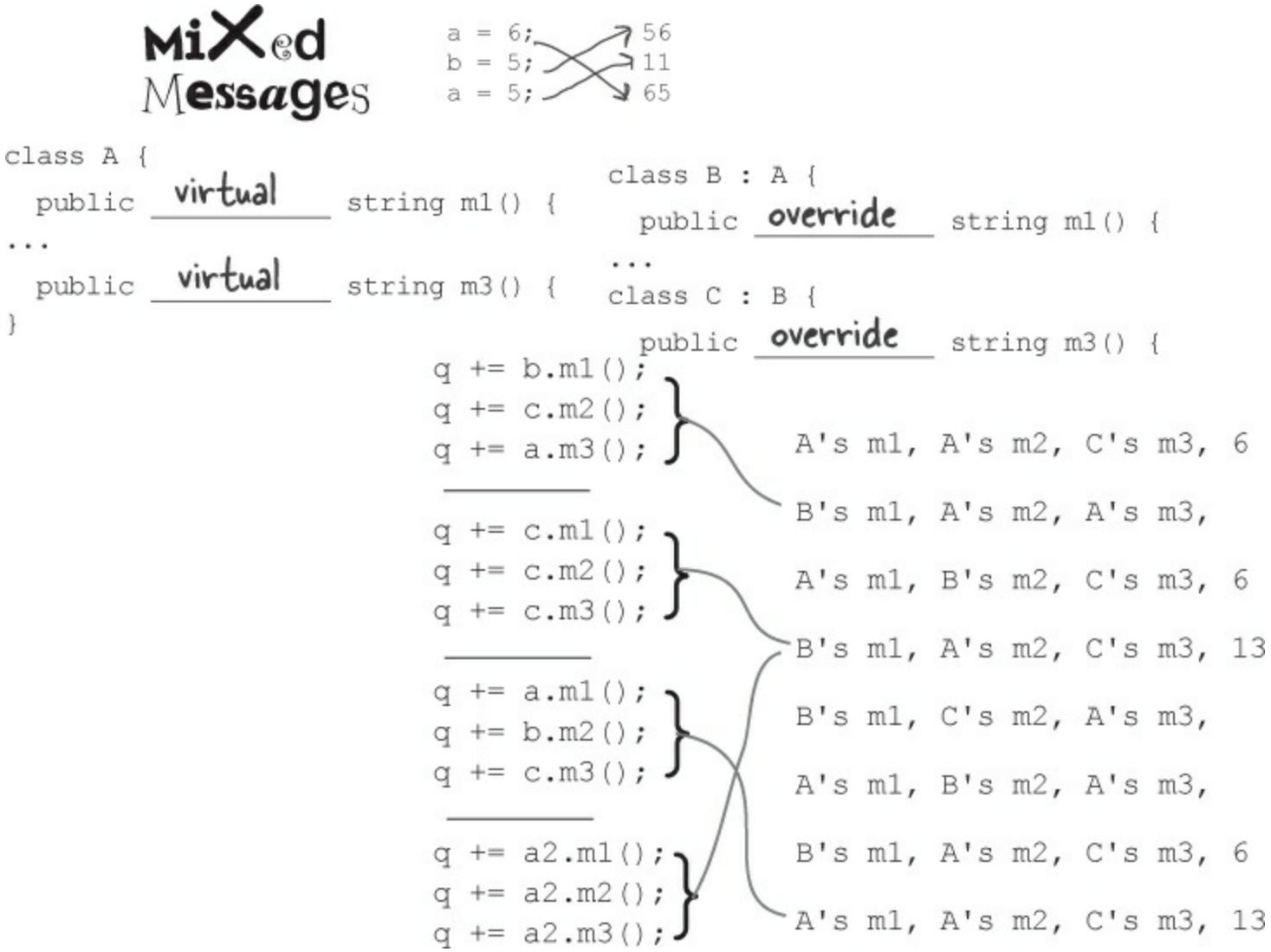
Hint: this is the entry point for the program.

OUTPUT:




Rowboat
subclasses
Sailboat
;
override
Boat
Testboats
drift
hoist sail
int len
return
virtual
stroke natasha
rowTheBoat
continue
int length
string
move
b1
break
:
int b1
void
setLength
int b3
int b2
public
b2
b3
length
int
static
int b2
private
getLength
len

EXERCISE SOLUTION



You can always substitute a reference to a subclass in place of a base class. In other words, you can always use something more specific in place of something more general — so if you've got a line of code that asks for a `Canine`, you can send it a reference to a `Dog`. So this line of code:

```
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 a little hard to understand. Here are a few lines that follow the same pattern, but have names that you can understand:

```
Sandwich mySandwich = new BLT();
Cheese ingredient= new AgedVermontCheddar();
Songbird tweety = new NorthernMockingbird();
```

POOL PUZZLE SOLUTION

```
class Rowboat : Boat {
   public string rowTheBoat() {
      return "stroke natasha";
   }
}
class Boat {
   private int length;
   public void setLength( int len ) {
      length = len;
   }
   public int getLength() {
      return length;
   }
   public virtual string move() {
      return " drift ";
   }
}
```

```
class TestBoats {
   public static void Main(){
      string xyz = "";
      Boat b1 = new Boat();
      Sailboat b2 = new Sailboat();
      Rowboat b3 = new Rowboat();
      b2.setLength(32);
      xyz = b1.move();
      xyz += b3.move();
      xyz += b2.move();
      System.Windows.Forms.MessageBox.Show(xyz);
   }
}
class Sailboat : Boat {
   public override string move() {
      return " hoist sail ";
   }
}
```

THERE ARE NO DUMB QUESTIONS

Q: **Q: About the entry point that you pointed out in the Pool Puzzle — does this mean I can have a program that doesn't have a Form1 form?**

A: **A:** Yes. When you create a new Windows Application project, the IDE creates all the files for that project for you, including *Program.cs* (which contains a static class with an entry point) and *Form1.cs* (which contains an empty form called Form1).
Try this: instead of creating a new Windows Application project, create an empty project by selecting Empty Project instead of Windows Application when you create a new project in the IDE. Then add a class file to it in the Solution Explorer and type in everything in the Pool Puzzle solution. Since your program uses a message box, you need to add a **reference** by right-clicking on References in the Solution Explorer, selecting Add Reference, and choosing System.Windows.Forms. (That's another thing the IDE does for you automatically when you create a Windows Application.) Finally, select Properties from the Project menu and choose the Windows Application output type.
Now run it...you'll see the results! Congratulations, you just created a C# program from scratch.

NOTE

Flip back to Chapter 2 if you need a refresher on Main() and the entry point!

Q: **Q: Can I inherit from the class that contains the entry point?**

A: **A:** Yes. The entry point ***must*** be a static method, but that method ***doesn't have to be*** in a static class. (Remember, the `static` keyword means that the class can't be instantiated, but that its methods are available as soon as the program starts. So in the Pool Puzzle program, you can call `TestBoats.Main()` from any other method without declaring a reference variable or instantiating an object using a `new` statement.)

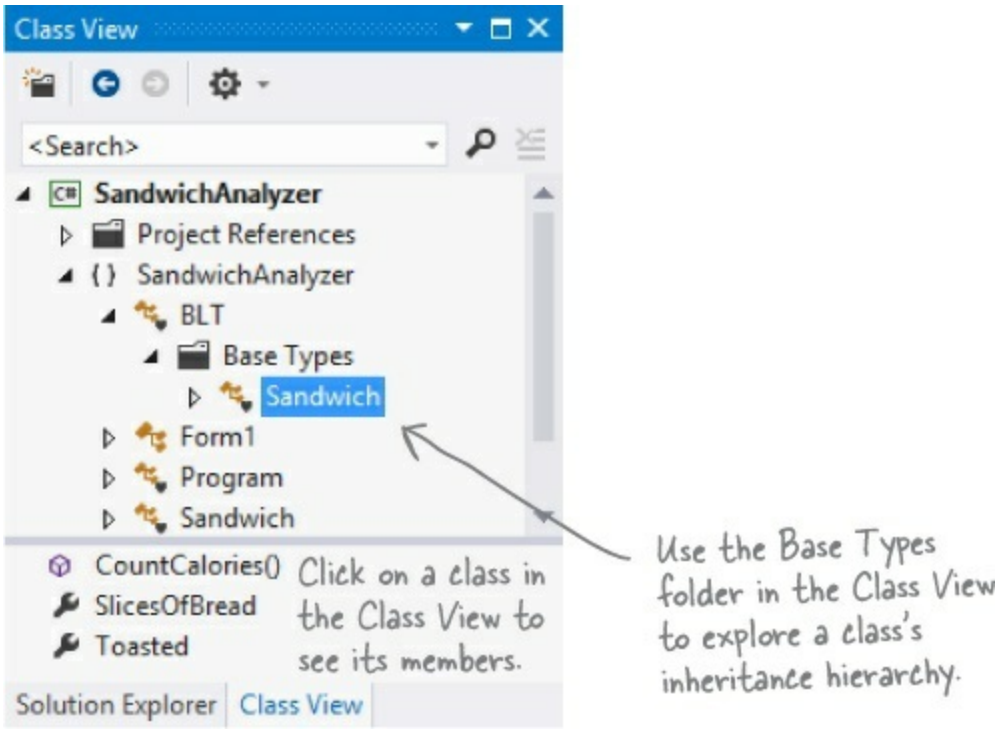
Q: **Q: I still don't get why they're called "virtual" methods — they seem real to me!**

A: **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: **Q: What did you mean by only being able to move up the class diagram but not being able to move down?**

A: **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!

You can show the Class View page using the View menu, and it's yet another tool the IDE gives you to help you explore C#. It's usually docked in the Solution Explorer window, and it lets you explore the classes in your solution — which can come in very handy.



You can always pass an instance of a subclass to any method whose parameters expect a class that it extends.

There's an important reason for `virtual` and `override`!

The `virtual` and `override` keywords aren't just for decoration. They actually make a real difference in how your program works. But don't take our word for it — here's a real example to show you how they work.

CONSOLE APPLICATIONS DON'T USE FORMS

If you create a console application instead of a Windows Forms application, all the IDE creates for you is a new class called Program with an empty Main() entry point method. When you run it, it pops up a command window to display the output. You'll get a lot of practice using console applications over the next few chapters.

DO THIS!

Instead of creating a Windows Forms application, you're going to create a new console application instead! This means it won't have a form.

- **❶ CREATE A NEW CONSOLE APPLICATION AND ADD CLASSES.**
  Right-click on the project in the Solution Explorer and add classes, just like normal. Add the following five classes: `Jewels`, `Safe`, `Owner`, `Locksmith`, and `JewelThief`.
- **❷ ADD THE CODE FOR THE NEW CLASSES.**
  Here's the code for the five new classes you added:

```
class Jewels {
    public string Sparkle() {
        return "Sparkle, sparkle!";
    }
}

class Safe {
    private Jewels contents = new Jewels();
    private string safeCombination = "12345";
    public Jewels Open(string combination)
    {
        if (combination == safeCombination)
            return contents;
        else
            return null;
    }
    public void PickLock(Locksmith lockpicker) {
        lockpicker.WriteDownCombination(safeCombination);
    }
}
```

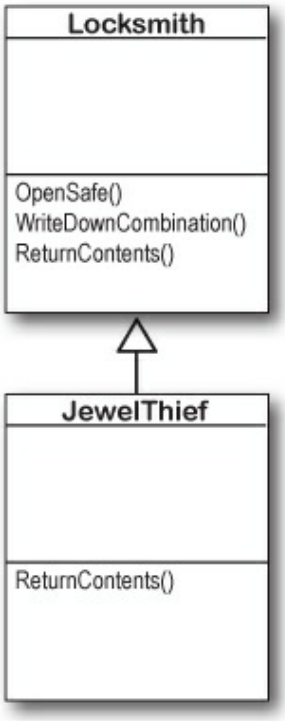
A Safe object keeps a Jewels reference in its contents field. It doesn't return that reference unless Open() is called with the right combination.

Notice how the private keyword hides the contents and combination.

A locksmith can pick the combination lock and get the combination by calling the PickLock() method and passing in a reference to himself. The safe calls his WriteDownCombination() method with the combination.

```
class Owner {
    private Jewels returnedContents;
    public void ReceiveContents(Jewels safeContents) {
        returnedContents = safeContents;
        Console.WriteLine("Thank you for returning my jewels! " + returnedContents.Sparkle());
    }
}
```

- **❸ THE JEWELTHIEF CLASS INHERITS FROM LOCKSMITH.**
  Jewel thieves are locksmiths gone bad! They can pick the lock on the safe, but instead of returning the jewels to the owner, they steal them!




```
class Locksmith {
    public void OpenSafe(Safe safe, Owner owner) {
        safe.PickLock(this);
        Jewels safeContents = safe.Open(writtenDownCombination);
        ReturnContents(safeContents, owner);
    }

    private string writtenDownCombination = null;
    public void WriteDownCombination(string combination) {
        writtenDownCombination = combination;
    }

    public void ReturnContents(Jewels safeContents, Owner owner) {
        owner.ReceiveContents(safeContents);
    }
}
```

A Locksmith's OpenSafe() method picks the lock, opens the safe, and returns the contents to the owner.

```
class JewelThief : Locksmith {
    private Jewels stolenJewels = null;
    public void ReturnContents(Jewels safeContents, Owner owner) {
        stolenJewels = safeContents;
        Console.WriteLine("I'm stealing the contents! " + stolenJewels.Sparkle());
    }
}
```

A JewelThief object inherits the OpenSafe() and WriteDownCombination() methods. But when the OpenSafe() method calls ReturnContents() to return the jewels to the owner, the JewelThief steals them instead!

- **❹ HERE'S THE MAIN() METHOD FOR THE PROGRAM.**
  But ***don't run it just yet!*** Before you run the program, try to figure out what it's going to print to the console.

```
class Program {
    static void Main(string[] args) {
        Owner owner = new Owner();
        Safe safe = new Safe();

        JewelThief jewelThief = new JewelThief();
        jewelThief.OpenSafe(safe, owner);

        Console.ReadKey();
    }
}
```

ReadKey() waits for the user to press a key. It keeps the program from ending.

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 `JewelThief` inherits from `Locksmith`!)

A subclass can hide methods in the superclass

Go ahead and run the JewelThief program. Since it's a console application, instead of writing its console output to the Output window, it'll pop up a command window and print the output there. Here's what you should see:

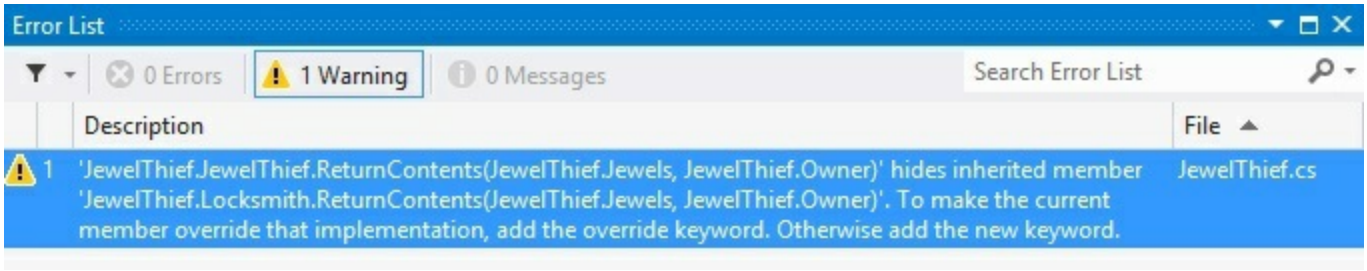
Did you expect the program's output to be different? Maybe something like this:

```
I'm stealing the contents! Sparkle, sparkle!
```

It looks like the `JewelThief` acted just like a `Locksmith`! 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:



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.

> **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 different references to call hidden methods

The `JewelThief` only hides the `ReturnContents()` method (as opposed to overriding it), and that causes it to act like a `Locksmith` object whenever it's called like a `Locksmith` object. `JewelThief` inherits one version of `ReturnContents()` from `Locksmith`, and it defines a second version of it, which means that there are two different methods with the same name. That means your class needs two different ways to call it.

And, in fact, it has exactly that. 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'll call the hidden `Locksmith ReturnContents()` method.

```
// 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);
```

Use the new keyword when you're hiding methods

Take a close look at that warning message. Sure, we never really read most of our warnings, right? But this time, actually read what it says: **To make the current member override that implementation, add the `override` keyword. Otherwise add the `new` keyword.**

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.

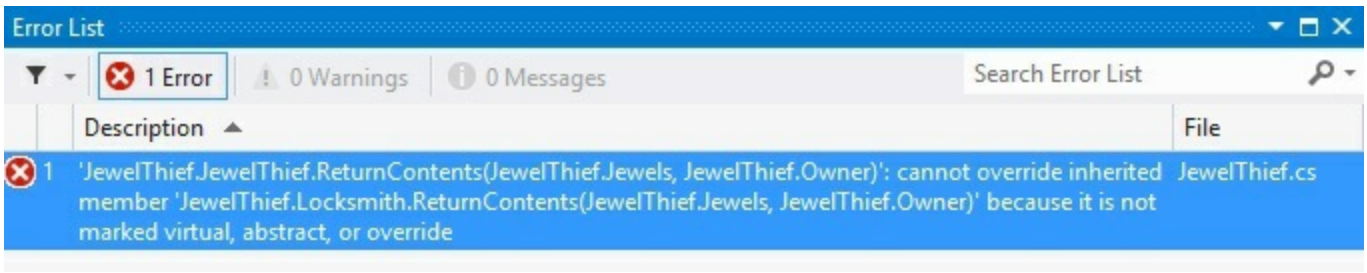
Can you figure out how to get `JewelThief` to override the `ReturnContents()` method instead of just hiding it? See if you can do it before turning to the next page!

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, like this:

```
class JewelThief {
    ...
    override public void ReturnContents
              (Jewels safeContents, Owner owner)
```

But that's not everything you need to do. If you just add that override and try to compile, you'll get an error that looks like this:



Again, take a really close look and actually read the error. `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 {
    ...
    virtual public void ReturnContents
          (Jewels safeContents, Owner owner)
```

Now run your program again. Here's what you should see:

And *that's* the output we were looking for.

Exactly. Most of the time you want to override methods, but hiding them is an option.

When you're working with a subclass that extends a base class, you're much more likely to use overriding than you are to use hiding. So when you see that compiler warning about hiding a method, pay attention to it! Make sure you really want to hide the method, and didn't just forget to use the `virtual` and `override` keywords. If you always use the `virtual`, `override`, and `new` keywords correctly, you'll never run into a problem like this again!

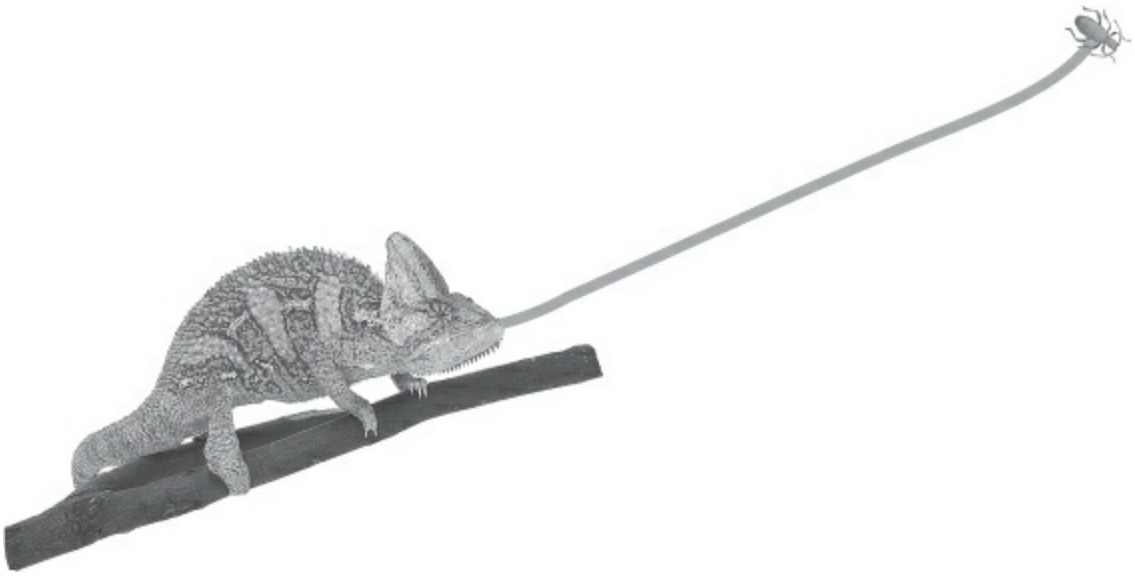
> **If you want to override a method in a base class, always mark it with the virtual keyword, and always use the override keyword any time you want to override the method in a subclass. If you don't, you'll end up accidentally hiding methods instead.**

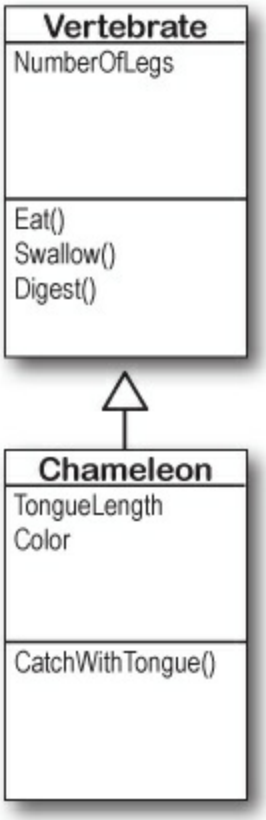
A subclass can access its base class using the base keyword

Even when you override a method or property in your base class, sometimes you'll still want to access it. Luckily, we can use **`base`**, which lets us access any method in the base class.

- ❶ All animals eat, so the `Vertebrate` class has an `Eat()` method that takes a `Food` object as its parameter.

```
class Vertebrate {
   public virtual void Eat(Food morsel) {
      Swallow(morsel);
      Digest();
   }
}
```





- ❷ 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();
   }
}
```

The chameleon needs to swallow and digest the food, just like any other animal. Do we really need to duplicate this code, though?

- ❸ Instead of duplicating the code, we can use the **`base`** keyword to call the method that was overridden. Now we have access to both the old and the new version of `Eat()`.

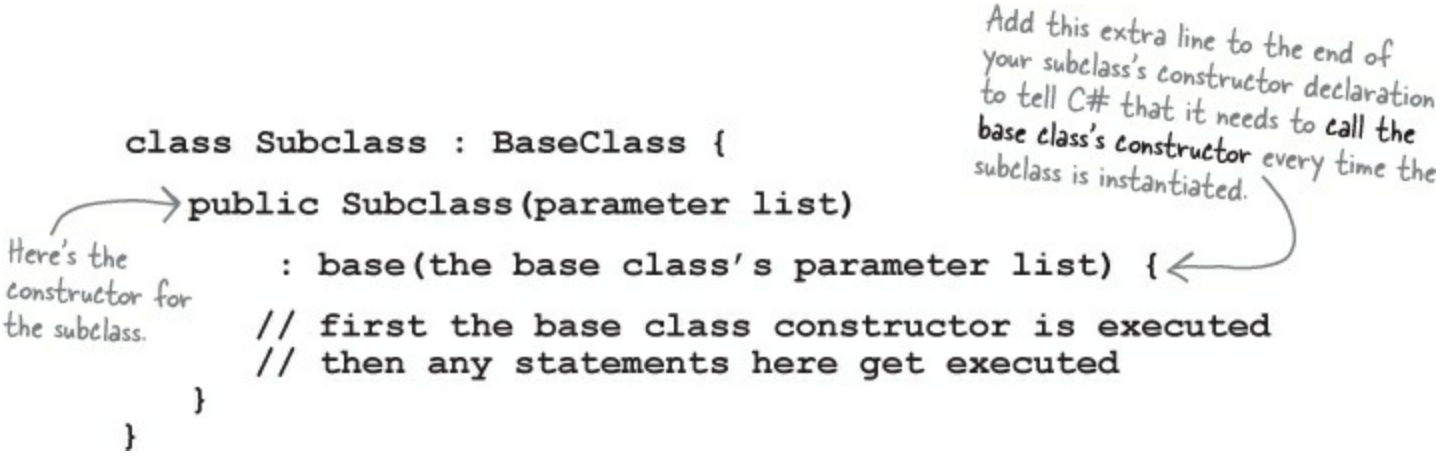
```
class Chameleon : Vertebrate {
   public override void Eat(Food morsel) {
      CatchWithTongue(morsel);
      base.Eat(morsel);
   }
}
```

This line calls the Eat() method in the base class that Chameleon inherited from.

Now that you've had a chance to absorb some of the ideas behind inheritance, here's something to think about. While reusing code is a good way to save keystrokes, another valuable part of inheritance is that it makes it easier to maintain your code later. **Can you think of a reason why that's true?**

When a base class has a constructor, your subclass needs one, too

If your class has constructors that take parameters, then any class that inherits from it **must call one of those constructors**. The subclass's constructor can have different parameters from the base class constructor.



You can call the `new` statement without assigning the result to a variable. The following statement creates an instance of `MySubclass:new MySubclass();` It will be garbage-collected quickly because there's no reference to it.

The base class constructor is executed before the subclass constructor

But don't take our word for it — see for yourself !

DO THIS!

- ❶ Create a base class with a constructor that pops up a message box. Then add a button to a form that instantiates this ***base class*** and shows a message box:

```
using System.Windows.Forms;
class MyBaseClass {
    public MyBaseClass(string baseClassNeedsThis) {
        MessageBox.Show("This is the base class: " + baseClassNeedsThis);
    }
}
```

This is a parameter that the base class constructor needs.

- ❷ Try adding a subclass, but don't call the constructor. Then add a button to a form that instantiates this ***subclass*** and shows a message box:

```
using System.Windows.Forms;
class MySubclass : MyBaseClass{
public MySubclass(string baseClassNeedsThis, int anotherValue) {
    MessageBox.Show("This is the subclass: " + baseClassNeedsThis
        + " and " + anotherValue);
}
}
```

Select Build→Build Solution in the IDE and you'll get an error from this code.

1 'CallBaseClassConstructor.MyBaseClass' does not contain a constructor that takes 0 arguments

Keep an eye out for this error. It means that your subclass didn't call the base constructor.

- ❸ Fix the error by making the constructor call the one from the base class. Then instantiate the subclass and ***see what order*** the two message boxes pop up in!

```
class MySubclass : MyBaseClass{
    public MySubclass(string baseClassNeedsThis, int anotherValue)
        : base(baseClassNeedsThis)
    {
        // the rest of the subclass is the same
```

This is how we send the base class the parameter its constructor needs.

Add this line to tell C# to call the constructor in the base class. It has a parameter list that shows what gets passed to the base class constructor. Then the error will go away and you can make a button to see the two message boxes pop up!

Now you're ready to finish the job for Kathleen!

When you last left Kathleen, you'd finished adding birthday parties to her program. She needs you to **charge an extra $100 for parties over 12**. It seemed like you were going to have to write the same exact code twice, once for each class. Now that you know how to use inheritance, you can have them inherit from the same base class that contains all of their shared code, so you only have to write it once.

If we play our cards right, we should be able to change the two classes without making any changes to the form!

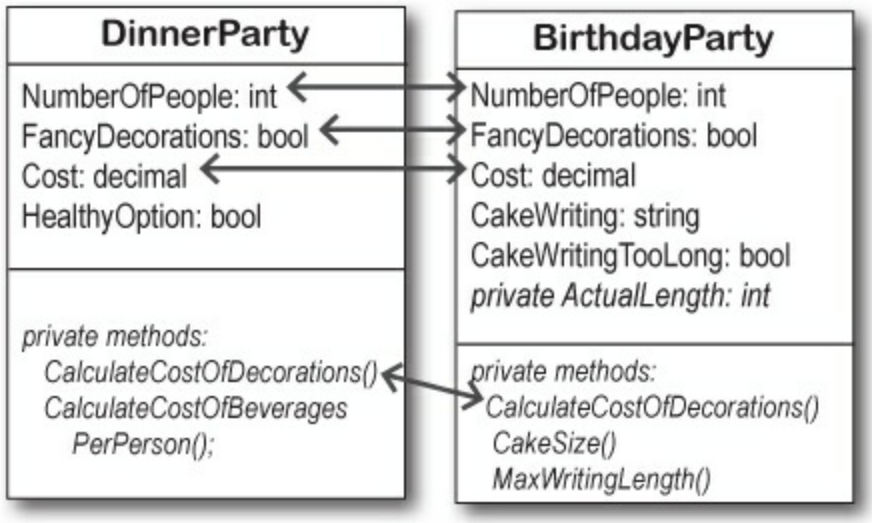
EXERCISE

Finish the job for Kathleen by creating a `Party` base class that has all of the shared behavior from `DinnerParty` and `BirthdayParty`. You're going to be reusing a lot of code from the previous project, so you may just want to copy it to another folder to do this exercise.

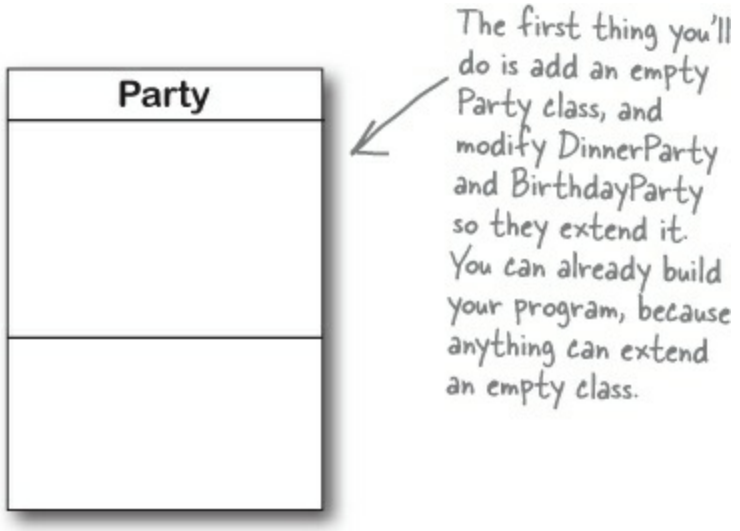
NOTE

Look at the two classes side by side. What methods and properties do they have in common?

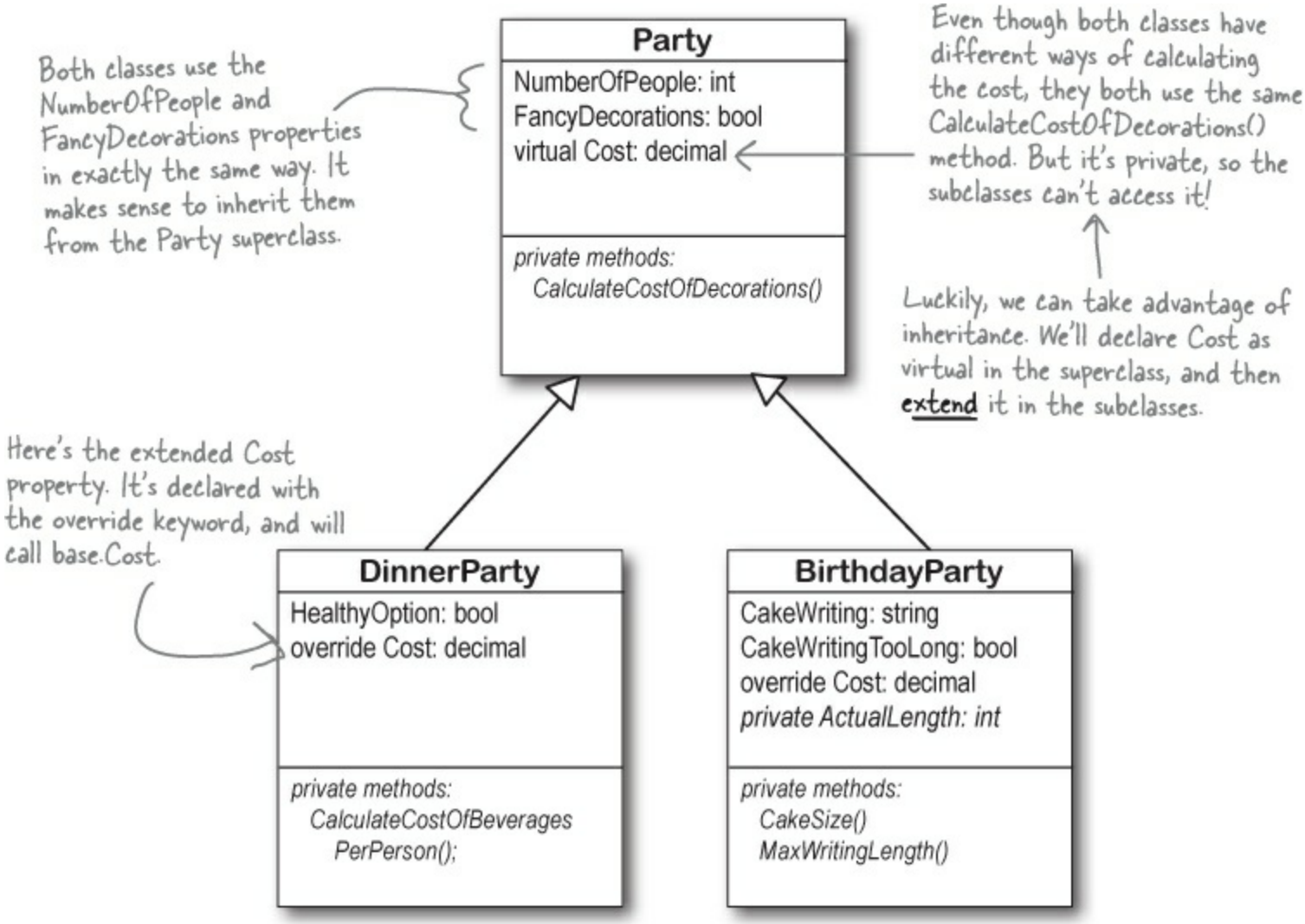
- **❶ THINK ABOUT THE NEW CLASS MODEL.**
  The first step to writing a good program is **thinking about its design**. We'll still have the same `DinnerParty` and `BirthdayParty` classes, but now they'll inherit from a single `Party` class. We need them to have exactly the same properties so we don't have to make any changes to the form.



- **❷ ADD THE PARTY BASE CLASS.**
  Create a **new Windows Forms application**. Add a class called `Party` to the program. Then add the `DinnerParty` and `BirthdayParty` classes from the project at the beginning of this chapter, and update the `DinnerParty` and `BirthdayParty` classes so they extend `Party`.



- **❸ MOVE SHARED BEHAVIOR INTO THE PARTY SUPERCLASS.**
  Cut the `CostOfFoodPerPerson` constant, the `NumberOfPeople` and `FancyDecorations` properties, and the `CalculateCostOfDecorations()` method from either the `DinnerParty` or `BirthdayParty` class (they're identical in both), then paste them into `Party`. Make sure you delete them from both subclasses.
  Create a `Cost` property in `Party` and mark it `virtual`, and mark the `Cost` in the subclasses `override`.



- ❹ The hardest part of this exercise is figuring out what part of the two `Cost` properties in the subclasses should be copied to the `Party` base class. That's because you have a lot of choices. You could just create an automatic `Cost` property in the `Party` class, and keep the `Cost` property in the subclasses the same. But for this exercise, your job is to look at the `Cost` properties in the original `DinnerParty` and `BirthdayParty` classes, figure out what they have in common, and move as many lines as you can into the base class.
  Here's a hint. Both `DinnerParty` and `BirthdayParty` `Cost` properties should start with these lines:

```
override public decimal Cost {
    get {
        decimal totalCost = base.Cost;
```

Don't forget to **add the $100 charge for parties over 12** to the base `Cost` property in `Party`.

EXERCISE SOLUTION

Check it out — you changed the `DinnerParty` and `BirthdayParty` classes so that they inherited from the same base class, `Party`. Then you were able to make the change to the cost calculation to add the $100 fee, and you didn't have to change the form at all. Neat!

```
class Party
{
    public const int CostOfFoodPerPerson = 25;

    public int NumberOfPeople { get; set; }

    public bool FancyDecorations { get; set; }

    private decimal CalculateCostOfDecorations()
    {
        decimal costOfDecorations;
        if (FancyDecorations)
            costOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        else
            costOfDecorations = (NumberOfPeople * 7.50M) + 30M;
        return costOfDecorations;
    }

    virtual public decimal Cost
    {
        get {
            decimal totalCost = CalculateCostOfDecorations();
            totalCost += CostOfFoodPerPerson * NumberOfPeople;

            if (NumberOfPeople > 12)
                totalCost += 100;

            return totalCost;
        }
    }
}

class BirthdayParty : Party
{
    public BirthdayParty(int numberOfPeople,
                         bool fancyDecorations, string cakeWriting)
    {
        NumberOfPeople = numberOfPeople;
        FancyDecorations = fancyDecorations;
        CakeWriting = cakeWriting;
    }
```

These properties and the constant were identical in DinnerParty and BirthdayParty, so they were cut from the subclasses and pasted straight into the superclass.

This method was also identical in both subclasses, so it was moved to the Party base class too.

Don't forget to mark Cost virtual!

These two lines were identical in both original DinnerParty and BirthdayParty classes, so we moved them to the base Cost property. We moved as much behavior as we could into the Party class.

Now that the birthday and dinner parties have their own classes that extend the Party base class, it's easy to add the $100 charge for parties over 12. Just add it to the base class, and the subclasses will inherit the behavior.

BirtdayParty extends Party.

The BirthdayParty constructor stays the same, even though it sets properties that are in the base class.

```
    public string CakeWriting { get; set; }

    private int ActualLength
    {
        get
        {
            if (CakeWriting.Length > MaxWritingLength())
                return MaxWritingLength();
            else
                return CakeWriting.Length;

        }
    }

    private int CakeSize() {
        if (NumberOfPeople <= 4)
            return 8;
        else
            return 16;
    }

    private int MaxWritingLength() {
        if (CakeSize() == 8)
            return 16;
        else
            return 40;
    }


    public bool CakeWritingTooLong {
        get {
            if (CakeWriting.Length > MaxWritingLength())
                return true;
            else
                return false;
        }
    }

    override public decimal Cost {
        get {
            decimal totalCost = base.Cost;
            decimal cakeCost;
            if (CakeSize() == 8)
                cakeCost = 40M + ActualLength * .25M;
            else
                cakeCost = 75M + ActualLength * .25M;
            return totalCost + cakeCost;
        }
    }
}
```

CakeWriting and ActualLength are only used by BirthdayParty but not Party, so they stay in BirthdayParty.

The CakeWriting property, ActualLength property, and the methods that they use stay in the BirthdayParty class. So does the CakeWritingTooLong property.

We moved the first two statements of the Cost property into the base class because they were identical in both DinnerParty and BirthdayParty. The first thing the BirthdayParty's Cost property does is call base.Cost to execute those two statements.

Continues in Note.

Here's the last class in Kathleen's solution. There's no change to the form code at all!

```
class DinnerParty : Party {
    public bool HealthyOption { get; set; }

    public DinnerParty(int numberOfPeople, bool healthyOption,
                       bool fancyDecorations) {
        NumberOfPeople = numberOfPeople;
        FancyDecorations = fancyDecorations;
        HealthyOption = healthyOption;
    }

    private decimal CalculateCostOfBeveragesPerPerson() {
        decimal costOfBeveragesPerPerson;
        if (HealthyOption)
            costOfBeveragesPerPerson = 5.00M;
        else
            costOfBeveragesPerPerson = 20.00M;
        return costOfBeveragesPerPerson;
    }

    override public decimal Cost {
        get {
            decimal totalCost = base.Cost;
            totalCost += CalculateCostOfBeveragesPerPerson() * NumberOfPeople;
            if (HealthyOption)
                totalCost *= .95M;
            return totalCost;
        }
    }
}
```

The HealthyOption property is only used in dinner parties, not birthday parties, so it stays in the class.

The CalculatecostOfBeveragesPerPerson() method and the constructor stay in the DinnerParty class because they're not used by BirthdayParty.

The Cost property works just like in the BirthdayParty class. It uses base.Cost to execute the statements in Party.Cost, and uses the result as a starting point to finish the calculation.

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. It would have been a lot of work to add that $100 charge for parties over 12 to the separate `DinnerParty` and `BirthdayParty` classes. But after you redesigned your program with inheritance, it just took two lines of code. This was easy because you moved *only the behavior that was shared between the Cost properties in the subclasses* into a shared property in the base class.

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 for dinner parties goes in `DinnerParty`, code for birthday parties goes in `BirthdayParty`, and code that's shared between them goes in `Party`.

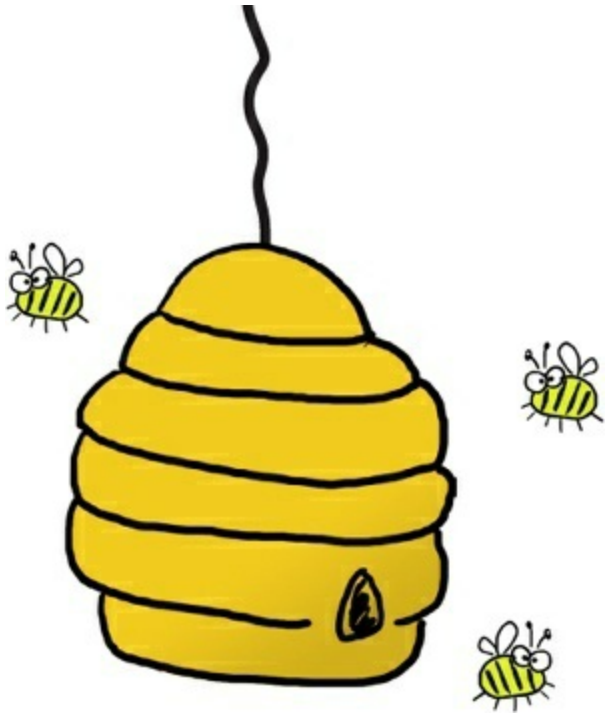
Here's something to think about. We separated the concerns about the user interface into the `Form` object. It doesn't do cost calculations itself — that's encapsulated behind the `Cost` properties of the `DinnerParty` and `BirthdayParty` classes. But we decided that converting the decimal cost to a current string is a concern of the `Form`, not something that the party classes need to be concerned with. Did we make the right call?

NOTE

Remember, any program can be written in many ways, and usually there's no single "right" answer. Not even if it's written in a book!

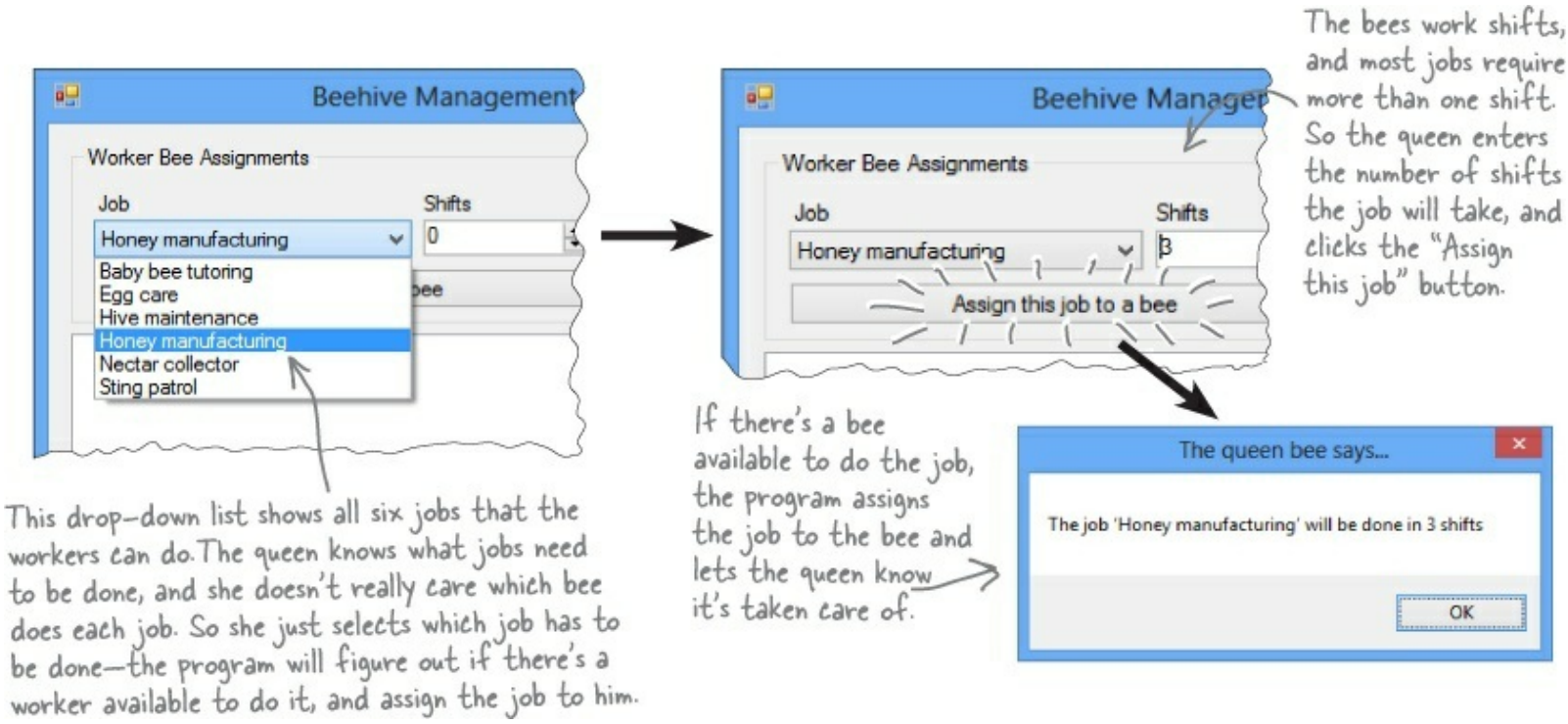
Build a beehive management system

A queen bee needs your help! Her hive is out of control, and she needs a program to help manage it. 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:

- **❶ THE QUEEN ASSIGNS JOBS TO HER WORKERS.**
  There are six possible jobs that the workers can do. Some know how to collect nectar and manufacture honey; others can maintain the hive and patrol for enemies. A few bees can do every job in the hive. So your program will need to give her a way to assign a job to any bee that's available to do it.



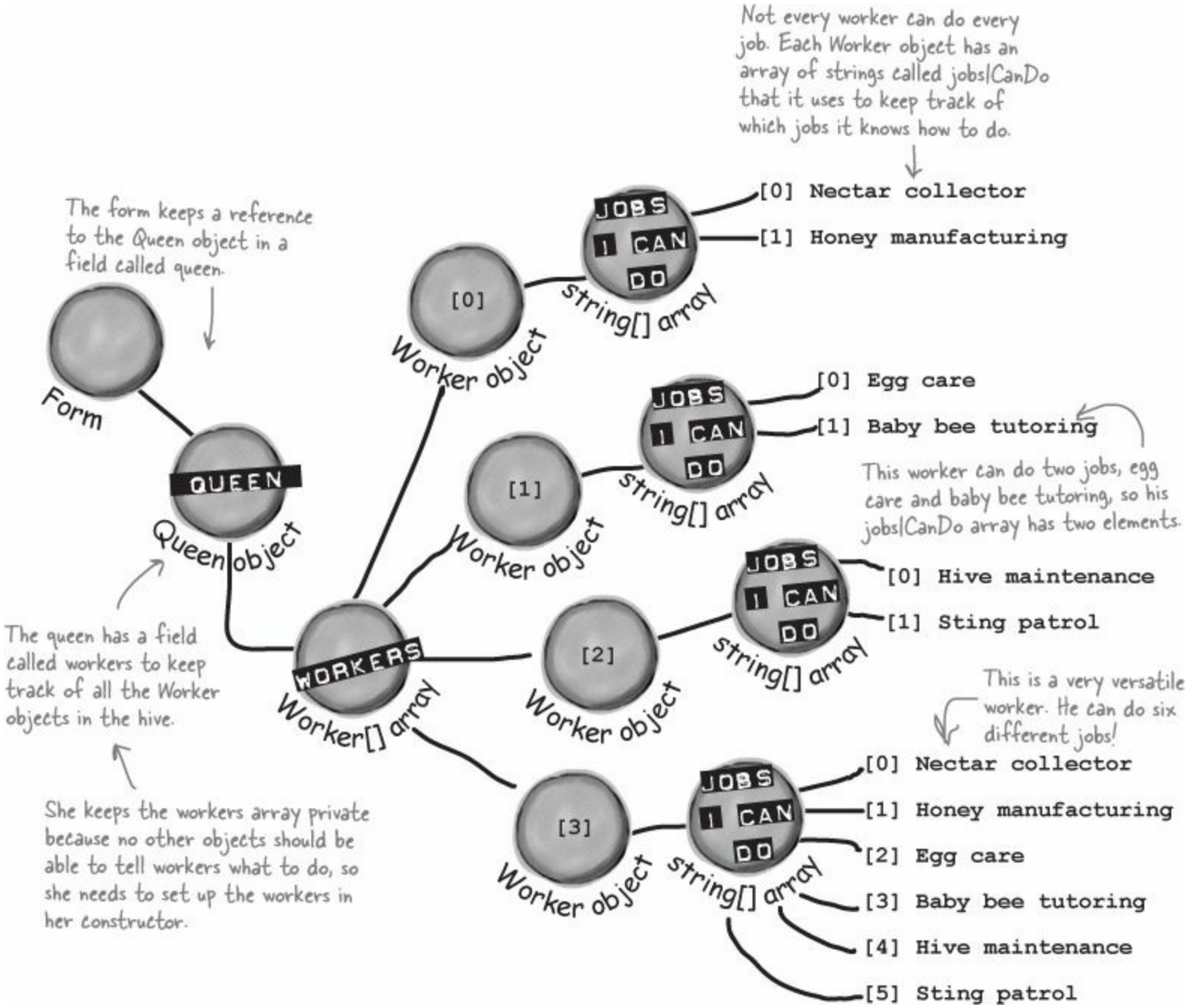
- **❷ WHEN THE JOBS ARE ALL ASSIGNED, IT'S TIME TO WORK.**
  Once the queen's done assigning the work, she'll tell the bees to work the next shift by clicking the "Work the next shift" button. The program then generates a shift report that tells her which bees worked that shift, what jobs they did, and how many more shifts they'll be working each job.

```
Report for shift #1
Worker #1 is doing 'Honey manufacturing' for 2 more shifts
```

How you'll build the beehive management system

This project is divided into two parts. The first part is a bit of a review, where you'll create the basic system to manage the hive. It's got two classes, `Queen` and `Worker`. You'll build the form for the system, and hook it up to the two classes. And you'll **make sure the classes are well-encapsulated** so they don't get in your way when you move on to the second part later.

This is the object model that you'll build. The form has a reference to an instance of Queen, who keeps track of her Worker objects using an array of Worker references.



The form creates the array of workers. Then it creates each worker and adds it to the array.

```
Worker[] workers = new Worker[4];
workers[0] = new Worker(new string[] { "Nectar collector", "Honey manufacturing" });
workers[1] = new Worker(new string[] { "Egg care", "Baby bee tutoring" });
workers[2] = new Worker(new string[] { "Hive maintenance", "Sting patrol" });
workers[3] = new Worker(new string[] { "Nectar collector", "Honey manufacturing",
   "Egg care", "Baby bee tutoring", "Hive maintenance", "Sting patrol" });
queen = new Queen(workers);
```

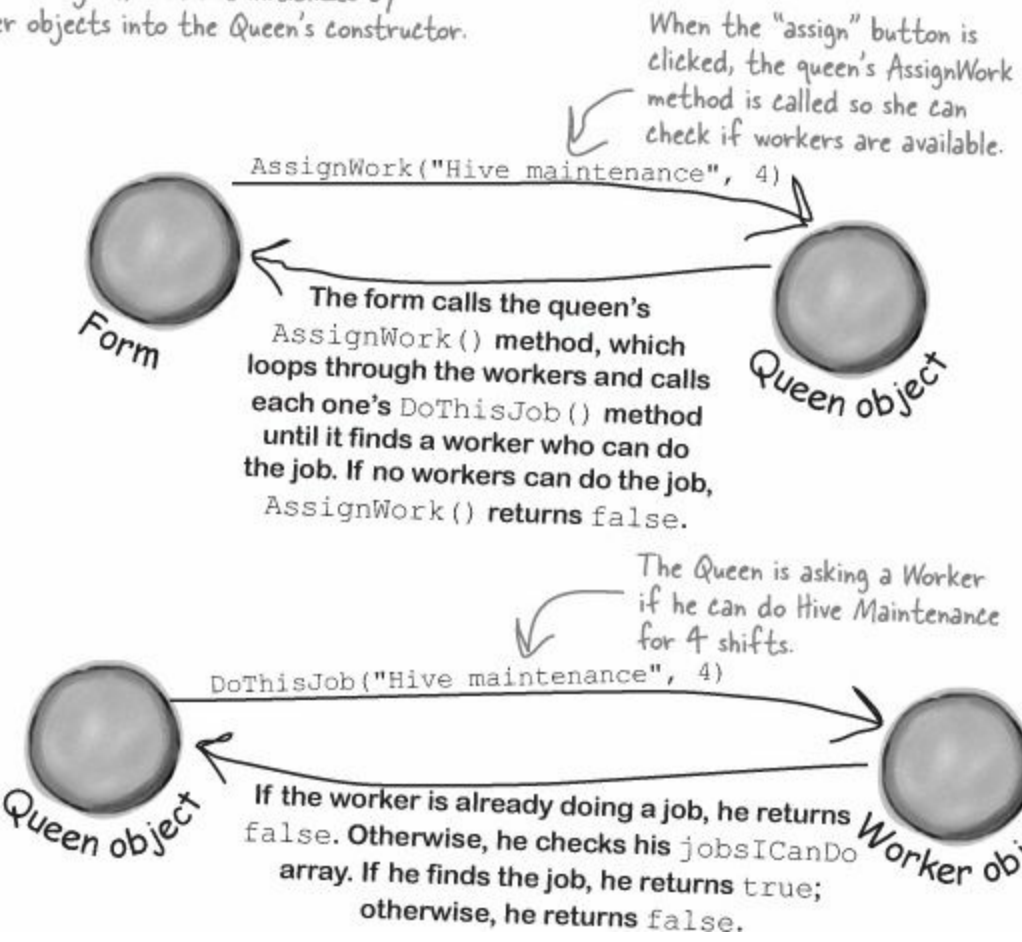
Each Worker object's constructor takes one parameter, an array of strings that tell it what jobs it knows how to do.

The form has a field that points to a Queen object, which it initializes by passing the newly created array of Worker objects into the Queen's constructor.

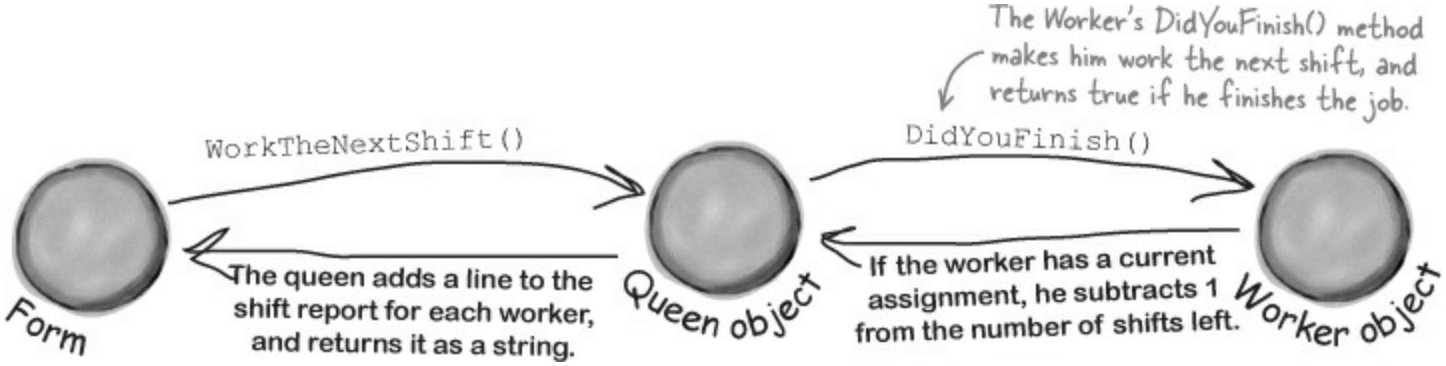
The queen checks each worker to see if he's available to do the job.

The queen's AssignWork() method goes through the array of workers, calling each one's DoThisJob() method until she finds one who can do the job.

The queen can assign work to workers and then tell them to work the next shift.

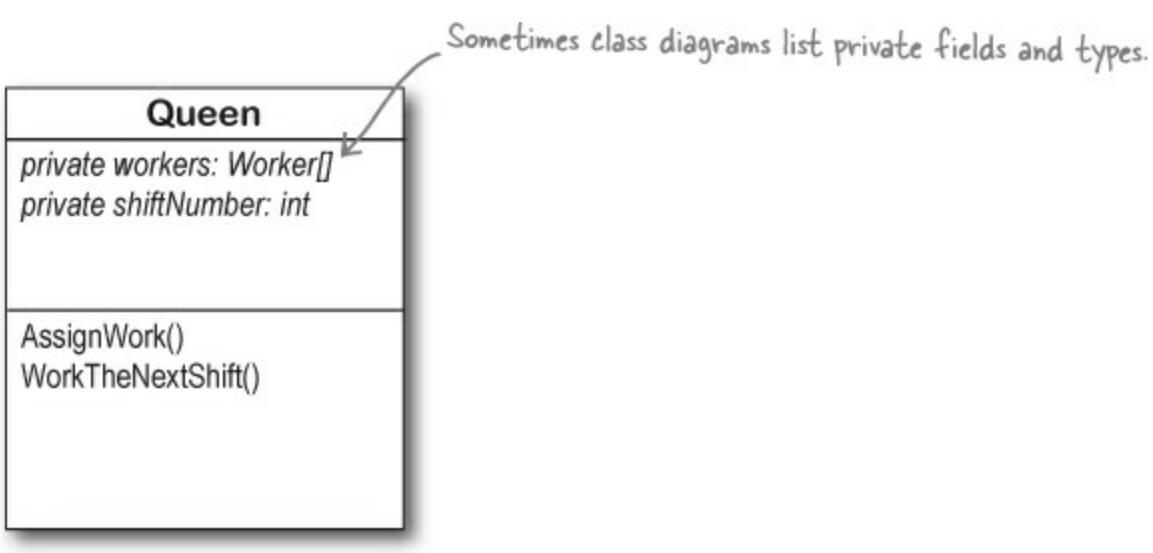


The queen tells each worker to work a shift, then compiles the results into a shift report.



EXERCISE

A queen bee needs your help! Use what you've learned about classes and objects to build a beehive management system to help her track her worker bees. In this first part of the project you'll design the form, add the `Queen` and `Worker` classes, and get the basic system working.

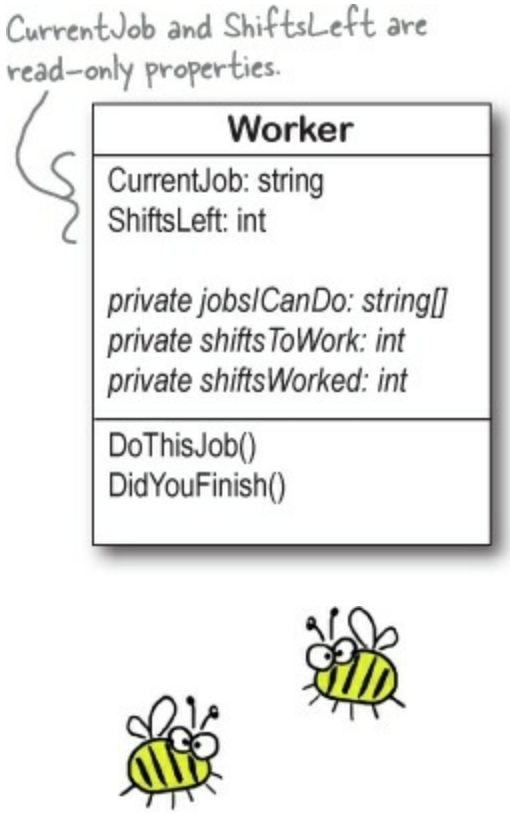


The program has one `Queen` object that manages the work being done.

- The `Queen` uses an array of `Worker` objects to **track each of the worker bees** and whether or not those bees have been assigned jobs. It's stored in a private `Worker[ ]` field called `workers`.
- The form calls the `AssignWork()` method, passing a string for the job that needs to be performed and an `int` for the number of shifts. It'll return `true` **if it finds a worker to assign the job to**, or `false` if it couldn't find a worker to do that job.
- The form's "Work the next shift" button calls `WorkTheNextShift()`, which **tells the workers to work and returns a shift report** to display. It tells each `Worker` object to work one shift, and then checks that worker's status so it can add a line to the shift report.
- Look closely at the **screenshot on the facing page** to see exactly what the `WorkTheNextShift()` method returns. First it creates a string ("Report for shift #13"). Then it uses a `for` loop to execute two `if` statements for each `Worker` in the `workers[]` array. The first `if` statement checks if the worker finsished the job ("Worker #2 finished the job"). The second `if` statement checks if the `Worker` is currently doing a job, and if so, prints how many more shifts he'll be working.

The queen uses an array of `Worker` objects to keep track of all of the workers and what jobs they're doing.

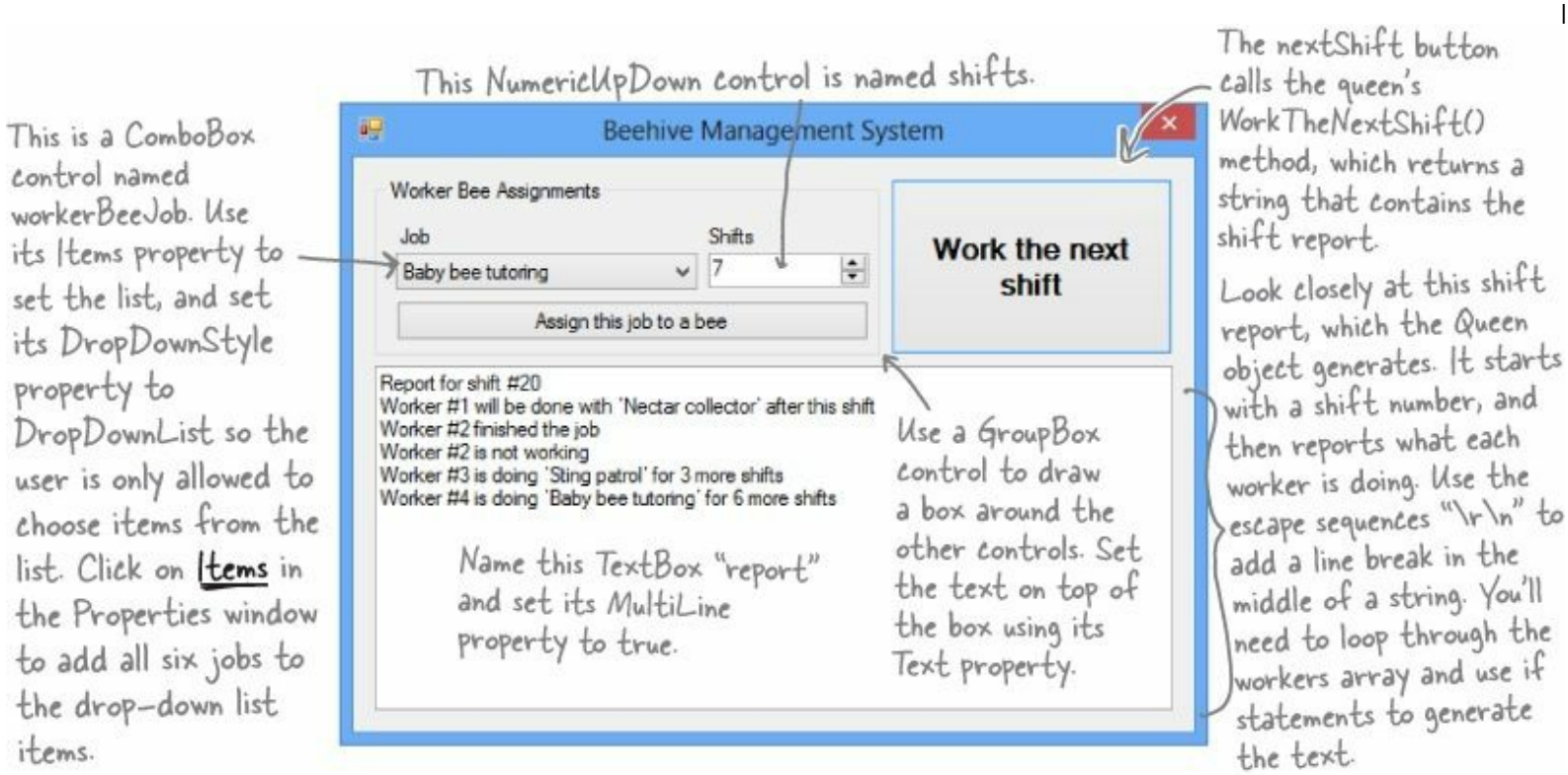
- `CurrentJob` is a read-only property that tells the `Queen` object **what job the worker's doing** ("Sting patrol," "Hive maintenance," etc.). If the worker isn't doing any job, it'll return an empty string.
- The `Queen` object attempts to assign a job to a worker using its `DoThisJob()` method. If that worker is not already doing the job, and if it's a job that he knows how to do, then **he'll accept the assignment** and the method returns `true`. Otherwise, it returns `false`.
- When the `DidYouFinish()` method is called, **the worker works a shift**. He keeps track of how many shifts are left in the current job. If the job is done, then he resets his current job to an empty string so that he can take on his next assignment. The method returns `true` if the worker finished a job this shift; otherwise, it returns `false`.



STRING.ISNULLOREMPTY()

Each bee stores his current job as a string. So a worker can figure out if he's currently doing a job by checking his CurrentJob property — it'll be equal to an empty string if he's waiting for his next job. C# gives you an easy way to do that: String.IsNullOrEmpty(currentJob) will return true if the currentJob string property is either empty or null, and false otherwise.

- **❶ BUILD THE FORM.**
  The form is pretty simple — all of the intelligence is in the `Queen` and `Worker` classes. The form has a private Queen field, and two buttons call its `AssignWork()` and `WorkTheNextShift()` methods. You'll need to add a `ComboBox` control for the bee jobs (flip back to the screenshot to see its list items), a `NumericUpDown` control, two buttons, and a multiline text box for the shift report. You'll also need the form's constructor — it's below the screenshot.




```
public Form1() {
    InitializeComponent();
    workerBeeJob.SelectedIndex = 0;
    Worker[] workers = new Worker[4];
    workers[0] = new Worker(new string[] { "Nectar collector", "Honey manufacturing" });
    workers[1] = new Worker(new string[] { "Egg care", "Baby bee tutoring" });
    workers[2] = new Worker(new string[] { "Hive maintenance", "Sting patrol" });
    workers[3] = new Worker(new string[] { "Nectar collector", "Honey manufacturing",
                 "Egg care", "Baby bee tutoring", "Hive maintenance", "Sting patrol" });
    queen = new Queen(workers);
}
```

Here's the complete constructor for the form. It's got the code from the previous page. It also has this additional line that sets the ComboBox to show its first item (so it's not blank when the form loads).

Your form will need a Queen field called queen. You'll pass that array of Worker object references to the Queen object's constructor.

- **❷ BUILD THE WORKER AND QUEEN CLASSES.**

  You've got almost everything you need to know about the `Worker` and `Queen` classes. There are just a couple more details. `Queen.AssignWork()` loops through the `Queen` object's `workers` array and attempts to assign the job to each `Worker` using its `DoThisJob()` method. The `Worker` object checks its `jobsICanDo` string array to see if it can do the job. If it can, it sets its private `shiftsToWork` field to the job duration, its `CurrentJob` to the job, and its `shiftsWorked` to zero. When it works a shift, it increases `shiftsWorked` by one. The read-only `ShiftsLeft` property returns `shiftsToWork - shiftsWorked` — the queen uses it to see how many shifts are left on the job.

EXERCISE SOLUTION

```
class Worker {
    public Worker(string[] jobsICanDo) {
        this.jobsICanDo = jobsICanDo;
    }

    public int ShiftsLeft {
        get {
            return shiftsToWork - shiftsWorked;
        }
    }

    private string currentJob = "";
    public string CurrentJob {
        get {
            return currentJob;
        }
    }

    private string[] jobsICanDo;
    private int shiftsToWork;
    private int shiftsWorked;

    public bool DoThisJob(string job, int numberOfShifts) {
        if (!String.IsNullOrEmpty(currentJob))
            return false;
        for (int i = 0; i < jobsICanDo.Length; i++)
            if (jobsICanDo[i] == job) {
                currentJob = job;
                this.shiftsToWork = numberOfShifts;
                shiftsWorked = 0;
                return true;
            }
        return false;
    }

    public bool DidYouFinish() {
        if (String.IsNullOrEmpty(currentJob))
            return false;
        shiftsWorked++;
        if (shiftsWorked > shiftsToWork) {
            shiftsWorked = 0;
            shiftsToWork = 0;
            currentJob = "";
            return true;
        }
        else
            return false;
    }
```

The constructor just sets the jobsICanDo field, which is a string array. It's private because we want the queen to ask the worker to do a job, rather than make her check whether he knows how to do it.

ShiftsLeft is a read-only property that calculates how many shifts are left on the current job.

CurrentJob is a read-only property that tells the queen which job needs to be done.

The queen uses the worker's DoThisJob() method to assign work to him—he checks his jobsICanDo field to see if he knows how to do the job.

We used ! —the NOT operator—to check if the string is NOT null or empty. It's just like checking to see if something's false.

The queen uses the worker's DidYouFinish() method to tell him to work the next shift. The method only returns true if this is the **very last shift** that he's doing the job. That way, the queen can add a line to the report that the bee will be done after this shift.

Take a close look at the logic here. First it checks the currentJob field: if the worker's not working on a job, it just returns false, which stops the method. If not, then it increments ShiftsWorked, and then checks to see if the job's done by comparing it with ShiftsToWork. If it is, the method returns true. Otherwise, it returns false.

```
class Queen {
    public Queen(Worker[] workers) {
        this.workers = workers;
    }

    private Worker[] workers;
    private int shiftNumber = 0;

    public bool AssignWork(string job, int numberOfShifts) {
        for (int i = 0; i < workers.Length; i++)
            if (workers[i].DoThisJob(job, numberOfShifts))
                return true;
        return false;
    }

    public string WorkTheNextShift() {
        shiftNumber++;
        string report = "Report for shift #" + shiftNumber + "\r\n";
        for (int i = 0; i < workers.Length; i++)
        {
            if (workers[i].DidYouFinish())
                report += "Worker #" + (i + 1) + " finished the job\r\n";
            if (String.IsNullOrEmpty(workers[i].CurrentJob))
                report += "Worker #" + (i + 1) + " is not working\r\n";
            else
              if (workers[i].ShiftsLeft > 0)
                report += "Worker #" + (i + 1) + " is doing '" + workers[i].CurrentJob
                    + "' for " + workers[i].ShiftsLeft + " more shifts\r\n";
              else
                report += "Worker #" + (i + 1) + " will be done with '"
                    + workers[i].CurrentJob + "' after this shift\r\n";
        }
        return report;
    }
}
```

The queen keeps her array of workers private because once they're assigned, no other class should be able to change them...or even see them, since she's the only one who gives them orders. The constructor sets the field's value.

When she assigns work to her worker bees, she starts with the first one and tries assigning him the job. If he can't do it, she moves on to the next. When a bee who can do the job is found, the method returns (which stops the loop).

The queen's WorkTheNextShift() method tells each worker to work a shift and adds a line to the report depending on the worker's status.

We already gave you the constructor. Here's the rest of the code for the form:

> **NOTE**
>
> The form uses its queen field to keep a reference to the Queen object, which in turn has an array of references to the worker objects.

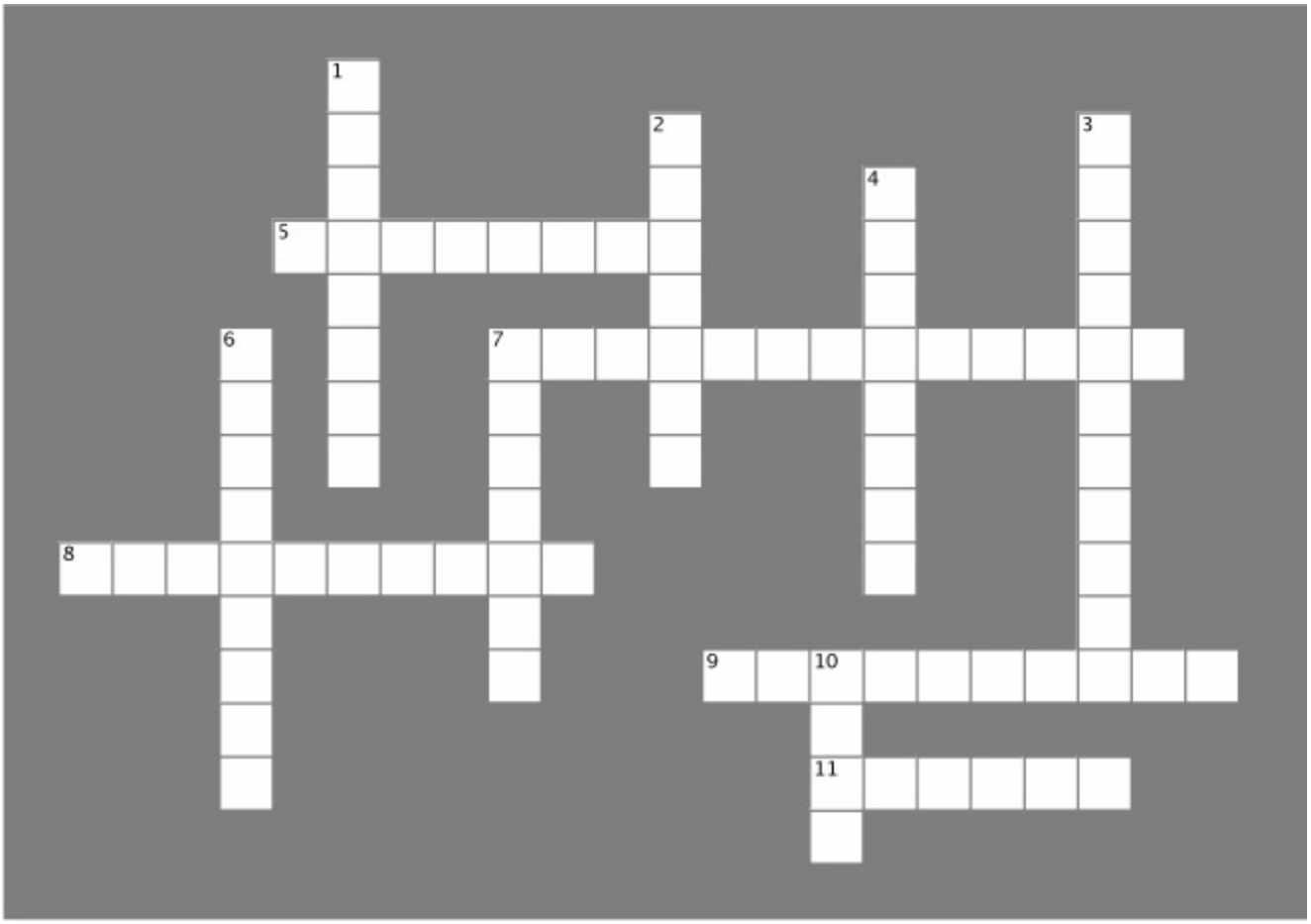
```
private Queen queen;

private void assignJob_Click(object sender, EventArgs e) {
    if (queen.AssignWork(workerBeeJob.Text, (int)shifts.Value) == false)
        MessageBox.Show("No workers are available to do the job '"
            + workerBeeJob.Text + "'", "The queen bee says...");
    else
        MessageBox.Show("The job '" + workerBeeJob.Text + "' will be done in "
            + shifts.Value + " shifts", "The queen bee says...");
}

private void nextShift_Click(object sender, EventArgs e) {
    report.Text = queen.WorkTheNextShift();
}
```

The assignJob button calls the queen's AssignWork() method to assign work to a worker, and displays a message box, depending on whether or not a worker's available to do the job.

The nextShift button tells the queen to work the next shift. She generates a report, which it displays in the report text box.

INHERITANCECROSS

Before you move on to the next part of the exercise, give your brain a break with a quick crossword.



Across	Down
5. This method gets the value of a property. 7. This method returns true if you pass it “”. 8. The constructor in a subclass doesn’t need the same _____ as the constructor in its base class. 9. A control on a form that lets you create tabbed applications. 11. This type of class can’t be instantiated.	1. A _____ can override methods from its base class. 2. If you want a subclass to override a method, mark the method with this keyword in the base class. 3. A method in a class that’s run as soon as it’s instantiated. 4. What a subclass does to replace a method in the base class. 6. This contains base classes and subclasses. 7. What you’re doing by adding a colon to a class declaration. 10. A subclass uses this keyword to call the members of the class it inherited from.

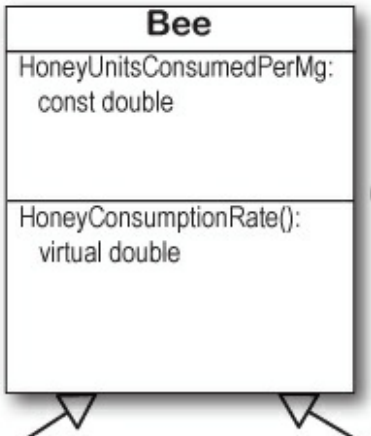
Answers in Inheritancecross Solution.

Use inheritance to extend the bee management system

Now that you have the basic system in place, use inheritance to let it track how much honey each bee consumes. Different bees consume different amounts of honey, and the queen consumes the most honey of all. You'll use what you've learned about inheritance to create a `Bee` base class that `Queen` and `Worker` inherit from.

You'll add a Bee class that both Queen and Worker extend. The Bee class will have the basic members for calculating honey consumption.

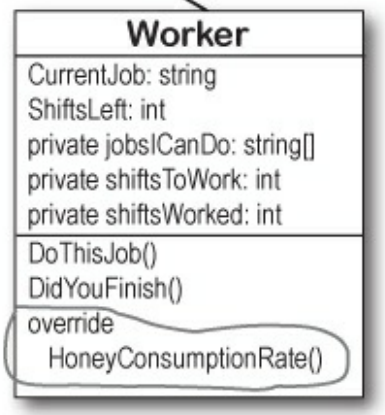
The HoneyConsumptionRate() method calculates how much honey the bee consumes during a shift.

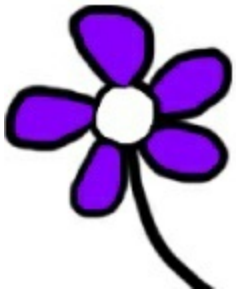


Sometimes we'll show you return values and private members in class diagrams.

Worker bees use more honey with each consecutive shift worked on a job. The Worker class will extend Bee and override HoneyConsumptionRate().

The queen will need to extend Bee and call the HoneyConsumptionRate() method to add information about honey consumed per shift to the report.





ADD EXISTING ITEM

Whenever you have a two-part exercise, it's always a good idea to start a new project for the second part. That way, you can always get back to the first solution if you need it. An easy way to do that is to right-click on the project name in the new project's Solution Explorer in the IDE, select Add Existing Item from the menu, navigate to the old project's folder, and select the files you want to add. The IDE will make new copies of those files in the new project's folder, and add them to the project. There are a few things to watch out for, though. The IDE will NOT change the namespace, so you'll need to edit each class file and change its namespace line by hand. And if you add a form, make sure to add its designer (.Designer.cs) and resource (.resx) files — and make sure you change their namespaces, too.

EXERCISE

We're not done yet! The queen got a call from her accountants, who told her she needs to keep track of how much honey the hive is spending on its workers. Here's a perfect chance to use your new inheritance skills! Add a new `Bee` superclass and use it to calculate honey consumption for each shift.

- **❶ CREATE THE BEE CLASS AND MODIFY QUEEN AND WORKER TO EXTEND IT.**
  The Bee class has a `HoneyConsumptionRate()` method that calculates how much honey the bee uses per shift. Your job will be to modify the `Worker` and `Queen` classes to extend it.

```
class Bee {
    public const double HoneyUnitsConsumedPerMg = .25;

    public double WeightMg { get; private set; }

    public Bee(double weightMg) {
        WeightMg = weightMg;
    }

    virtual public double HoneyConsumptionRate() {
        return WeightMg * HoneyUnitsConsumedPerMg;
    }
}
```

The Bee constructor takes one parameter, the weight of the bee in milligrams, which is used in the base honey consumption calculation.

- **❷ MODIFY THE QUEEN AND WORKER CLASSES TO EXTEND BEE.**
  The `Queen` and `Worker` classes will inherit the basic honey consumption behavior from their new parent `Bee` superclass. You'll need to set up their constructors to call the base class constructor.

  Modify the `Queen` class to extend `Bee`. You'll need to add a `double` parameter called `weightMg` to the constructor that gets passed back to the base constructor.
  Modify the `Worker` class to extend `Bee`, too — you'll need to make the same modification to the `Worker` constructor that you did for the `Queen`.

NOTE

— Hint: you can use the "does not contain a constructor" error message you saw earlier in the chapter to your advantage! Have the Worker class inherit from Bee, then build your project. When the IDE displays the error, double-click on it and the IDE will jump right to the Worker constructor automatically. How convenient!

- **❸ MODIFY THE FORM TO INITIALIZE THE QUEEN AND WORKERS WITH THEIR WEIGHTS.**
  Since you changed the `Queen` and `Worker` constructors, you'll also need to **change the form's constructor** so that when it creates its new `Worker` and new `Queen` instances, it passes the additional weights into their constructors. Worker #1 weighs 175mg, worker #2 weighs 114mg, worker #3 weighs 149mg, worker #4 weighs 155mg, and the queen weighs 275mg.
  (Your code should now compile.)

- **❹ OVERRIDE THE WORKER'S HONEYCONSUMPTIONRATE() METHOD**

The `Queen` consumes honey just like the base `Bee` class. And workers consume the same amount of honey...but only while they're idle! When they're working a shift, they consume .65 additional units for each shift they worked so far.
This means that the `Queen` can use the base `HoneyConsumptionRate()` method that she inherits from her `Bee` superclass, but the `Worker` will need to override the method to add the additional .65 units per shift worked. You can also add a constant called `honeyUnitsPerShiftWorked` to make it really clear exactly what this method is doing.
You can use the IDE to get started. Go to the `Worker` class and type "public override" — when you add the space, the IDE will automatically list all the methods you can override:

Choose the `HoneyConsumptionRate()` method from the IntelliSense window. When you do, the IDE will generate a method stub that just calls the base method. Modify your new method so that it starts with the output of `base.HoneyConsumptionRate()` and then adds the extra .65 units consumed per shift worked.

- **❺ ADD HONEY CONSUMPTION TO THE SHIFT REPORT.**
You'll need to modify the `Queen`'s `WorkTheNextShift()` method to keep track of the honey consumed by the `Queen` object and each of the `Worker` objects, calling each object's `HoneyConsumptionRate()` method and adding it to a total. Then it should add this line to the end of the report (replacing `XXX` with the number of units of honey consumed):

NOTE

You should be able to do this by adding just three lines of code to the WorkTheNextShift() method.

```
Total honey consumed for the shift: XXX units
```

BRAIN POWER

Since all bees have a `HoneyConsumptionRate()` method, and the `Queen` and `Worker` are both Bees, shouldn't there be a single, consistent way to call that method for any `Bee` object, no matter what kind of `Bee` it is?

EXERCISE SOLUTION

The constructor gets a new parameter, which it passes back to the base constructor. This lets the form initialize the object with the bee's weight.

```
class Worker : Bee
{
    public Worker(string[] jobsICanDo, double weightMg)
        : base(weightMg)
    {
        this.jobsICanDo = jobsICanDo;
    }

    const double honeyUnitsPerShiftWorked = .65;

    public override double HoneyConsumptionRate()
    {
        double consumption = base.HoneyConsumptionRate();
        consumption += shiftsWorked * honeyUnitsPerShiftWorked;
        return consumption;
    }

    // The rest of the Worker class is the same
    // ...

}
```

The Worker class overrides the HoneyConsumptionRate() method to add the additional honey consumption for bees currently doing a job.

Only the form constructor changed—the rest of the form is exactly the same.

```
public Form1()
{
    InitializeComponent();
    workerBeeJob.SelectedIndex = 0;
    Worker[] workers = new Worker[4];
    workers[0] = new Worker(new string[] { "Nectar collector", "Honey manufacturing" }, 175);
    workers[1] = new Worker(new string[] { "Egg care", "Baby bee tutoring" }, 114);
    workers[2] = new Worker(new string[] { "Hive maintenance", "Sting patrol" }, 149);
    workers[3] = new Worker(new string[] { "Nectar collector", "Honey manufacturing",
        "Egg care", "Baby bee tutoring", "Hive maintenance", "Sting patrol" }, 155);
    queen = new Queen(workers, 275);
}
```

The only change to the form is that the weights need to be added to the Worker and Queen constructors.

Inheritance made it less work for you to update your code and add the new honey consumption behavior to the Queen and Worker classes. It would have been a lot harder to make this change if you'd had a lot of duplicated code.

```
class Queen : Bee
{
    public Queen(Worker[] workers, double weightMg)
        : base(weightMg)
    {
        this.workers = workers;
    }

    private Worker[] workers;
    private int shiftNumber = 0;

    public bool AssignWork(string job, int numberOfShifts)
    {
        for (int i = 0; i < workers.Length; i++)
            if (workers[i].DoThisJob(job, numberOfShifts))
                return true;
        return false;
    }

    public string WorkTheNextShift()
    {
        double honeyConsumed = HoneyConsumptionRate();

        shiftNumber++;
        string report = "Report for shift #" + shiftNumber + "\r\n";
        for (int i = 0; i < workers.Length; i++)
        {
            honeyConsumed += workers[i].HoneyConsumptionRate();

            if (workers[i].DidYouFinish())
                report += "Worker #" + (i + 1) + " finished the job\r\n";
            if (String.IsNullOrEmpty(workers[i].CurrentJob))
                report += "Worker #" + (i + 1) + " is not working\r\n";
            else
                if (workers[i].ShiftsLeft > 0)
                    report += "Worker #" + (i + 1) + " is doing '" + workers[i].CurrentJob
                        + "' for " + workers[i].ShiftsLeft + " more shifts\r\n";
                else
                    report += "Worker #" + (i + 1) + " will be done with '"
                        + workers[i].CurrentJob + "' after this shift\r\n";
        }

        report += "Total honey consumed for the shift: " + honeyConsumed + " units\r\n";

        return report;
    }
}
```

The Queen's constructor gets the same modification as the Worker's does.

This code is the same as before.

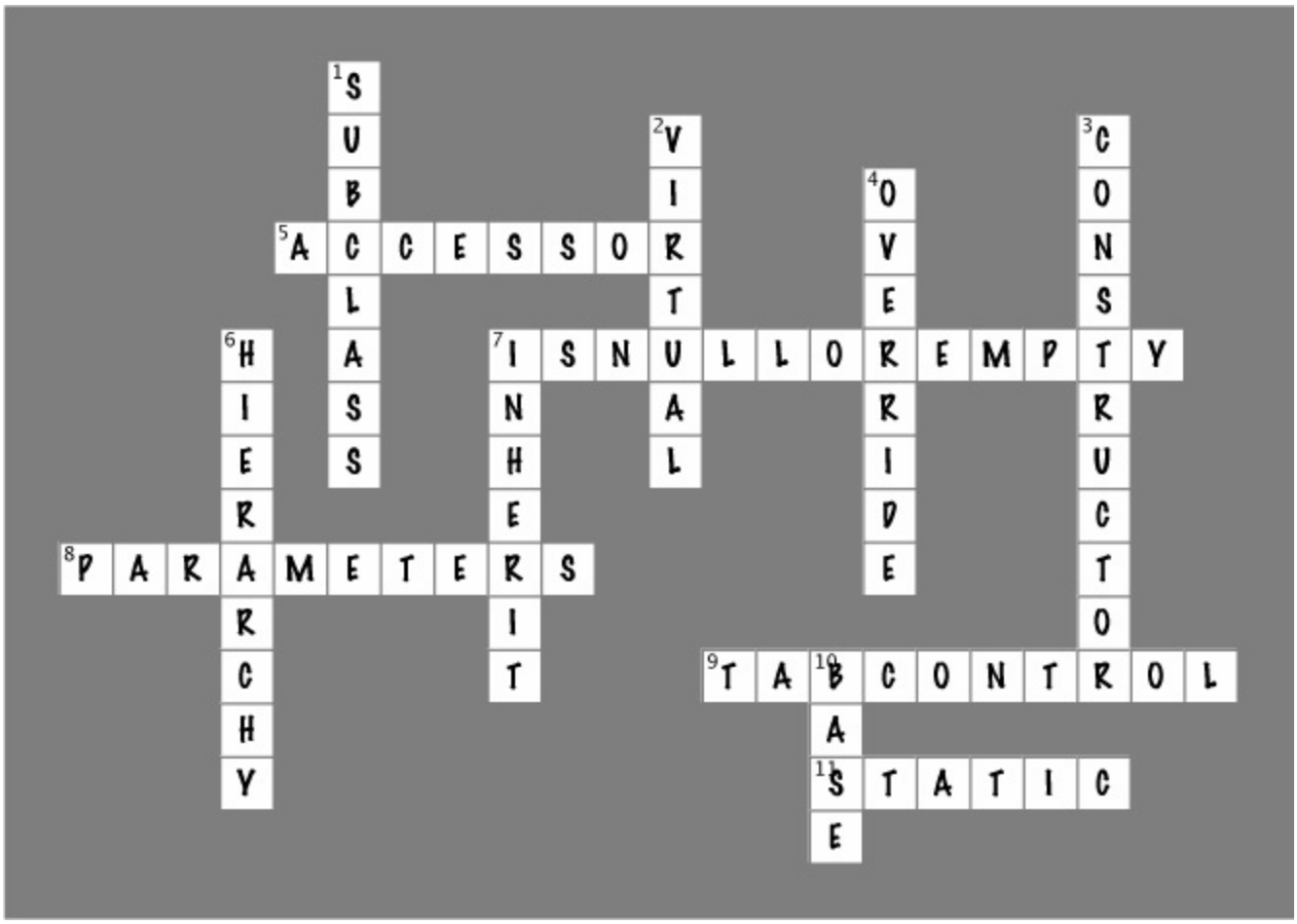
The shift honey calculation needs to start with the Queen's current honey consumption.

As the method loops through each worker, it adds that worker's consumption to the total.

This code also stays exactly the same.

After each worker's line is added to the report, the queen just needs to add one last line with the total honey consumed for the shift.

INHERITANCECROSS SOLUTION



Chapter 7. Interfaces and Abstract Classes: 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?

Let's get back to bee-sics

The General Bee-namics corporation wants to make the Beehive Management System you created in the last chapter into a full-blown Hive Simulator. Here's an overview of the specification for the new version of the program:

General Bee-namics Hive Simulator

To better represent life in the hive, we'll need to add specialized capabilities to the worker bees.

- All bees consume honey and have a weight.
- Queens assign work, monitor shift reports, and tell workers to work the next shift.
- All worker bees work shifts.
- Sting patrol bees will need to be able to sharpen their stingers, look for enemies, and sting them.
- Nectar collector bees are responsible for finding flowers, gathering nectar, and then returning to the hive.

The Bee and Worker classes don't look like they'll change much. We can extend the classes we already have to handle these new features.

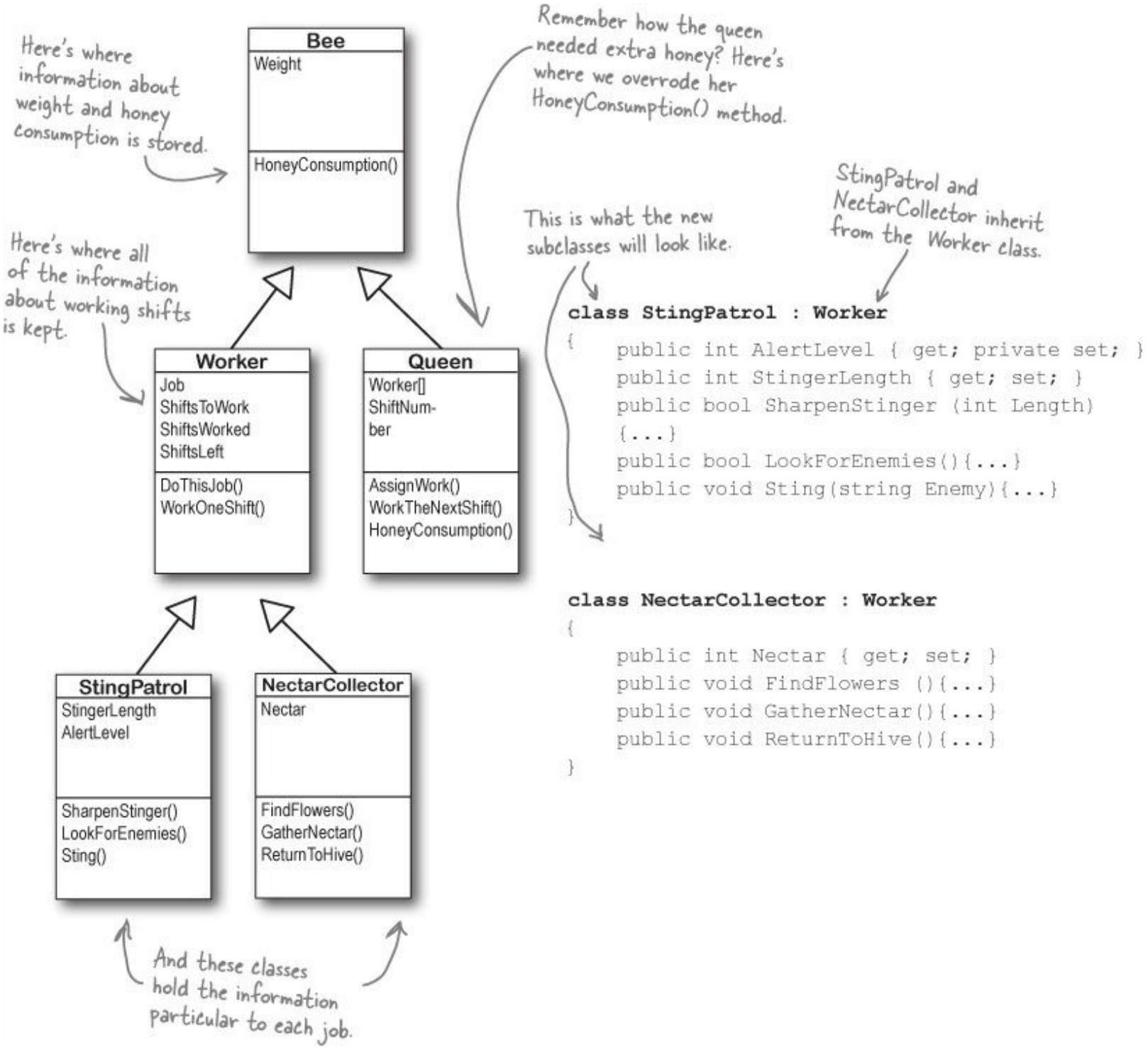
Looks like we'll need to be able to store different data for the worker bees depending on the job they do.

Lots of things are still the same

The bees in the new Hive Simulator will still consume honey in the same way they did before. The queen still needs to be able to assign work to the workers and see the shift reports that tell who's doing what. The workers work shifts just like they did before, too; it's just that the jobs they are doing have been elaborated a little bit.

We can use inheritance to create classes for different types of bees

Here's a class hierarchy with `Worker` and `Queen` classes that inherit from `Bee`, and `Worker` has subclasses `NectarCollector` and `StingPatrol`.




```
class StingPatrol : Worker
{
    public int AlertLevel { get; private set; }
    public int StingerLength { get; set; }
    public bool SharpenStinger (int Length)
    {...}
    public bool LookForEnemies(){...}
    public void Sting(string Enemy){...}
}
```

```
class NectarCollector : Worker
{
    public int Nectar { get; set; }
    public void FindFlowers (){...}
    public void GatherNectar(){...}
    public void ReturnToHive(){...}
}
```

BRAIN POWER

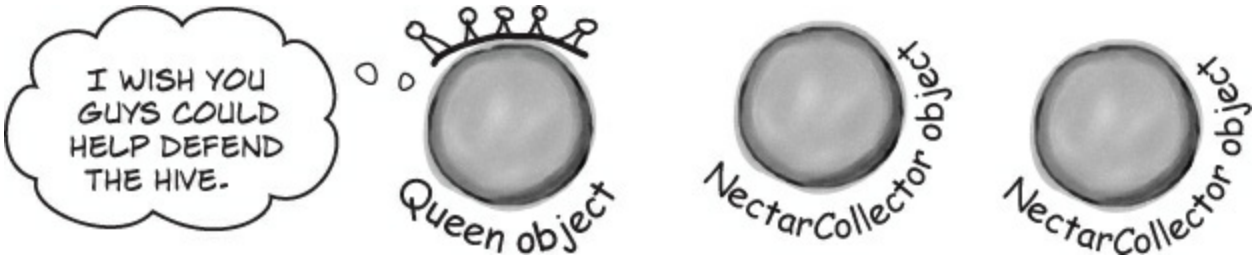
What happens if you have a bee that needs to sting ***and*** collect nectar?

An interface tells a class that it must implement certain methods and properties

A class can only inherit from one other class. So creating two separate subclasses for the `StingPatrol` and `NectarCollector` bees won't help us if we have a bee that can do **both** jobs.

The queen's `DefendTheHive()` method can only tell `StingPatrol` objects to keep the hive safe. She'd love to train the other bees to use their stingers, but she doesn't have any way to command them to attack:

```
class Queen {
  private void DefendTheHive(StingPatrol patroller) { ... }
}
```



You use an interface to require a class to include all of the methods and properties listed inside the interface — if it doesn't, the compiler will throw an error.

There are `NectarCollector` objects that know how to collect nectar from flowers, and instances of `StingPatrol` that can sharpen their stingers and patrol for enemies. But even if the queen could teach the `NectarCollector` to defend the hive by adding methods like `SharpenStinger()` and `LookForEnemies()` to its class definition, she still couldn't pass it into her `DefendTheHive()` method. She could use two different methods:

```
private void DefendTheHive(StingPatrol patroller);
private void AlternateDefendTheHive(NectarCollector patroller);
```

Even if the queen adds sting patrol methods to a NectarCollector object, she still can't pass it to her DefendTheHive() method because it expects a StingPatrol reference. She can't just set a StingPatrol reference equal to a NectarCollector object.

She could add a second method called AlternateDefendTheHive() that takes a NectarCollector reference instead, but that would be cumbersome and difficult to work with.

Plus, the DefendTheHive() and AlternateDefendTheHive() methods would be identical except for the type of the parameter. If she wanted to teach the BabyBeeCare or Maintenance objects to defend the hive, she'd need to keep adding new methods. What a mess!

But that's not a particularly good solution. Both of those methods would be identical, because they'd call the same methods in the objects passed to them. The only difference is that one method would take a `StingPatrol`, and the other would take a `NectarCollector` that happens to have the methods

necessary for patrolling the hive. And you already know how painful it is to maintain two identical methods.

Luckily, C# gives us **interfaces** to handle situations like that. Interfaces let you define a bunch of methods that a class ***must*** have.

An interface ***requires*** that a class has certain methods, and the way that it does that is by ***making the compiler throw errors*** if it doesn't find all the methods required by the interface in every class that implements it. Those methods can be coded directly in the class, or they can be inherited from a base class. The interface doesn't care how the methods or properties get there, as long as they're there when the code is compiled.

Use the interface keyword to define an interface

Adding an interface to your program is a lot like adding a class, except you never write any methods. You just define the methods' return type and parameters, but instead of a block of statements inside curly brackets, you just end the line with a semicolon.

Interfaces do not store data, so you **can't add any fields**. But you *can* add definitions for properties. The reason is that get and set accessors are just methods, and interfaces are all about forcing classes to have certain methods with specific names, types, and parameters. So if you've got a problem that looks like it could be solved by adding a field to an interface, try **using a property instead** — odds are, it'll do what you're looking for.

You declare an interface like this:

Interfaces don't store data. So they don't have fields...but they can have properties.

```
interface IStingPatrol
{
    int AlertLevel { get;}
    int StingerLength { get; set;}
    bool LookForEnemies();
    int SharpenStinger(int length);
}
```

Any class that implements this method must have all of these methods and properties, or the program won't compile.

Any class that implements this interface will need a SharpenStinger() method that takes an int parameter.

```
interface INectarCollector
{
    void FindFlowers();
    void GatherNectar();
    void ReturnToHive();
}
```

You don't write the code for the methods in the interface, just their names. You write the code in the class that implements it.

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.

So how does this help the queen? Now she can make one single method that takes any object that knows how to defend the hive:

```
private void DefendTheHive(IStingPatrol patroller)
```

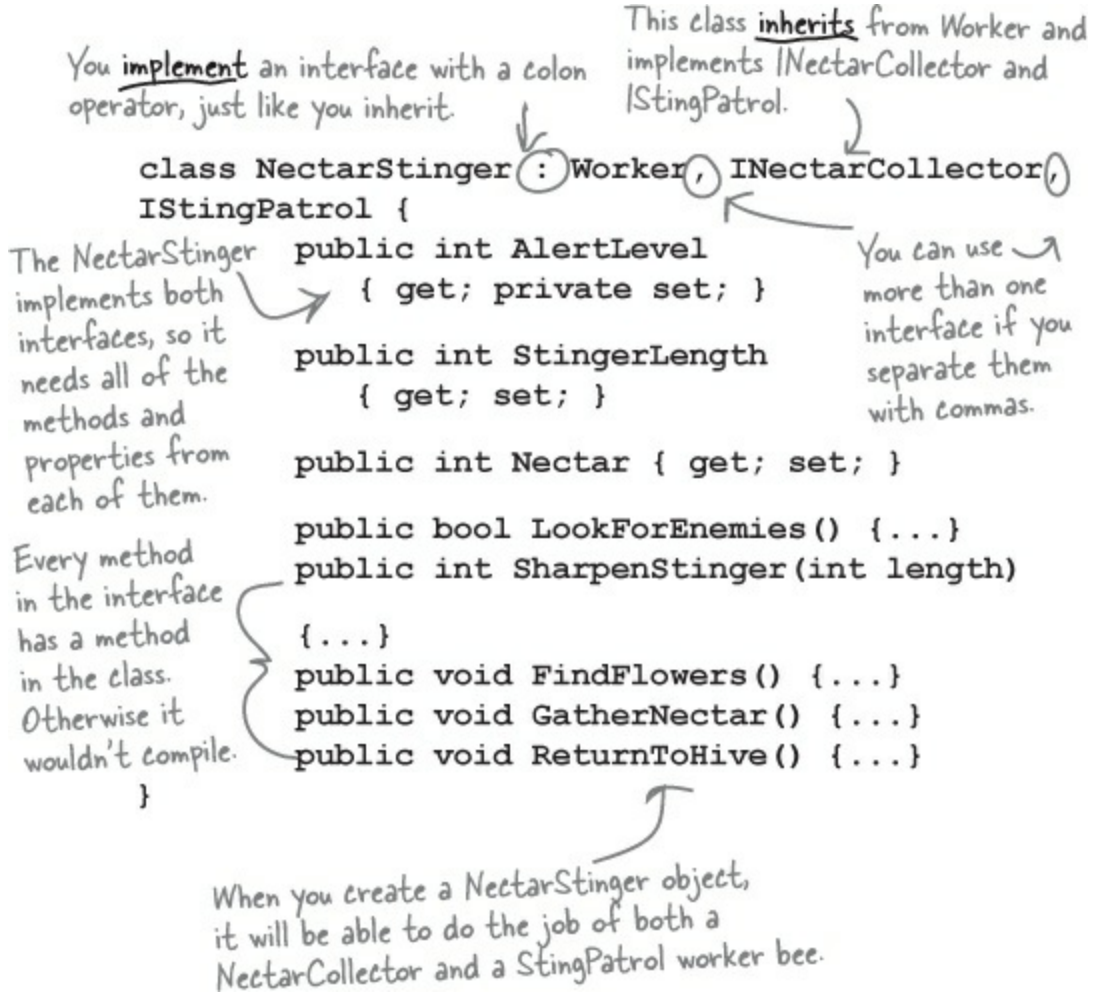
Since this takes an IStingPatrol reference, you can pass it ANY object that implements IStingPatrol.

This gives the queen a single method that can take a `StingPatrol`, `NectarCollector`, and any other bee that knows how to defend the hive — it doesn't matter which class she passes to the method. As long as it implements `IStingPatrol`, the `DefendTheHive()` method is guaranteed that the object has the methods and properties it needs to defend the hive.

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.

Now you can create an instance of NectarStinger that does both jobs

You use the **colon operator** to **implement** an interface, just like you do for inheritance. It works like this: the first thing after the colon is the class it inherits from, followed by a list of interfaces — unless it doesn't inherit from a class, in which case it's just a list of interfaces (in no particular order).



When you've got a class that implements an interface, it acts just like any other class. You can instantiate it with `new` and use its methods:

```
NectarStinger bobTheBee = new NectarStinger();
bobTheBee.LookForEnemies();
bobTheBee.FindFlowers();
```

This is one of the tougher concepts to get into your brain. If it's not quite clear yet, keep reading. We'll have lots of examples throughout the chapter.

THERE ARE NO DUMB QUESTIONS

Q: **Q: I still don't quite get how interfaces improve the beehive code. You'll still need to add a `NectarStinger` class, and it'll still have duplicate code...right?**

A: **A:** Interfaces aren't about preventing you from duplicating code. They're about letting you use one class in more than one situation. The goal is to create one worker bee class that can do two different jobs. You'll still need to create classes for them — that's not the point. The point of the interfaces is that now you've got a way to have a class that does any number of jobs. Say the Queen has a `PatrolTheHive()` method that takes a `StingPatrol` object and a `CollectNectar()` method that takes a `NectarCollector` object. But you don't want `StingPatrol` to inherit from `NectarCollector` or vice versa — each class has public methods and properties that the other one shouldn't have. Now take a minute and try to think of a way to create one single class whose instances could be passed to both methods. Seriously, put the book down, take a minute and try to think up a way! How do you do it?

Interfaces fix that problem. Now you can create an `IStingPatrol` reference — and it can point to any object that implements `IStingPatrol`, no matter what the actual class is. It can point to a `StingPatrol`, or a `NectarStinger`, or even a totally unrelated object. If you've got an `IStingPatrol` reference pointing to an object, then you know you can use all of the methods and properties that are part of the `IStingPatrol` interface, regardless of the actual type of the object.

But the interface is only part of the solution. You'll still need to create a new class that implements it, since it doesn't actually come with any code. Interfaces aren't about avoiding the creation of extra classes or avoiding duplicate code. They're about making one class that can do more than one job without relying on inheritance, as inheritance brings a lot of extra baggage — you'll have to inherit every method, property, and field, not just those that have to do with the specific job.

Can you think of ways that you could still avoid duplicating code while using an interface? You could create a separate class called `Stinger` or `Proboscis` to contain the code that's specific to stinging or collecting nectar. `NectarStinger` and `NectarCollector` could both create a private instance of `Proboscis`, and any time they needed to collect nectar, they'd call its methods and set its properties.

Classes that implement interfaces have to include ALL of the interface's methods

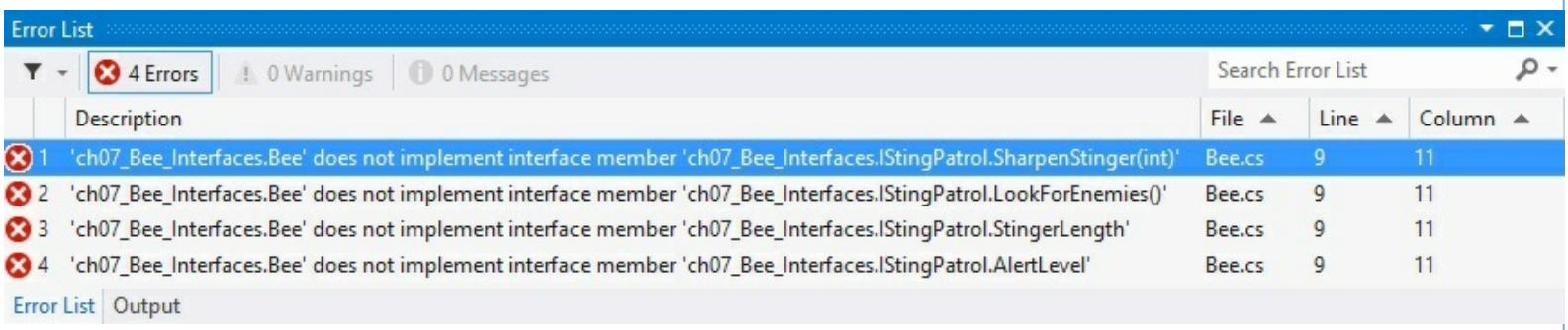
Implementing an interface means that you have to have a method in the class for each and every property and method that's declared in the interface — if it doesn't have every one of them, it won't compile. If a class implements more than one interface, then it needs to include all of the properties and methods in each of the interfaces it implements. But don't take our word for it...

DO THIS!

- **❶ CREATE A NEW CONSOLE APPLICATION AND ADD A NEW CLASS FILE CALLED ISTINGPATROL.CS.**
  The IDE will add a file that has the line `class IStingPatrol` as usual. Replace that line with **`interface IStingPatrol`**, and type in the **`IStingPatrol`** interface from two pages ago. You've now *added an interface* to your project! Your program should now compile.
- **❷ ADD A BEE CLASS TO THE PROJECT.**
  Don't add any properties or methods yet. Just have it implement `IStingPatrol`:

```
class Bee : IStingPatrol
{
}
```

- **❸ TRY TO COMPILE THE PROGRAM.**
  Select Rebuild from the Build menu. Uh oh — the compiler won't let you do it:



NOTE

You'll see one of these "does not implement" errors for every member of IStingPatrol that's not implemented in the class. The compiler **really** wants you to implement every method in the interface.

- **❹ ADD THE METHODS AND PROPERTIES TO THE BEE CLASS.**
  Add a `LookForEnemies()` method and a `SharpenStinger()` method. Make sure that their signatures match the ones in the interface — so `LookForEnemies()` has to return a `bool`, and `SharpenStinger()` takes an `int` parameter (choose any name) and returns an `int`; they don't have to do anything for now, so just return dummy values. Add an `int` property called `AlertLevel` with a get accessor (have it return any number), and an automatic `int` property called `StingerLength` with get and set accessors. One more thing: make sure all the `Bee` members are marked `public`. Now the program will compile!

Get a little practice using interfaces

Interfaces are really easy to use, and the best way to understand them is to start using them. So **create a new Console Application** project and get started!

DO THIS!

- ❶ Here's the `TallGuy` class, and the code for the `Main()` method in *Program.cs* 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.");
    }
}
static void Main(string[] args) {
    TallGuy tallGuy = new TallGuy() { Height = 74, Name = "Jimmy" };
    tallGuy.TalkAboutYourself();
}
```

- ❷ 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 this declaration:

```
interface IClown
{
```

Now try to declare a private method inside the interface:

```
private void Honk();
```

Select Build→Build Solution in the IDE. You'll see this error:

Now go ahead and **delete the `private` access modifier** — the error will go away and your program will compile just fine.

NOTE

You don't need to type "public" inside the interface, because it automatically makes every property and method public.

- ❸ Before you go on to the next page, see if you can create the rest of the `IClown` interface, and modify the `TallGuy` class to implement this interface. Your new `IClown` interface should have a `void` method called `Honk` that doesn't take any parameters, and a `string` read-only property called `FunnyThingIHave` that has a get accessor but no set accessor.
- ❹ Here's the interface — did you get it right?

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();
}
```

Here's an example of an interface that has a get accessor without a set accessor. Remember, interfaces can't contain fields, but when you implement this read-only property in a class, it'll look like a field to other objects.

OK, 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
```

TallGuy will implement the IClown interface.

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, including this one:

What the IDE is telling you is that when you said TallGuy would implement IClown, you promised to add all of the properties and methods in that interface...and then you broke that promise!

```
'TallGuy' does not implement interface
member 'IClown.Honk()'
```

- ❺ 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!");
}
```

All the interface requires is that a class that implements it has a property called FunnyThingIHave with a get accessor. You can put any get accessor in there, even one that just returns the same string every time. Most get accessors won't do this, but this will work just fine if it does what you need it to do.

The interface 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.

- ❻ Now your code will compile! Update your `Main()` method so that it calls the `TallGuy` object's `Honk()` method to print the "Honk honk!" line to the console.

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

Say you had a method that needed an object that could perform the `FindFlowers()` method. Any object that implemented the `INectarCollector` interface would do. It could be a `Worker` object, `Robot` object, or `Dog` object, as long as it implements the `INectarCollector` interface.

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.

This won't work...

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.

```
IStingPatrol dennis = new IStingPatrol();
```

1 Cannot create an instance of the abstract class or interface

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 this will.

```
NectarStinger fred = new NectarStinger();
IStingPatrol george = fred;
```

Remember how you could pass a BLT reference into any class that expects a Sandwich, because BLT inherits from Sandwich? Well, this is the same thing—you can use a NectarStinger in any method or statement that expects an IStingPatrol.

Even though this object can do more, when you use an interface reference you only have access to the methods in the interface.

The first line is an ordinary `new` statement, creating a reference called `Fred` and pointing it to a `NectarStinger` object.

The second line is where things start to get interesting, because that line of code **creates a new reference variable using `IStingPatrol`**. That line may look a little odd when you first see it. But look at this:

```
NectarStinger ginger = fred;
```

You know what this third statement does — it creates a new `NectarStinger` reference called `ginger` and points it at whatever object `fred` is pointing to. The `george` statement uses `IStingPatrol` the same way.

So what happened?

There's only one `new` statement, so **only one object** was created. The second statement created a reference variable called `george` that can point to an instance of **any class that implements `IStingPatrol`**.

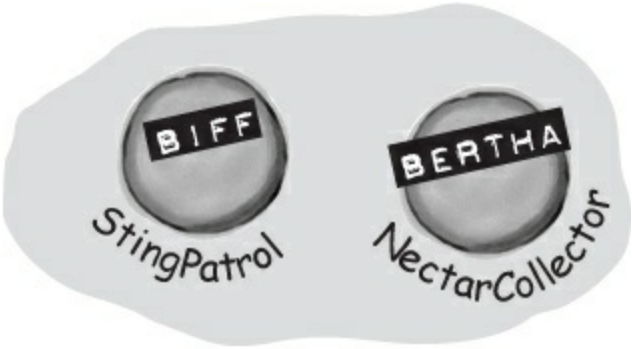
Interface references work just like object references

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.

- **❶ OBJECTS ARE CREATED AS USUAL.**
  Both of these classes implement `IStingPatrol`.

```
StingPatrol biff = new StingPatrol();
NectarCollector bertha = new NectarCollector();
```

Let's assume that StingPatrol implements the IStingPatrol interface and NectarCollector implements the INectarCollector interface.



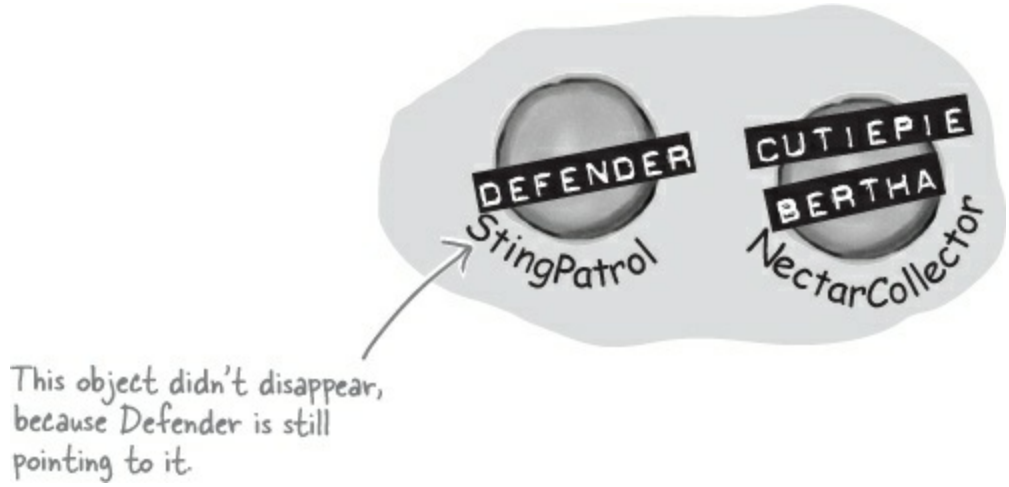
- **❷ ADD ISTINGPATROL AND INECTARCOLLECTOR REFERENCES.**
  You can use interface references just like you use any other reference type.

```
IStingPatrol defender = biff;
INectarCollector cutiePie = bertha;
```

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.

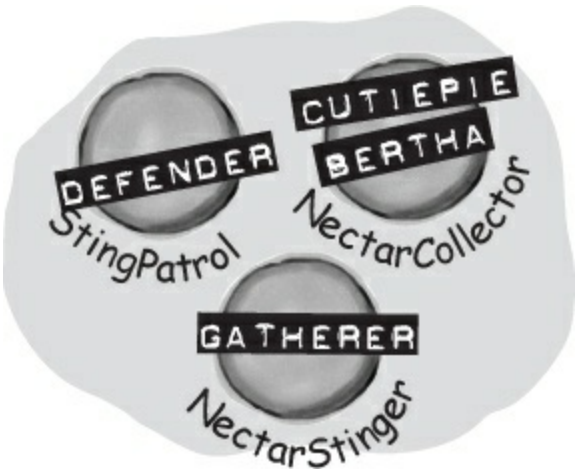
- **❸ AN INTERFACE REFERENCE WILL KEEP AN OBJECT ALIVE.**
  When there aren't any references pointing to an object, it disappears. But there's no rule that says those references all have to be the same type! An interface reference is just as good as an object reference when it comes to keeping track of objects.

```
biff = null;
```



- **❹ ASSIGN A NEW INSTANCE TO AN INTERFACE REFERENCE.**
  You don't actually *need* an object reference — you can create a new object and assign it straight to an interface reference variable.

```
INectarCollector gatherer = new NectarStinger();
```



You can find out if a class implements a certain interface with “is”

Sometimes you need to find out if a certain class implements an interface. Suppose we have all our worker bees in an array, called `Bees`. We can make the array hold the type `Worker`, since all worker bees will be `Worker` classes, or subclasses of that type.

But which of the worker bees can collect nectar? In other words, we want to know if the class implements the `INectarCollector` interface. We can use the **`is`** keyword to find out exactly that.

All the workers are in an array of Workers. We'll use "is" to sort out which type of worker each bee is.

We've got an array of Worker bees who are all eligible to go on a nectar-collecting mission. So we'll loop through the array, and use "is" to figure out which ones have the right methods and properties to do the job.

```
Worker[] bees = new Worker[3];
bees[0] = new NectarCollector();
bees[1] = new StingPatrol();
bees[2] = new NectarStinger();
for (int i = 0; i < bees.Length; i++)
{
    if (bees[i] is INectarCollector)
    {
        bees[i].DoThisJob("Nectar Collector", 3);
    }
}
```

is lets you compare interfaces AND also other types, too!

This is like saying, if this bee implements the INectarCollector interface...do this.

Now that we know the bee is a nectar collector, we can assign it the job of collecting nectar.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Wait a minute. 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: **A:** No, not at all. It's true that a property inside an interface looks very similar to an automatic property — like `Job` and `ShiftsLeft` in the `IWorker` interface on the next page. But they're definitely not automatic properties. You could implement `Job` 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 "Accountant";
  }
}
```

and the compiler will be perfectly happy with that, too. You can also add a set accessor — 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.)

BRAIN POWER

If you have some other class that doesn't inherit from `Worker` but ***does*** implement the `INectarCollector` interface, then it'll be able to do the job, too! But since it doesn't inherit from `Worker`, you can't get it into an array with other bees. Can you think of a way to get around the problem and create an array with both bees and this new class?

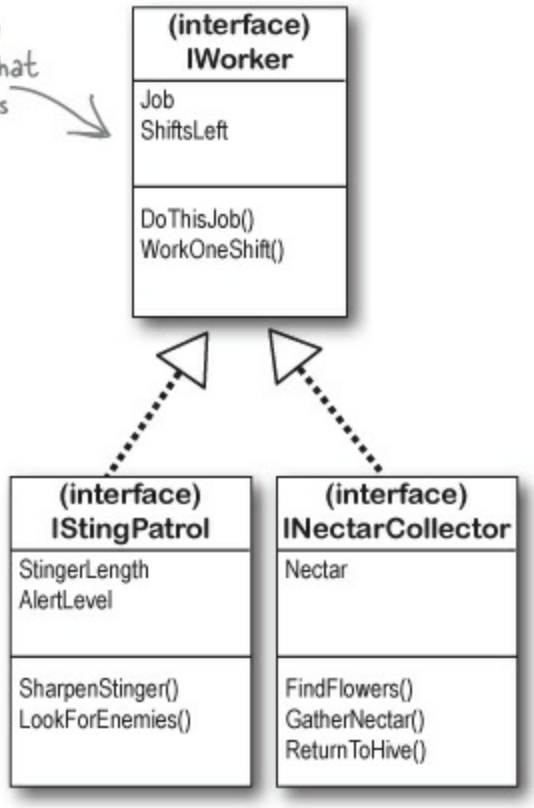
Interfaces can inherit from other interfaces

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 simply accumulate all of the methods and properties from the interfaces they inherit from.

```
interface IWorker
{
  string Job { get; }
  int ShiftsLeft { get; }
  void DoThisJob(string job, int shifts);
  void WorkOneShift();
}
```

We've created a new IWorker interface that the other interfaces inherit from.

When we draw an interface on a class diagram, we'll show inheritance using dashed lines.



Any class that implements an interface that inherits from IWorker must implement its methods and properties

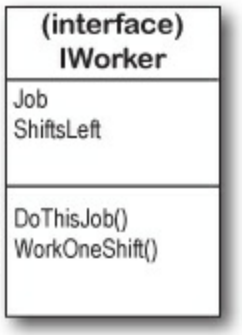
When a class implements an interface, it has to include every property and method in that interface. And if that interface inherits from another one, then all of *those* properties and methods need to be implemented, too.

```
interface IStingPatrol : IWorker
{
    int AlertLevel { get;}
    int StingerLength { get; set;}
    bool LookForEnemies();
    int SharpenStinger(int length);
}
```

Here's the same IStingPatrol interface, but now it inherits from the IWorker interface. It looks like a tiny change, but it makes a huge difference in any class that implements IStingPatrol.

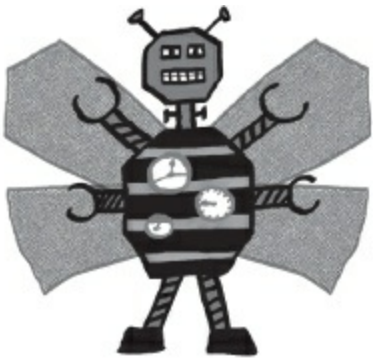
A class that implements IStingPatrol must not only implement these methods...

...but the methods and properties of the IWorker interface this interface inherits from, too.



The RoboBee 4000 can do a worker bee's job without using valuable honey

Let's create a new bee, a RoboBee 4000, that runs on gas. We can have it inherit from the `IWorker` interface, though, so it can do everything a normal worker bee can.



This is our basic Robot class, so robots can run on gasoline.

```
class Robot
{
     public void ConsumeGas() {...}
}

class RoboBee : Robot, IWorker
{
     private int shiftsToWork;
     private int shiftsWorked;
     public int ShiftsLeft
       {get {return shiftsToWork - shiftsWorked;}}
     public string Job { get; private set; }
     public bool DoThisJob(string job, int shiftsToWork){...}
     public void WorkOneShift() {...}
}
```

The RoboBee class inherits from Robot and implements IWorker. That means it's a robot, but can do the job of a worker bee. Perfect!

The RoboBee class implements all the methods from the IWorker interface.

If RoboBee didn't implement everything in the IWorker interface, the code wouldn't compile.

Remember, for other classes in the application, there's no functional difference between a `RoboBee` and a normal worker bee. They both implement the interface, so both act like worker bees as far as the rest of the program is concerned.

But, you could distinguish between the types by using:

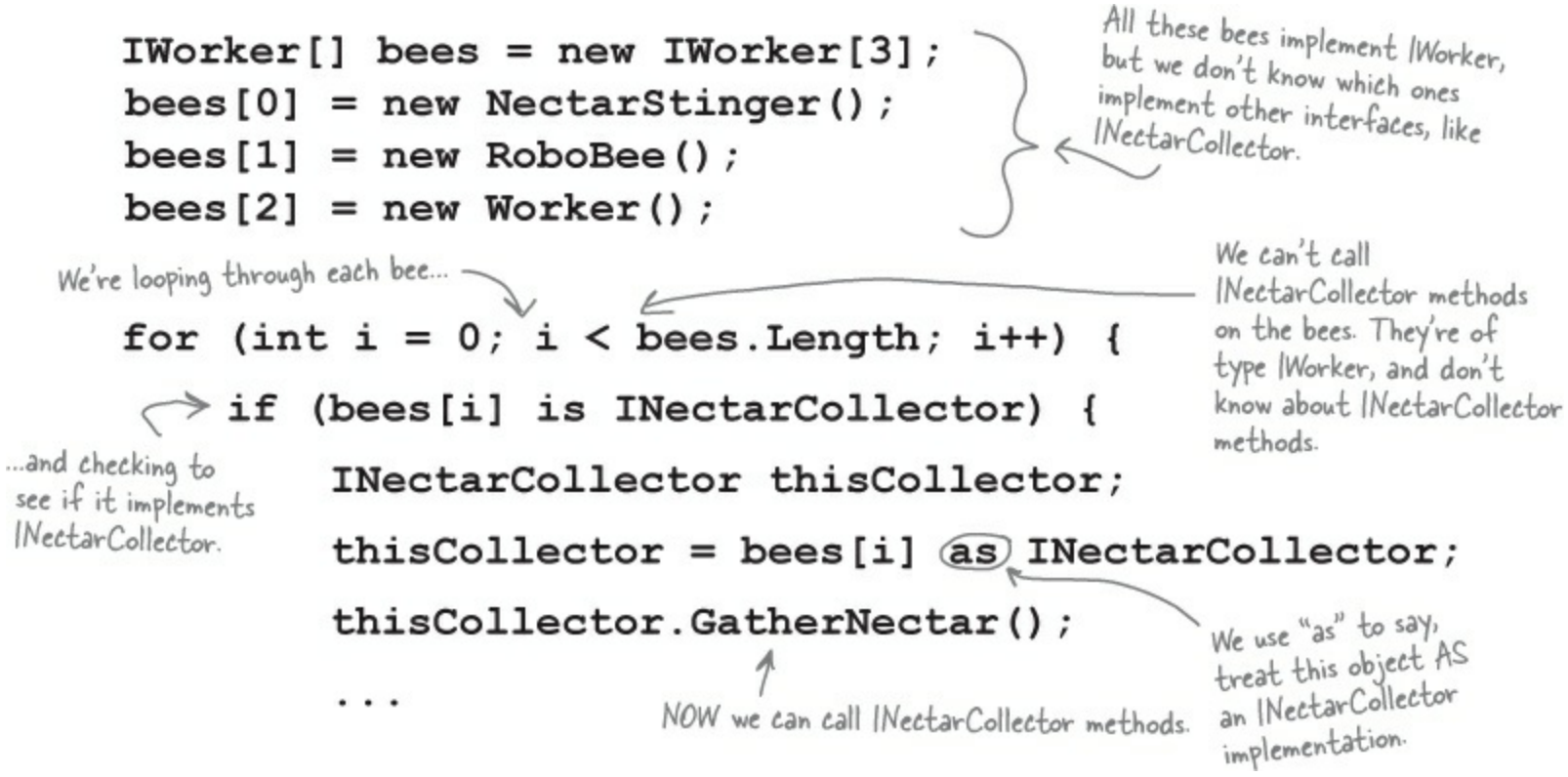
```
if (workerBee is Robot) {
  // now we know workerBee
  // is a Robot object
}
```

We can see what class or interface workerBee subclasses or implements with "is".

Any class can implement ANY interface as long as it keeps the promise of implementing the interface's methods and properties.

is tells you what an object implements; as tells the compiler how to treat your object

Sometimes you need to call a method that an object gets from an interface it implements. But what if you don't know if that object is the right type? You use **`is`** to find that out. Then, you can use **`as`** to treat that object — which you now know is the right type — as having the method you need to call.



SHARPEN YOUR PENCIL

Take a look at the array on the left. For each of these statements, write down which values of `i` would make it evaluate to `true`. Also, two of them won't compile — cross those lines out.

```
IWorker[] Bees = new IWorker[8];
Bees[0] = new NectarStinger();
Bees[1] = new RoboBee();
Bees[2] = new Worker();
Bees[3] = Bees[0] as IWorker;
Bees[4] = IStingPatrol;
Bees[5] = null;
Bees[6] = Bees[0];
Bees[7] = new INectarCollector();
```

1. `(Bees[i] is INectarCollector)`

   _____

2. `(Bees[i] is IStingPatrol)`

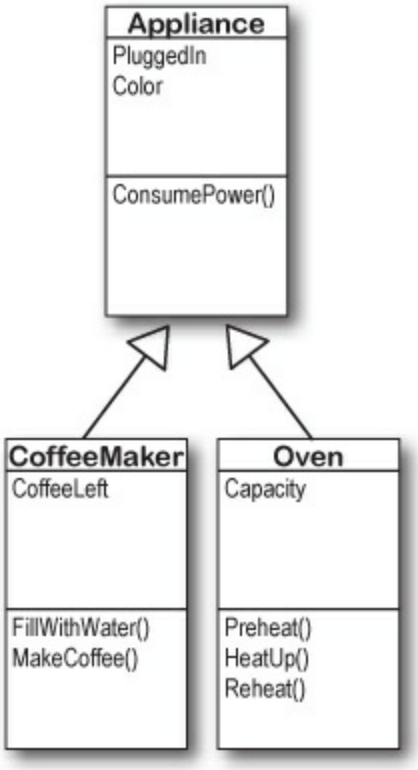
   _____

3. `(Bees[i] is IWorker)`

   _____

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 needed to be able to distinguish a coffee maker from an oven, you'd have to build a class hierarchy. So you'd add the methods and properties that are specific to a coffee maker or oven to some `CoffeeMaker` and `Oven` classes, and they'd inherit from an `Appliance` class that has their common methods and properties.




```
public void MonitorPower(Appliance appliance) {
    // code to add data to a household
    // power consumption database
}
```

Here's a method in the program to monitor the power consumption for a house.

This code would appear later on in the program to monitor the coffee maker's power consumption.

```
    CoffeeMaker misterCoffee = new CoffeeMaker();
    MonitorPower(misterCoffee);
```

Even though the MonitorPower() method takes a reference to an Appliance object, you can pass it the misterCoffee reference because CoffeeMaker is a subclass of Appliance.

You already saw this in the last chapter, when you saw how you could pass a BLT reference to a method that expected a Sandwich.

SHARPEN YOUR PENCIL SOLUTION

Take a look at the array on the left. For each of these statements, write down which values of `i` would make it evaluate to `true`. Also, two of them won't compile — cross them out.

```
IWorker[] Bees = new IWorker[8];
Bees[0] = new NectarStinger();
Bees[1] = new RoboBee();
Bees[2] = new Worker();
Bees[3] = Bees[0] as IWorker;
Bees[4] = IStingPatrol;
Bees[5] = null;
Bees[6] = Bees[0];
Bees[7] = new INectarCollector();
```

NectarStinger() implements the IStingPatrol interface.

1. (Bees[i] is INectarCollector)

   0, 3, and 6

2. (Bees[i] is IStingPatrol)

   0, 3, and 6

3. (Bees[i] is IWorker)

   0, 1, 2, 3, and 6

Upcasting works with both objects and interfaces

When you substitute a subclass for a base class — like substituting a coffee maker for an appliance, or a BLT for a sandwich — 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).

- **❶ LET'S CREATE SOME OBJECTS.**
  We can create a `CoffeeMaker` and `Oven` class as usual:

```
CoffeeMaker misterCoffee = new CoffeeMaker();
Oven oldToasty = new Oven();
```

We'll start by instantiating an Oven object and a CoffeeMaker object as usual.

- **❷ 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.

- **❸ BUT YOU CAN'T TREAT 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`:

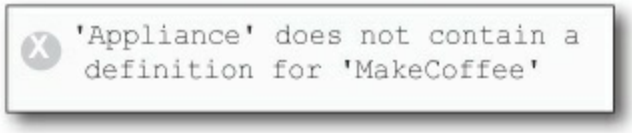
```
Appliance powerConsumer = new CoffeeMaker();
powerConsumer.ConsumePower();
```

  But as soon as you try to use it like a `CoffeeMaker`:

```
powerConsumer.MakeCoffee();
```

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:



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 lets you turn your appliance back into a coffee maker

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 once you know that, you can convert the `Appliance` back to a `CoffeeMaker` using the **`as`** keyword.

- **❶ We'll start with the `CoffeeMaker` we already upcast.**
  Here's the code that we used:

```
Appliance powerConsumer = new CoffeeMaker();
powerConsumer.ConsumePower();
```

- **❷ But what if we want to turn the `Appliance` back into a `CoffeeMaker`?**
  The first step in downcasting is using the `is` keyword to check if it's even an option.

```
if (powerConsumer is CoffeeMaker)
    // then we can downcast!
```

- **❸ Now that 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) {
   CoffeeMaker javaJoe = powerConsumer as CoffeeMaker;
   javaJoe.MakeCoffee();
}
```

The javaJoe reference points to the same CoffeeMaker object as powerConsumer. But it's a CoffeeMaker reference, so it can call the MakeCoffee() method.

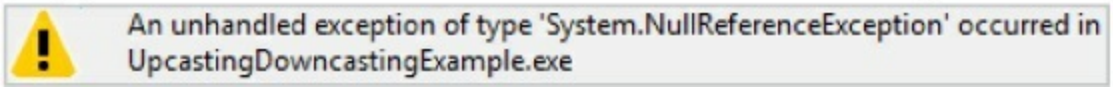
When downcasting fails, as returns null

So what happens if you try to use `as` to convert an `Oven` object into a `CoffeeMaker`? It returns `null` — and if you try to use it, .NET will cause your program to break.

```
if (powerConsumer is CoffeeMaker) {
    Oven foodWarmer = powerConsumer as Oven;
    foodWarmer.Preheat();
}
```

Uh oh, these don't match!

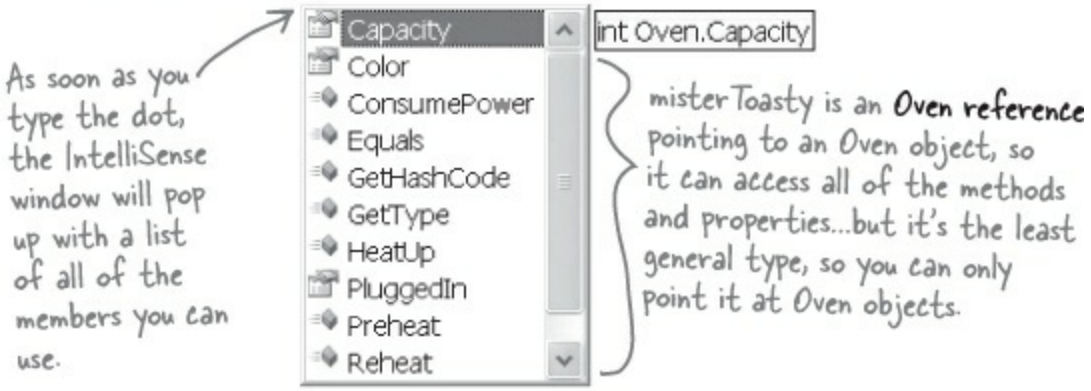
powerConsumer is NOT an Oven object. So when you try to downcast it with "as", the foodWarmer reference ends up set to null. And when you try to use a null reference, this happens when you run the program...



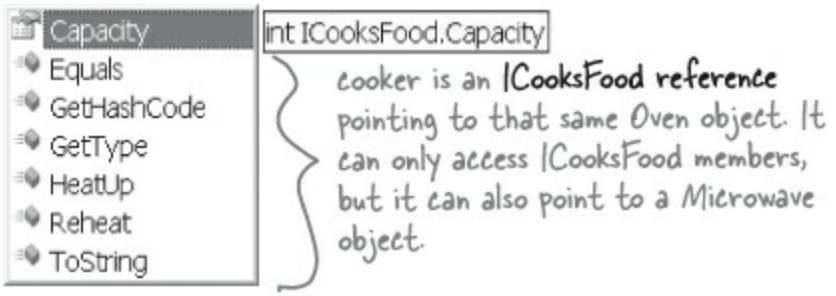
Upcasting and downcasting work with interfaces, too

You already know that `is` and `as` work with interfaces. Well, so do all of the upcasting and downcasting tricks. 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. And the IDE's IntelliSense can help you figure out exactly what you can and can't do with each of them:

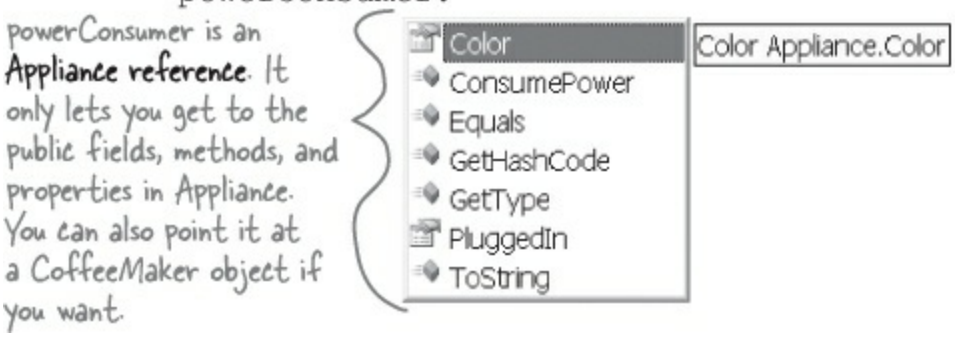
```
Oven misterToasty = new Oven();
misterToasty.
```

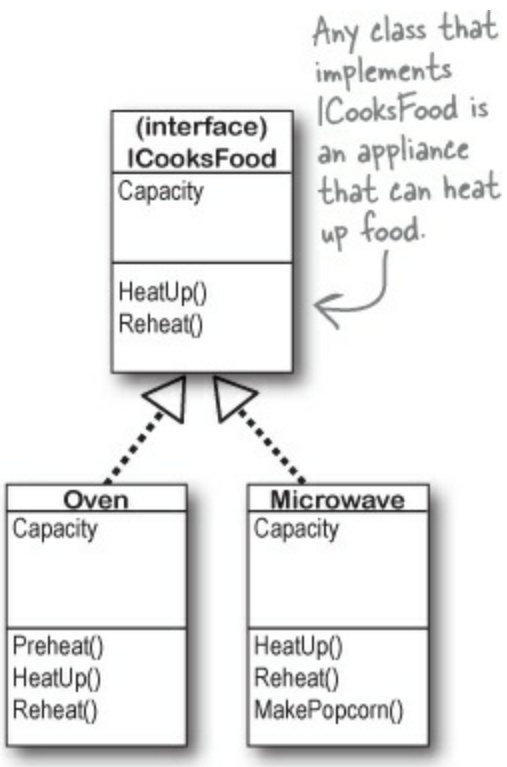



```
ICooksFood cooker;
if (misterToasty is ICooksFood)
    cooker = misterToasty as ICooksFood;
    cooker.
```




```
Appliance powerConsumer;
if (misterToasty is Appliance)
    powerConsumer = misterToasty;
    powerConsumer.
```





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: **Q: So back up — you told me that I can always upcast but I can't always downcast. Why?**

A: **A:** Because the compiler can warn you if your upcast is wrong. The only time an upcast won't work is 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. And the compiler can figure out immediately that you didn't upcast properly, and will give you an error.

On the other hand, the compiler doesn't know how to check if you're downcasting *from* an object or interface reference *to* a reference that's not valid. That's because it's perfectly legal to put any class or interface name on the righthand side of the `as` keyword. If the downcast is illegal, then the `as` statement will just return `null`. And it's a good thing that the compiler doesn't stop you from doing that, because there are plenty of times when you'd want to do it.

Q: **Q: Someone told me that an interface is like a contract, but I don't really get why. What does that mean?**

A: **A:** Yes, we've heard that too — a lot of people like to say that an interface is like a contract. (That's a really common question on job interviews.) And it's true, to some extent. 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.

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: **Q: What if I want to put a method body into my interface? Is that OK?**

A: **A:** No, the compiler won't let you do that. An interface isn't allowed to have any statements in it at all. Even though you use the colon operator to implement an interface, it's not the same thing as inheriting from a class. Implementing an interface doesn't add any behavior to your class at all, or make any changes to it. All it does is tell the compiler to make sure that your class has all of the methods that the interface says it should have.

Q: **Q: Then why would I want to use an interface? It seems like it's just adding restrictions, without actually changing my class at all.**

A: **A:** Because when your class implements an interface, then an interface reference can point to any instance of that class. 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.

Q: **Q: Is there an easier way to implement interfaces? It's a lot of typing!**

A: **A:** Why, yes, there is! The IDE gives you a very powerful shortcut that automatically implements an interface for you. Just start typing your class:

```
class
   Microwave : ICooksFood
   { }
```

Click on `ICooksFood` — you'll see a small bar appear underneath the "I". Hover over it and you'll see an icon appear underneath it:

NOTE

Sometimes it's hard to click on the icon, but Ctrl-period will work, too.

Click on the icon and choose "Implement Interface 'ICooksFood'" from the menu. It'll automatically add any members that you haven't implemented yet. Each one has a single `throw` statement in it — they'll cause your program to halt, as a reminder in case you forget to implement one of them. (You'll learn about `throw` in Chapter 10.)

An interface is like a checklist that the compiler runs through to make sure your class implemented a certain set of methods.

EXERCISE

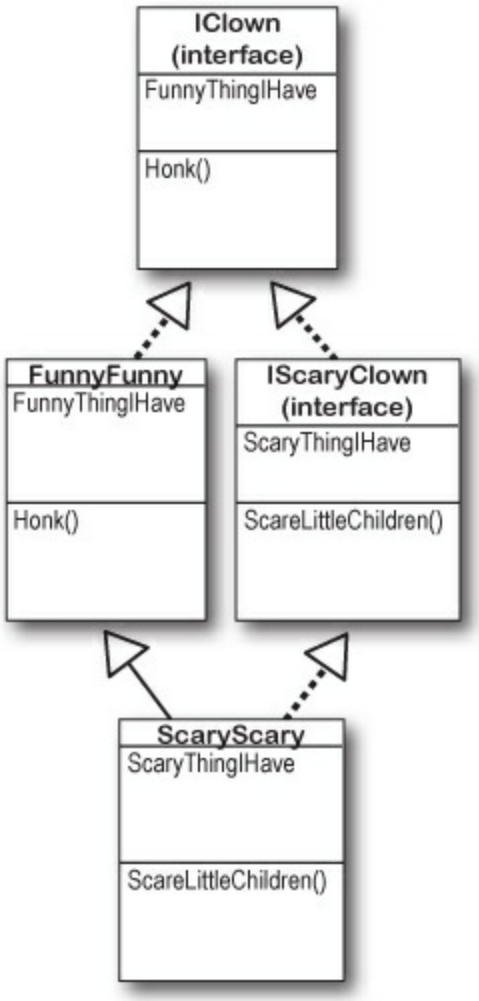
Extend the `IClown` interface and use classes that implement it by adding more code to the Console application you created earlier.

- ❶ Start with the `IClown` interface from the last "Do this!" in Get a little practice using interfaces:

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}
```

- ❷ Extend `IClown` by creating a new interface, `IScaryClown`, that inherits from `IClown`. It should have an additional `string` property called `ScaryThingIHave` with a get accessor but no set accessor, and a `void` method called `ScareLittleChildren()`.
- ❸ Create these classes:

A funny clown class called `FunnyFunny` that uses a private string variable to store a funny thing. Use a constructor that takes a parameter called `funnyThingIHave` and uses it to set the private field. The `Honk()` method should print: *"Hi kids! I have a "* followed by the funny thing it has. The `FunnyThingIHave` get accessor should return the same thing.
A scary clown class called `ScaryScary` that uses a private variable to store an integer that was passed to it by its constructor in a parameter called `numberOfScaryThings`. The `ScaryThingIHave` get accessor should return a string consisting of the number from the constructor followed by *"spiders"*. The `ScareLittleChildren()` prints a message that says, *"Boo! Gotcha!"*



- ❹ Here's new code for the `Main()` method — but it's not working. Can you figure out how to fix it?

```
static void Main(string[] args) {
    ScaryScary fingersTheClown = new ScaryScary("big shoes", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryClown = someFunnyClown;
    someOtherScaryClown.Honk();
    Console.ReadKey();
}
```

Fingers the Clown is scary.

YOU BETTER GET THIS ONE RIGHT... OR ELSE!

EXERCISE SOLUTION

Extend the `IClown` interface and use classes that implement it.

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}

interface IScaryClown : IClown {
    string ScaryThingIHave { get; }
    void ScareLittleChildren();
}

class FunnyFunny : IClown {
    public FunnyFunny(string funnyThingIHave) {
        this.funnyThingIHave = funnyThingIHave;
    }
    private string funnyThingIHave;
    public string FunnyThingIHave {
        get { return "Hi kids! I have " + funnyThingIHave; }
    }
    public void Honk() {
        Console.WriteLine(this.FunnyThingIHave);
    }
}

class ScaryScary : FunnyFunny, IScaryClown {
    public ScaryScary(string funnyThingIHave, int numberOfScaryThings)
        : base(funnyThingIHave) {
        this.numberOfScaryThings = numberOfScaryThings;
    }
    private int numberOfScaryThings;
    public string ScaryThingIHave {
        get { return "I have " + numberOfScaryThings + " spiders"; }
    }
    public void ScareLittleChildren() {
        Console.WriteLine("Boo! Gotcha!");
    }
}

static void Main(string[] args) {
    ScaryScary fingersTheClown = new ScaryScary("big shoes", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryClown = someFunnyClown as ScaryScary;
    someOtherScaryClown.Honk();
    Console.ReadKey();
}
```

The Honk() method just uses this get accessor to display its message—no need to have the same code twice.

You could have implemented the IClown method and property again, but why not just inherit from FunnyFunny?

Since ScaryScary is a subclass of FunnyFunny and FunnyFunny implements IClown, ScaryScary implements IClown too.

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 as keyword.

You can also use the someOtherScaryClown reference to call ScareLittleChildren()—but you can't get to it from the someFunnyClown reference.

There's more than just public and private

You already know how important the `private` keyword is, how you use it, and how it's different from `public`. C# has a name for these keywords: they're called **access modifiers**. The name makes sense, because when you change an access modifier on a property, field, or method of a class — its **members** — or the entire class, you change the way other classes can access it. There are a few more access modifiers that you'll use, but we'll start with the ones you know:

> **NOTE**
>
> We call a class's methods, fields, and properties its members. Any member can be marked with the public or private access modifier.

- **`public` means that anyone can access it.**

  > **NOTE**
  >
  > (as long as they can access the declaring class)

  When you mark a class or class member `public`, you're telling C# that any instance of any other class can access it. It's the least restrictive access modifier. And you've already seen how it can get you in trouble — only mark class members public if you have a reason. That's how you make sure your classes are well encapsulated.

- **`private` means that only other members can access it.**
  When you mark a class member `private`, then it can only be accessed from other members inside that class or **other instances of that class**. You can't mark a class `private` — unless that class **lives inside another class**, in which case it's only available to instances of its container class. Then it's private by default, and if you want it to be public you need to mark it public.

  > **NOTE**
  >
  > If you leave off the access modifier when you declare a class member, it defaults to private.

- **`protected` means public to subclasses, private to everyone else.**
  You've already seen how a subclass can't access the private fields in its base class — it has to use the `base` keyword to get to the private members of the base object. Wouldn't it be convenient if the subclass could access those private fields? That's why you have the `protected` access modifier. Any class member marked `protected` can be accessed by any other member of its class, and any member of a subclass of its class.

- **`internal` means public only to other classes in an assembly.**
  The built-in .NET Framework classes and all of the code in your projects are in **assemblies** — libraries of classes that are in your project's list of references. You can see a list of assemblies by right-clicking on References in the Solution Explorer and choosing "Add Reference..." — when you create a new Windows Forms application, the IDE automatically includes the references you need to build a Windows application. When you build an assembly, you can use the `internal` keyword to keep classes private to that assembly, so you can only expose the classes you want. You can combine this with `protected` — anything you mark `protected internal` can only be

accessed from within the assembly **or** from a subclass.

> **NOTE**
>
> If you leave off the access modifier when you declare a class or an interface, then by default it's set to internal. And that's just fine for most classes — it means that any other class in the assembly can read it. If you're not using multiple assemblies, internal will work just as well as public for classes and interfaces. Give it a shot — go to an old project, change some of the classes to internal, and see what happens.

- **`sealed` says that this class can't be subclassed.**
  There are some classes that you just can't inherit from. A lot of the .NET Framework classes are like this — go ahead, try to make a class that inherits from `String` (that's the class whose `IsEmptyOrNull()` method you used in the last chapter). What happens? The compiler won't let you build your code — it gives you the error "cannot derive from sealed type 'string'". You can do that with your own classes — just add `sealed` after the access modifier.

> **NOTE**
>
> Sealed is a modifier, but it's not an access modifier. That's because it only affects inheritance — it doesn't change the way the class can be accessed.

There's a little more to all of these definitions. Take a peek at leftover #3 in the appendix to learn more about them.

Access modifiers change visibility

Let's take a closer look at the access modifers and how they affect the **scope** of the various class members. We'll make two changes: the `funnyThingIHave` backing field will be `protected`, and we'll change the `ScareLittleChildren()` method so that it uses the `funnyThingIHave` field:

> **NOTE**
>
> Make these two changes to your own exercise solution. Then change the protected access modifier back to private and see what errors you get.

- ❶ Here are two interfaces. `IClown` defines a clown who honks his horn and has a funny thing. `IScaryClown` inherits from clown. A scary clown does everything a clown does, plus he has a scary thing and scares little children. (These haven't changed from earlier.)

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}

interface IScaryClown : IClown {
    string ScaryThingIHave { get; }
    void ScareLittleChildren();
}
```

- ❷ The `FunnyFunny` class implements the `IClown` interface. We changed the `funnyThingIHave` field to `protected` so that it can be accessed by any instance of a subclass of `FunnyFunny`.

The "this" keyword also changes what variable you're referring to. It says to C#, "Look at the current instance of the class to find whatever I'm connected to—even if that matches a parameter or local variable."

This is a really common way to use "this", since the parameter and backing field have the same name. funnyThingIHave refers to the parameter, while this.funnyThingIHave is the backing field.

By adding "this", we told C# that we're talking about the backing field, not the parameter that has the same name.

We changed FunnyThingIHave to protected. Look and see how it affects the ScaryScary.ScareLittleChildren() method.

```
class FunnyFunny : IClown {
    public FunnyFunny(string funnyThingIHave) {
        this.funnyThingIHave = funnyThingIHave;
    }
    protected string funnyThingIHave;
    public string FunnyThingIHave {
        get { return "Hi kids! I have " + funnyThingIHave; }
    }

    public void Honk() {
        Console.WriteLine(this.FunnyThingIHave);
    }
}
```

When you use "this" with a property, it tells C# to execute the set or get accessor.

- ❸ The `ScaryScary` class implements the `IScaryClown` interface. It also inherits from `FunnyFunny`, and since `FunnyFunny` implements `IClown`, that means `ScaryScary` does, too. Take a look at how the `ScareLittleChildren()` method accesses the `funnyThingIHave` backing field — it can do that because we used the `protected` access modifier. If we'd made it `private` instead, then this code wouldn't compile.

Access Modifiers Up Close

```
class ScaryScary : FunnyFunny, IScaryClown {
    public ScaryScary(string funnyThingIHave,
                      int numberOfScaryThings)
        : base(funnyThingIHave) {
        this.numberOfScaryThings = numberOfScaryThings;
    }

    private int numberOfScaryThings;
    public string ScaryThingIHave {
        get { return "I have " + numberOfScaryThings + " spiders"; }
    }

    public void ScareLittleChildren() {
        Console.WriteLine("You can't have my "
                          + base.funnyThingIHave);
    }
}
```

numberOfScaryThings is private, which is typical of a backing field. So only another instance of ScaryScary would be able to see it.

The protected keyword tells C# to make something private to everyone **except** instances of a subclass.

The "base" keyword tells C# to use the value from the base class. But we could also use "this" in this case. Can you figure out why?

If we'd left funnyThingIHave private, this would cause the compiler to give you an error. But when we changed it to protected, that made it visible to any subclass of FunnyFunny.

- ❹ Here's a `Main()` method that instantiates `FunnyFunny` and `ScaryScary`. Take a look at how it uses `as` to downcast `someFunnyClown` to an `IScaryClown` reference.

```
static void Main(string[] args) {
    ScaryScary fingersTheClown = new ScaryScary("big shoes", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryClown = someFunnyClown as ScaryScary;
    someOtherScaryClown.Honk();
    Console.WriteLine(someOtherScaryClown.ScaryThingIHave);
    someOtherScaryClown.ScareLittleChildren();
    Console.ReadKey();
}
```

Since the Main() method isn't part of FunnyFunny or ScaryScary, it can't access the protected funnyThingIHave field.

It's outside of both classes, so the statements inside it only have access to the public members of any FunnyFunny or ScaryScary objects.

We put in some extra steps to show you that you could upcast ScaryScary to FunnyFunny, and then downcast that to IScaryClown. But all three of those lines could be collapsed into a single line. Can you figure out how?

THERE ARE NO DUMB QUESTIONS

Q: **Q: Why would I want to use an interface instead of just writing all of the methods I need directly into my class?**

A: **A:** You might end up with a lot of different classes as you write more and more complex programs. 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 implement `ICarryPassenger`. Say the `ICarryPassenger` interface stipulates that any class that implements it has to have a `ConsumeEnergy()` method. Your program could use them both to carry passengers even though the `Sailboat` class's `ConsumeEnergy()` method uses wind power and the `Truck` class's method uses diesel fuel.

Imagine if you didn't have the `ICarryPassenger` interface. Then it would be tough to tell your program which vehicles could carry people and which couldn't. You would have to look through each class that your program might use and figure out whether or not there was a method for carrying people from one place to another. Then you'd have to call each of the vehicles your program was going to use with whatever method was defined for carrying passengers. And since there's no standard interface, they could be named all sorts of things or buried inside other methods. You can see how that'll get confusing pretty fast.

Q: **Q: Why do I need to use a property? Can't I just include a field?**

A: **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: **Q: What's the difference between a regular object reference and an interface reference?**

A: **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: **Q: Then why would I ever want to use an interface reference, if it limits what I can do with the object?**

A: **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: **Q: Why would I make something protected instead of private or public?**

A: **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.

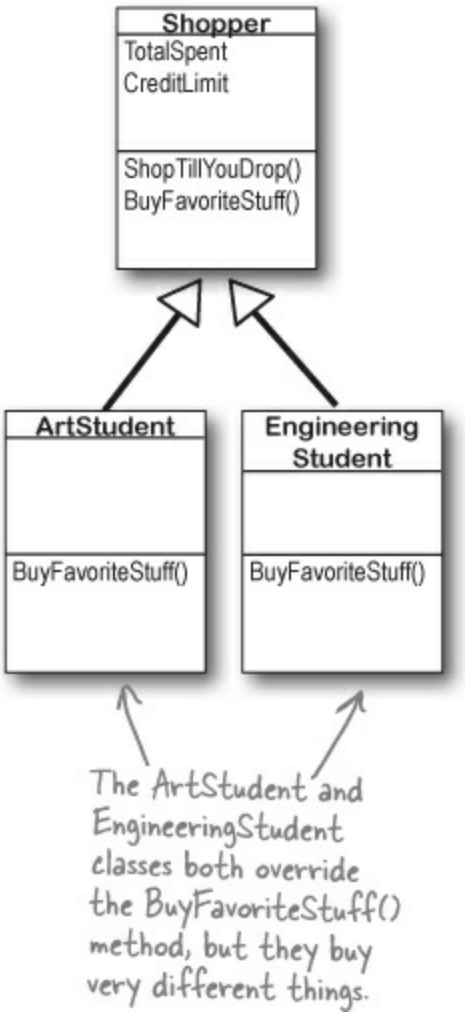
Interface references only know about the methods and properties that are defined in the interface.

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. Here's an example.

Let's start with a basic class for a student shopping at the student bookstore.

```
class Shopper {
    public void ShopTillYouDrop() {
        while (TotalSpent < CreditLimit)
            BuyFavoriteStuff();
    }
    public virtual void BuyFavoriteStuff () {
        // No implementation here - we don't know
        // what our student likes to buy!
    }
}
```



Here's the `ArtStudent` class — it subclasses `Shopper`:

```
class ArtStudent : Shopper {
    public override void BuyFavoriteStuff () {
        BuyArtSupplies();
        BuyBlackTurtlenecks();
        BuyDepressingMusic();
    }
}
```

And the `EngineeringStudent` class also inherits from `Shopper`:

```
class EngineeringStudent : Shopper {
    public override void BuyFavoriteStuff () {
        BuyPencils();
        BuyGraphingCalculator();
        BuyPocketProtector();
    }
}
```

So what happens when you instantiate `Shopper`? Does it ever make sense to do it?

An abstract class is like a cross between a class and an interface

Suppose you need something like an interface, that requires classes to implement certain methods and properties. But you need to include some code in that interface, so that certain methods don't have to be implemented in each inheriting class. What you want is an **abstract class**. You get the features of an interface, but you can write code in it like a normal class.

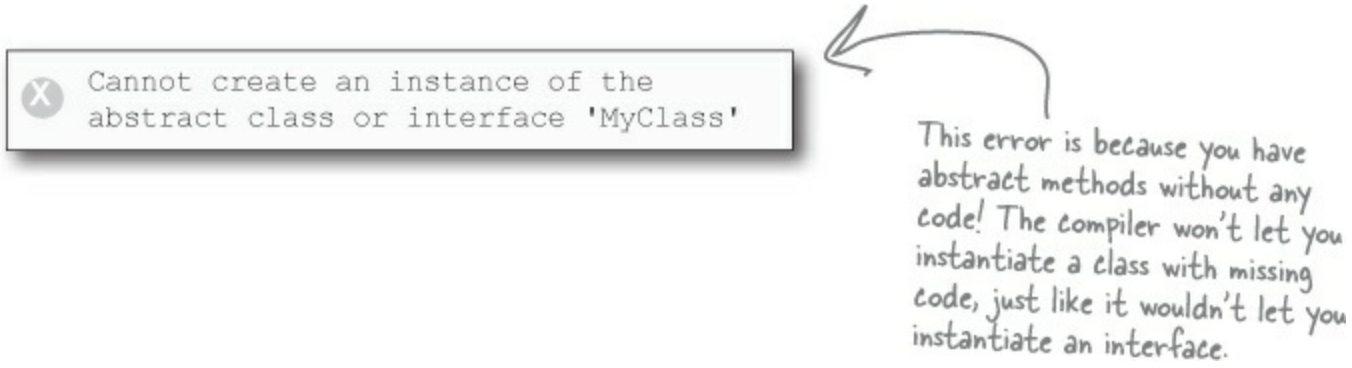
- **AN ABSTRACT CLASS IS LIKE A NORMAL CLASS.**
  You define an abstract class just like a normal one. It has fields and methods, and you can inherit from other classes, too, exactly like with a normal class. There's almost nothing new to learn here, because you already know everything that an abstract class does!
- **AN ABSTRACT CLASS IS LIKE AN INTERFACE.**
  When you create a class that implements an interface, you agree to implement all of the properties and methods defined in that interface. An abstract class works the same way — it can include declarations of properties and methods that, just like in an interface, must be implemented by inheriting classes.

> **NOTE**
>
> A method that has a declaration but no statements or method body is called an **abstract method**. Inheriting classes must implement all abstract methods, just like when they inherit from an interface.
>
> Only abstract classes can have abstract methods. If you put an abstract method 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.

- **But an abstract class can't be instantiated.**

> **NOTE**
>
> 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.



Because you want to provide some 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 created. 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 an example...

Here's a class that the Objectville Astrophysics Club uses to send their rockets to different planets.

```
class PlanetMission {
  public long RocketFuelPerMile;
  public long RocketSpeedMPH;
  public int MilesToPlanet;

  public long UnitsOfFuelNeeded() {
    return MilesToPlanet * RocketFuelPerMile;
  }

  public int TimeNeeded() {
    return MilesToPlanet / (int) RocketSpeedMPH;
  }

  public string FuelNeeded() {
    return "You'll need "
      + MilesToPlanet * RocketFuelPerMile
      + " units of fuel to get there. It'll take "
      + TimeNeeded() + " hours.";
  }
}
```

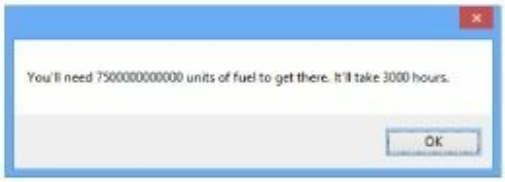
It doesn't make sense to set these fields in the base class, because we don't know what rocket or planet we'll be using.

The astrophysicists have two missions—one to Mars, and one to Venus.

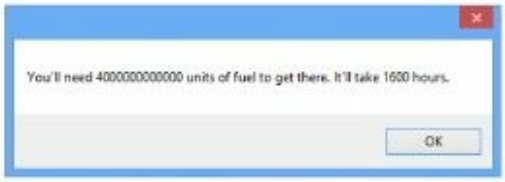
```
class Venus : PlanetMission {
    public Venus() {
        MilesToPlanet = 40000000;
        RocketFuelPerMile = 100000;
        RocketSpeedMPH = 25000;
    }
}

class Mars : PlanetMission {
    public Mars() {
        MilesToPlanet = 75000000;
        RocketFuelPerMile = 100000;
        RocketSpeedMPH = 25000;
    }
}
```

The constructors for the Mars and Venus subclasses set the three fields they inherited from PlanetMission. But those fields **won't get set** if you instantiate PlanetMission directly. So what happens when FuelNeeded() tries to use them?

```
private void button1_Click(object s, EventArgs e) {
  Mars mars = new Mars();
  MessageBox.Show(mars.FuelNeeded());
}
```




```
private void button2_Click(object s, EventArgs e) {
  Venus venus = new Venus();
  MessageBox.Show(venus.FuelNeeded());
}
```




```
private void button3_Click(object s, EventArgs e) {
  PlanetMission planet = new PlanetMission();
  MessageBox.Show(planet.FuelNeeded());
}
```

Before you flip the page, try to figure out what will happen when the user clicks the third button...

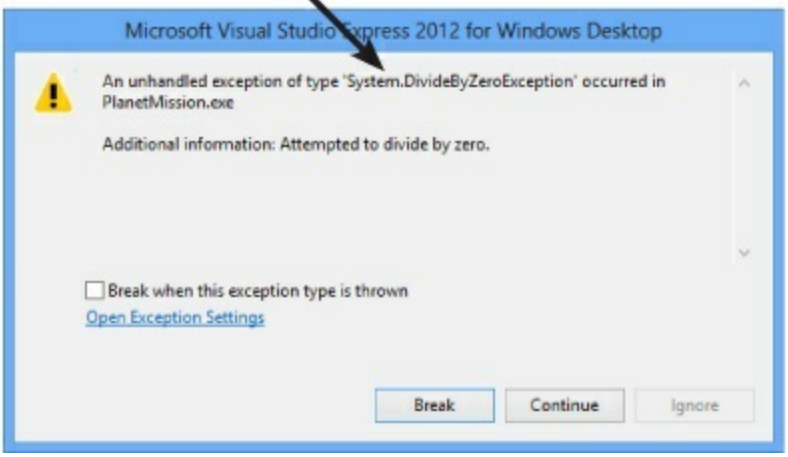
Like we said, some classes should never be instantiated

The problems all start when you create 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...

```
private void button3_Click(object s, EventArgs e) {
  PlanetMission planet = new PlanetMission();
  MessageBox.Show(planet.FuelNeeded());
}
```

The PlanetMission class wasn't written to be instantiated. We were only supposed to inherit from it. But we did instantiate it, and that's where the problems started.

When the FuelNeeded() method tried to divide by RocketSpeedMPH, it was zero. And when you divide by zero, this happens.



Solution: use an abstract class

When you mark a class `abstract`, C# won't let you write code to instantiate it. It's a lot like an interface — it acts like a template for the classes that inherit from it.

Adding the abstract keyword to the class declaration tells C# this is an abstract class, and can't be instantiated.

Now C# will refuse to compile our program until we remove the line that creates an instance of PlanetMission.

```
abstract class PlanetMission {
  public long RocketFuelPerMile;
  public long RocketSpeedMPH;
  public int MilesToPlanet;

  public long UnitsOfFuelNeeded() {
    return MilesToPlanet * RocketFuelPerMile;
  }

  // the rest of the class is defined here
}
```

BRAIN POWER

Flip back to the solution to Kathleen's party planning program in the previous chapter, and take another look at the class hierarchy. Would it ever make sense to instantiate `Party`, or would it make more sense to mark it as abstract to prevent that?

An abstract method doesn't have a body

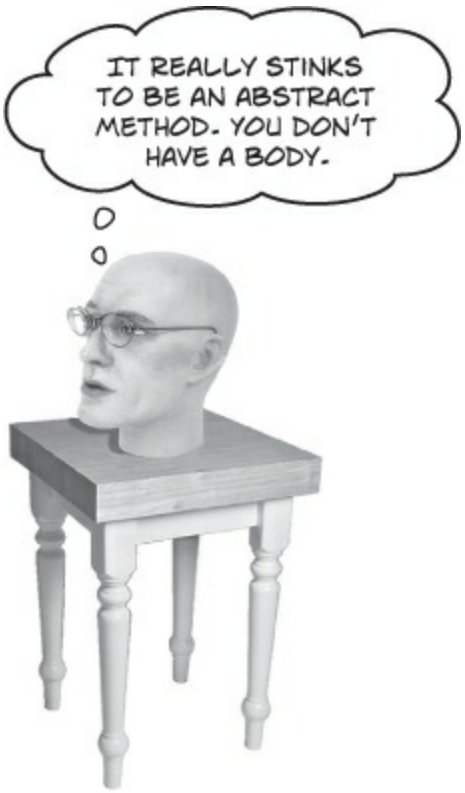
You know how an interface only has declarations for methods and properties, but it doesn't actually have any method bodies? That's because every method in an interface is an **abstract method**. So let's implement it! Once we do, the error will go away. Any time you extend an abstract class, you need to make sure that you override all of its abstract methods. Luckily, the IDE makes this job easier. Just type "public override" — as soon as you press space, the IDE will display a drop-down box with a list of any methods that you can override. Select the `SetMissionInfo()` method and fill it in:

Every method in an interface is automatically abstract, so you don't need to use the abstract keyword in an interface, just in an abstract class. Abstract classes can have abstract methods, but they can have concrete methods too.

```
abstract class PlanetMission {

    public abstract void SetMissionInfo(
        int milesToPlanet, int rocketFuelPerMile,
        long rocketSpeedMPH);

    // the rest of the class...
```

This abstract method is just like what you'd see in an interface—it doesn't have a body, but any class that inherits from PlanetMission has to implement the SetMissionInfo() method or the program won't compile.

If we add that method in and try to build the program, the IDE gives us an error:



So let's implement it! Once we do, the error will go away.

```
class Venus : PlanetMission {
    public Venus() {
        SetMissionInfo(40000000, 100000, 25000);
    }
    public override void SetMissionInfo(int milesToPlanet, int rocketFuelPerMile,
     long rocketSpeedMPH) {
        this.MilesToPlanet = milesToPlanet;
        this.RocketFuelPerMile = rocketFuelPerMile;
        this.RocketSpeedMPH = rocketSpeedMPH;
    }
}
```

When you inherit from an abstract class, you need to override all of its abstract methods.

The Mars class looks just like Venus, except with different numbers. What do you think about this class hierarchy? Does it really make sense to make `SetMissionInfo()` abstract? Should it be a concrete method in the `PlanetMission` class instead?

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. We did the first one for you. Don't forget to use a dashed line for implementing an interface and a solid line for inheriting from a class.

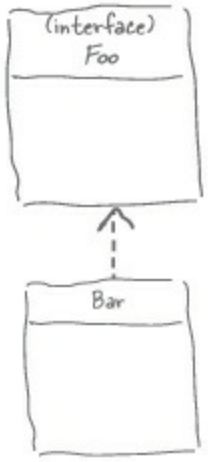
Given:

What's the picture ?

1)

```
interface Foo { }
class Bar : Foo { }
```

1)



2)

```
interface Vinn { }
abstract class Vout : Vinn { }
```

3)

```
abstract class Muffie : Whuffie { }
class Fluffie : Muffie { }
interface Whuffie { }
```

3)

4)

```
class Zoop { }
class Boop : Zoop { }
class Goop : Boop { }
```

4)

5)

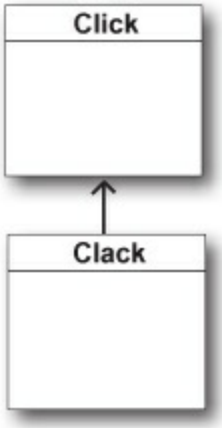
```
class Gamma : Delta, Epsilon { }
interface Epsilon { }
interface Beta { }
class Alpha : Gamma,Beta { }
class Delta { }
```

5)

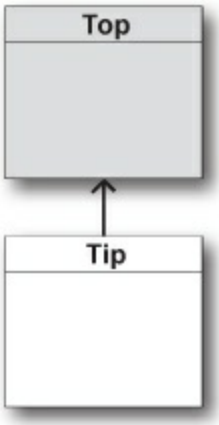
On the left you'll find sets of class diagrams. Your job is to turn these into valid C# declarations. We did number 1 for you.

Given:

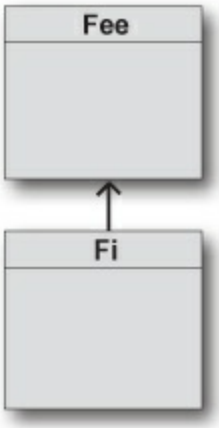
1



2

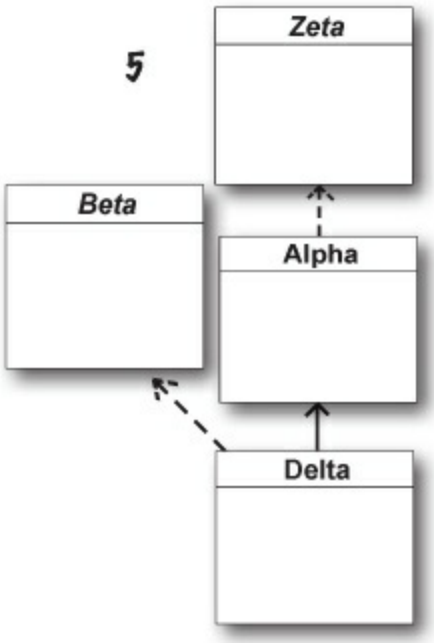


3



4

5



What's the declaration ?

1) public class Click { }
   public class Clack : Click { }

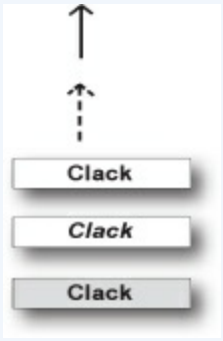
2)

3)

4)

5)

KEY



extends

implements

class

interface

abstract class

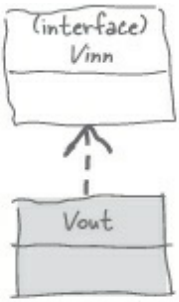
FIRESIDE CHATS

Tonight's talk: **An abstract class and an interface butt heads over the pressing question, "Who's more important?"**

Abstract Class	**Interface**
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.	
	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. I can have abstract methods or concrete ones. I can even have virtual methods if I want. Sure, I can't be instantiated — but then, neither can you. And I can do pretty much anything else a regular class does.	
	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.
You might be overstating your power a little bit.	
	You think that just because you can contain code, you're the greatest thing since sliced bread. But you can't change the fact that a program can only inherit from one class at a time. So you're a little limited. Sure, I can't include any code. But really, code is overrated.
That's exactly the kind of drivel I'd expect from an interface. Code is extremely important! It's what makes your programs run.	
	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 write an interface. Problem solved!
Yeah, sure, tell a coder he can't code.	
	What***ever***!

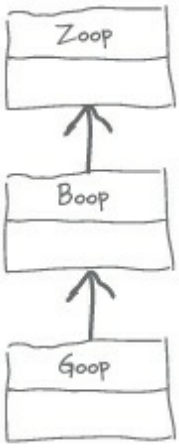
SHARPEN YOUR PENCIL SOLUTION

2)

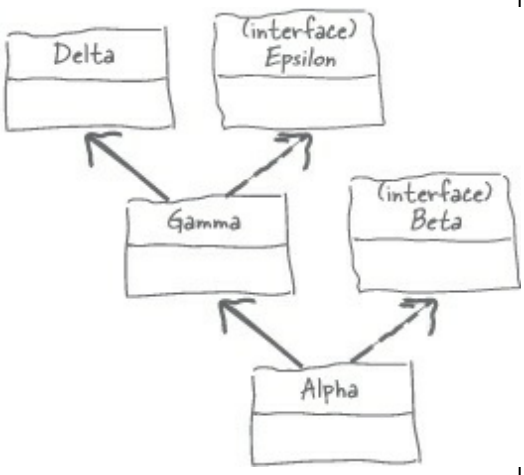


3)

4)



5)



What's the picture ?

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 { }
```

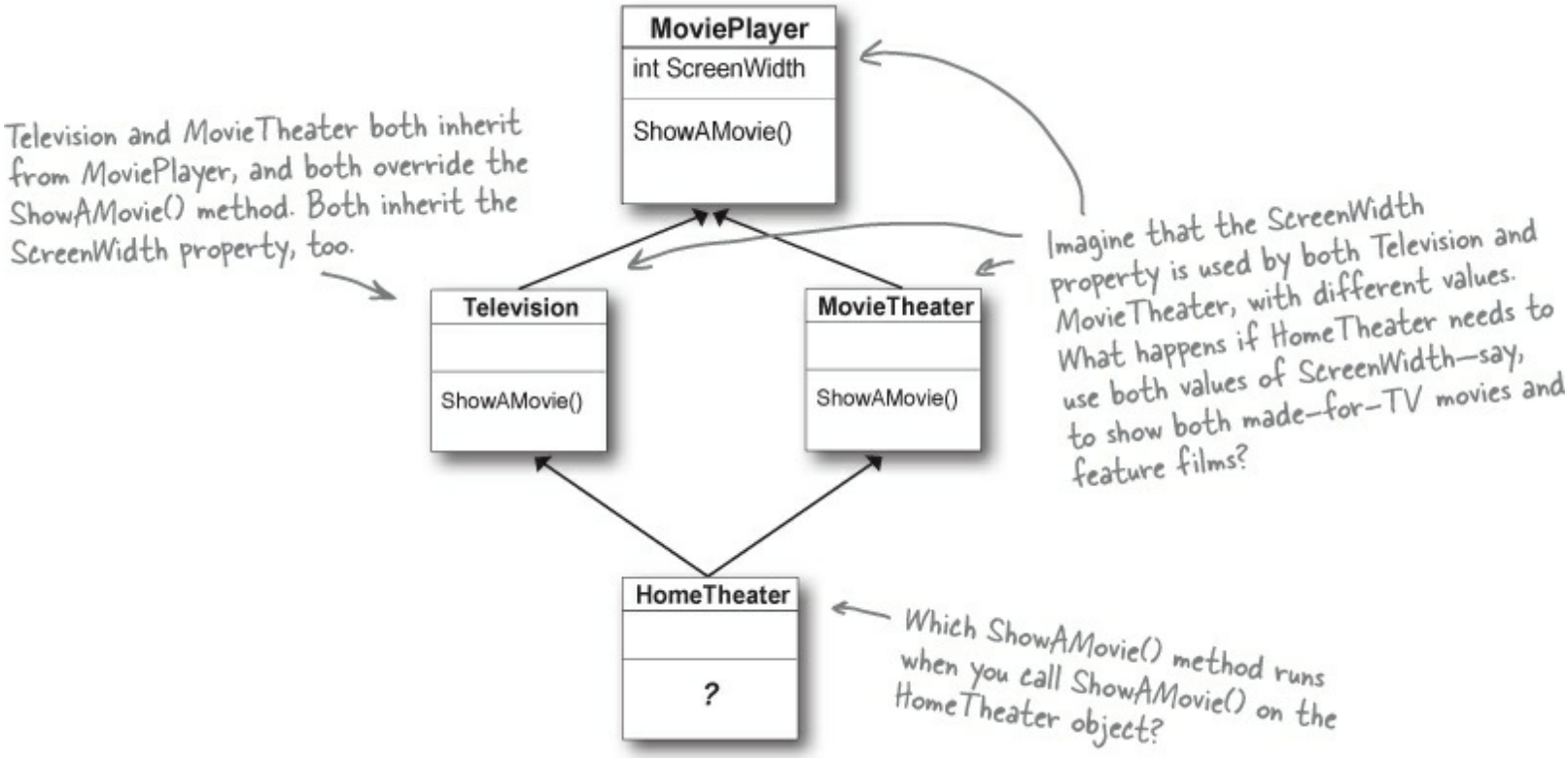
Delta inherits from Alpha and implements Beta.

What's the declaration ?

It's not a limitation, it's a protection.

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 mess that we like to call...

The Deadly Diamond of Death!



Avoid ambiguity!

A language that allows the Deadly Diamond of Death can lead to some pretty ugly situations, because you need special rules to deal with this kind of ambiguous situation...which means extra work for you when you're building your program! C# protects you from having to deal with this by giving you interfaces. If `Television` and `MovieTheater` are interfaces instead of classes, then the same `ShowAMovie()` method can satisfy both of them. All the interface cares about is that there's some method called `ShowAMovie()`.

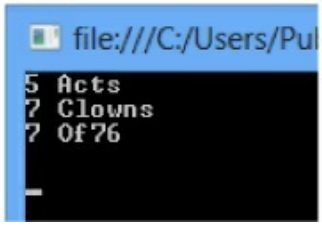
POOL PUZZLE

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 .....:.....{
    public virtual int Ear()
    {
        return 7;
    }
    public Picasso(string face)
    {
     ..... = face;
    }
    public virtual string Face {
     .....{ ..... ; }
    }
    string face;
}

class .....:.....{
    public Clowns() : base("Clowns") { }
}
```

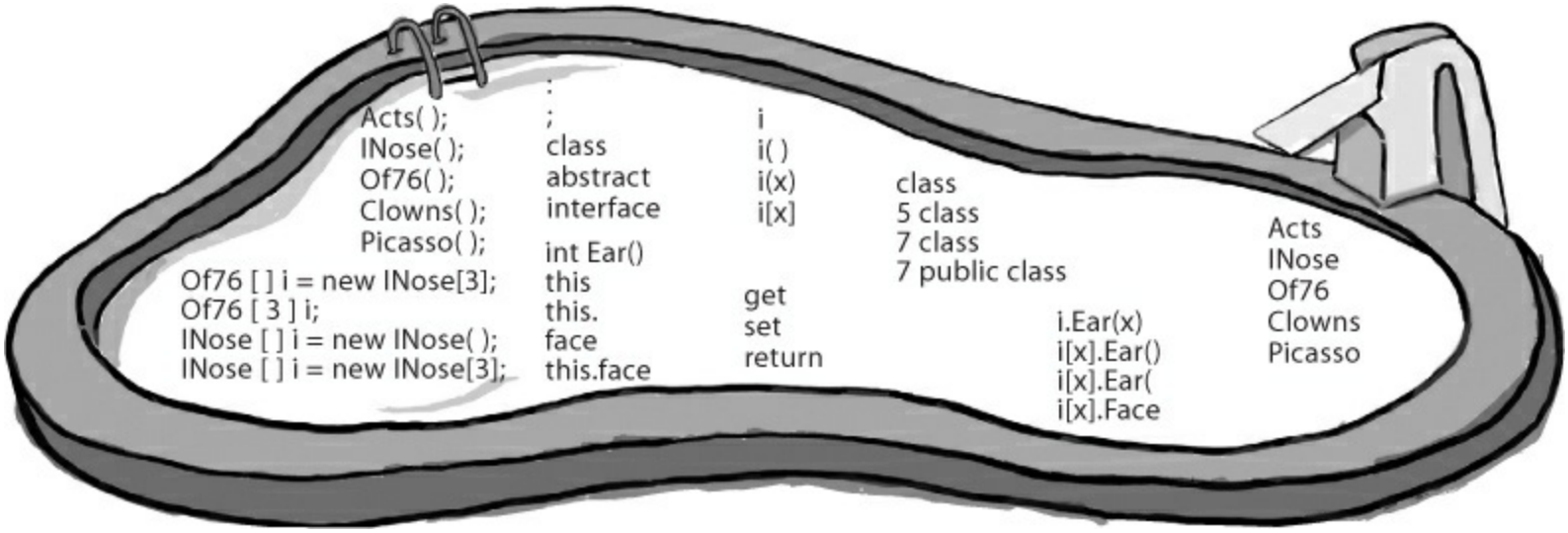
```
class .....:.....{
    public Acts() : base("Acts") { }
    public override ..... {
        return 5;
    }
}

class ..... : ..... {
    public override string Face {
      get { return "Of76"; }
    }
    public static void Main(string[] args) {
        string result = "";
        INose[] i = new INose[3];
        i[0] = new Acts();
        i[1] = new Clowns();
        i[2] = new Of76();
        for (int x = 0; x < 3; x++) {
            result += ( .....+ " "
                      + .....) + "\n";
        }
        Console.WriteLine(result);
        Console.ReadKey();
    }
}
```

Here's the entry point—this is a complete C# program.



Output

Note: each snippet from the pool can be used more than once!



Answers in Pool Puzzle Solution from Pool Puzzle.

You're an object-oriented programmer.

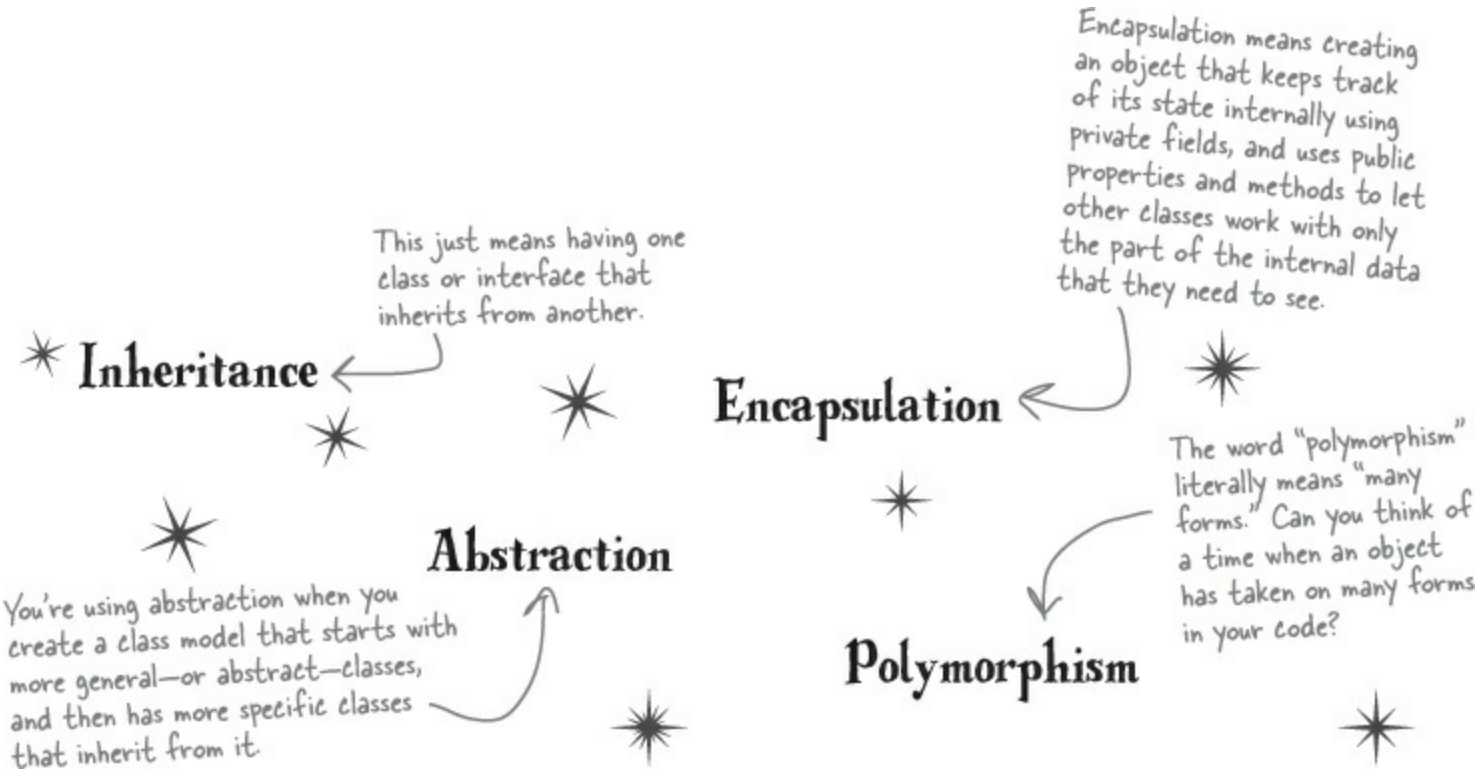
There's a name for what you've been doing. It's called **object-oriented programming**, or OOP. Before languages like C# came along, people didn't use objects and methods when writing their code. They just used functions (which is what they call methods in a non-OOP program) that were all in one place — 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#.

NOTE

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.

The four 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.



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.

> **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.**

Keep your eyes open for polymorphism in the next exercise!

You're about to do a really big exercise — the biggest one you've seen so far — and you'll be using a lot of polymorphism in it, so keep your eyes open. 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 you see similar code in what you write for the exercise, **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().

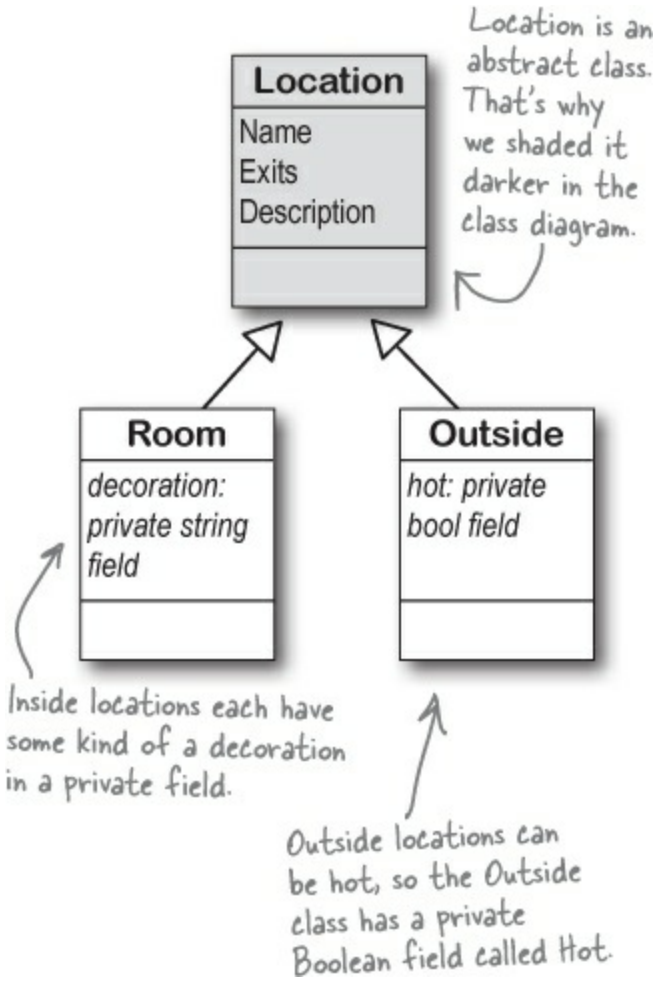
☐	Creating a reference variable whose type is an interface and pointing it to an object that implements that interface. `IStingPatrol defender = new StingPatrol();` This is upcasting, too!
☐	Downcasting using the `as` keyword.

LONG EXERCISE

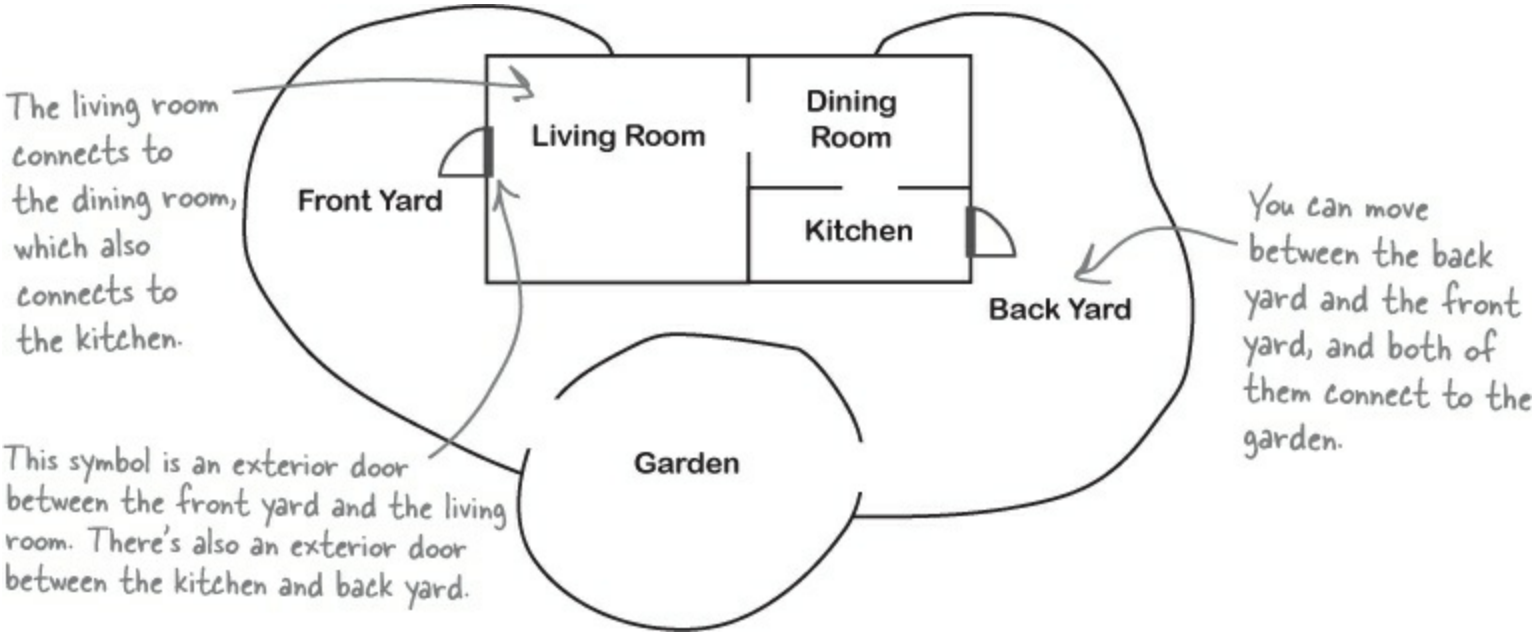
Let's build a house! Create a model of a house using classes to represent the rooms and locations, and an interface for any place that has a door.

- **❶ Start with this class model.**
  Every room or location in your house will be represented by its own object. The interior rooms all inherit from `Room`, and the outside places inherit from `Outside`, and both subclass the same base class, `Location`. The `Name` property is the name of the location ("Kitchen"). The `Exits` field is an array of `Location` objects that the current location connects to. So `diningRoom.Name` will be equal to `"Dining Room"`, and `diningRoom.Exits` will be equal to the array `{ LivingRoom, Kitchen }`.

  Create a Windows Forms Application project and add `Location`, `Room`, and `Outside` classes to it.

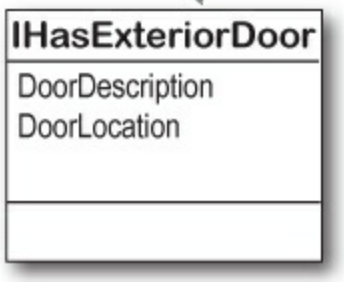


- **❷ You'll need the blueprint for the house.**
  This house has three rooms, a front yard, a back yard, and a garden. There are two doors: the front door connects the living room to the front yard, and the back door connects the kitchen to the back yard.



- **❸ Use the `IHasExteriorDoor` interface for rooms with an exterior door.**
  There are two exterior doors in the house, the front door and the back door. Every location that has one (the front yard, back yard, living room, and kitchen) should implement `IHasExteriorDoor`. The `DoorDescription` read-only property contains a description of the door (the front door is "an oak door with a brass knob," and the back door is "a screen door"). The `DoorLocation` property contains a reference to the `Location` where the door leads (`kitchen`).

All rooms have doors, but only a few rooms have an exterior door that leads inside or outside the house.



- **❹ Here's the `Location` class.**
  To get you started, here's the `Location` class:

```
abstract class Location {
    public Location(string name) {
        Name = name;
    }

    public Location[] Exits;

    public string Name { get; private set; }

    public virtual string Description {
        get {
            string description = "You're standing in the " + Name
                + ". You see exits to the following places: ";
            for (int i = 0; i < Exits.Length; i++) {
                description += " " + Exits[i].Name;
                if (i != Exits.Length - 1)
                    description += ",";
            }
            description += ".";
            return description;
        }
    }
}
```

The constructor sets the name field, which is the read-only Name property.

The public Exits field is an array of Location references that keeps track of all of the other places that this location connects to.

Description is a virtual property. You'll need to override it.

The base Description property returns a string that describes the room, including the name and a list of all of the locations it connects to (which it finds in the Exits[] field). Its subclasses will need to change the description slightly, so they'll override it.

The Room class will override and extend Description to add the decoration, and Outside will add the temperature.

Remember, Location is an abstract class—you can inherit from it and declare reference variables of type Location, but you can't instantiate it.

- **❺ Create the classes.**
  First create the `Room` and `Outside` classes based on the class model. Then create two more classes: `OutsideWithDoor`, which inherits from `Outside` and implements `IHasExteriorDoor`, and `RoomWithDoor`, which subclasses `Room` and implements `IHasExteriorDoor`.
  Here are the class declarations to give you a leg up:

Get the classes started now—we'll give you more details about them on the next page.

```
class OutsideWithDoor : Outside, IHasExteriorDoor
{
    // The DoorLocation property goes here
    // The read-only DoorDescription property goes here
}

class RoomWithDoor : Room, IHasExteriorDoor
{
    // The DoorLocation property goes here
    // The read-only DoorDescription property goes here
}
```

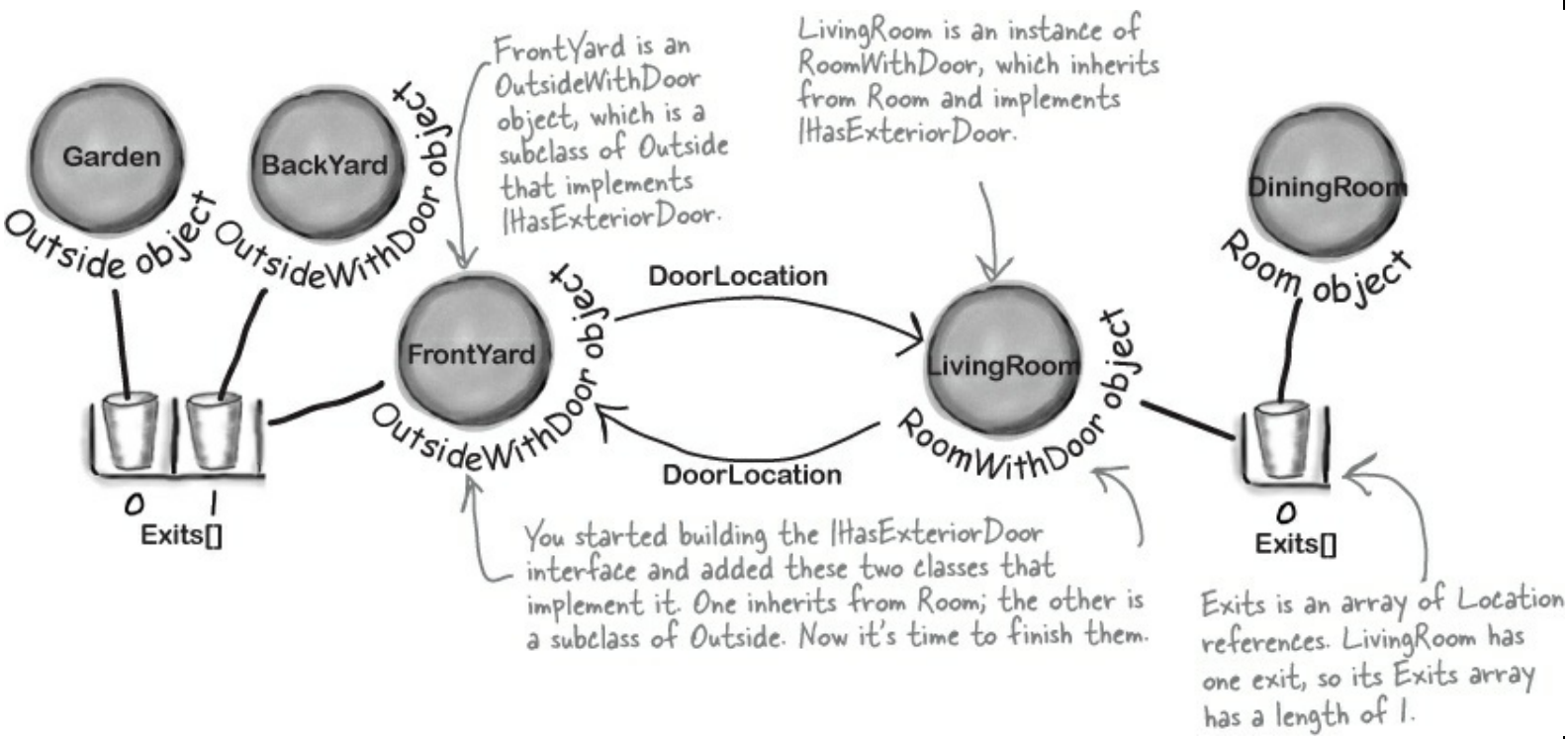
This one's going to be a pretty big exercise...but we promise it's a lot of fun! And you'll definitely know this stuff once you get through it.

We're not done yet — flip the page!

Now that you've got the class model, you can create the objects for all of the parts of the house, and add a form to explore it.

- **❻ Here's how your house objects work.**

Here's the architecture for two of your objects, `frontYard` and `livingRoom`. Since each of them has a door, they both need to be instances of a class that implements `IHasExteriorDoor`. The `DoorLocation` property keeps a reference to the location on the other side of the door.



- **❼ Finish building the classes, and instantiate their instances.**

You've got all the classes — now it's time to finish them and build your objects.

You'll need to make sure that the constructor for the `Outside` class sets the private `hot` fieldand overrides the `Description` property to add the text "It's very hot here." if `hot` is true. It's hot in the back yard, but not the front yard or garden.
The constructor for `Room` needs to set the `Decoration`, and should override the `Description` property to add, "You see *(the decoration)*." The living room has an antique carpet, the dining room has a crystal chandelier, and the kitchen has stainless steel appliances and a screen door that leads to the back yard.
Your form needs to create each of the objects and keep a reference to each one. So add a method to the form called `CreateObjects()` and call it from the form's constructor.

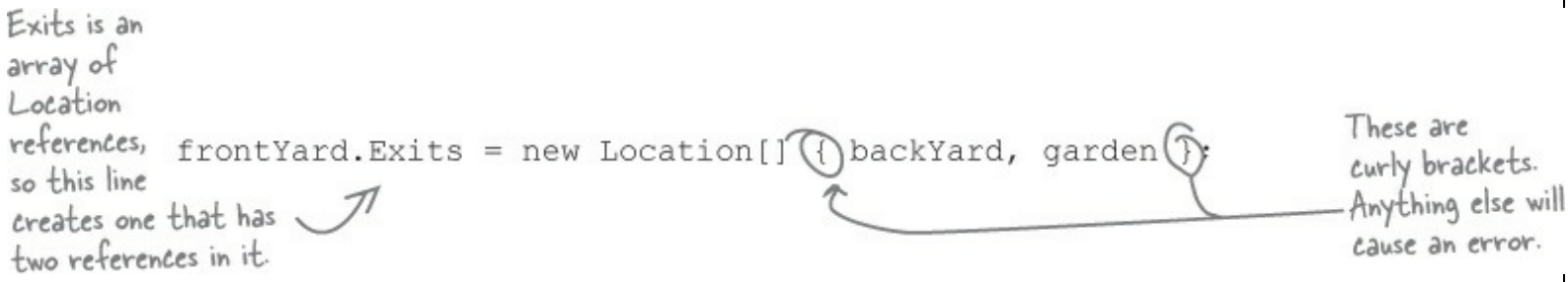
NOTE

— Every location will have its own field in the form class.

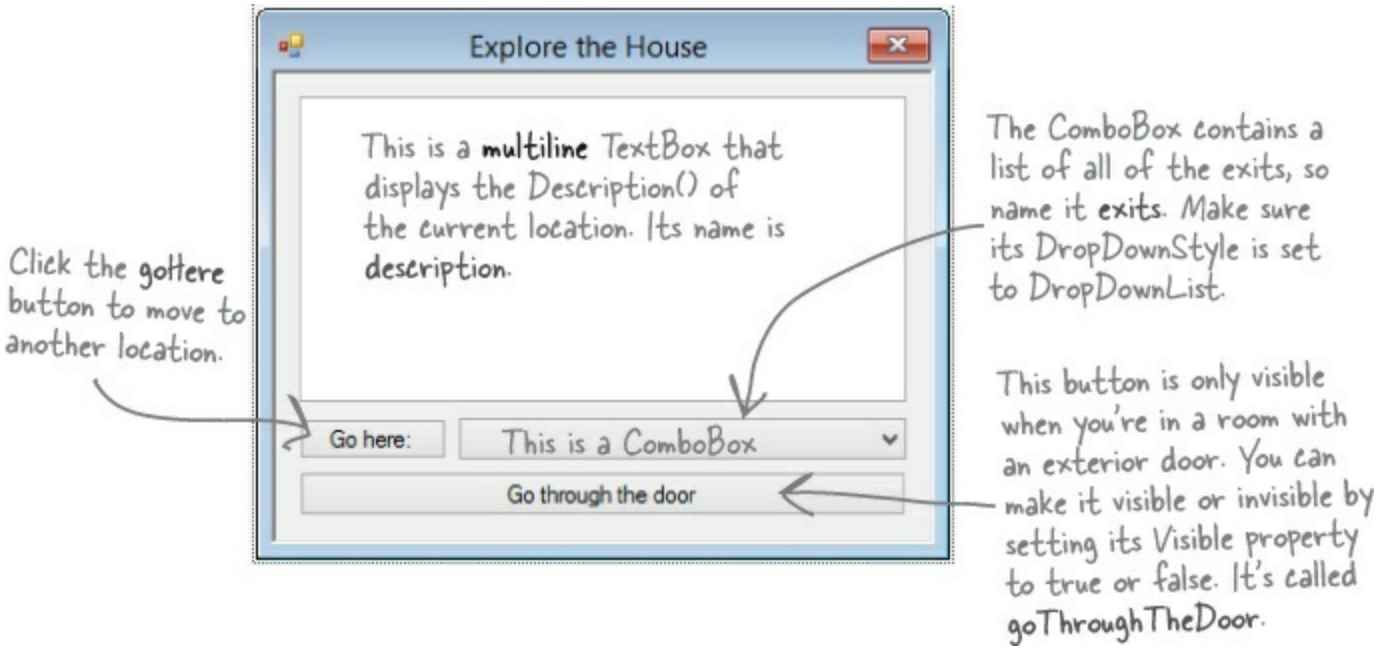
Instantiate each of the objects for the six locations in the house. Here's one of those lines:

```
RoomWithDoor livingRoom = new RoomWithDoor("Living Room",
     "an antique carpet" , "an oak door with a brass knob");
```

Your `CreateObjects()` method needs to populate the `Exits[]` field in each object:



- **❽ Build a form to explore the house.**
  Build a simple form to let you explore the house. It'll have a big multiline text box called `description` to show the description of the current room. A `ComboBox` called `exits` lists all of the exits in the current room. It's got two buttons: `goHere` moves to the room selected in the `ComboBox`, and `goThroughTheDoor` is only visible when there's an exterior door.



- **❾ Make the form work!**
  You've got all the pieces; now you just need to put them together.

  You'll need a field in your form called `currentLocation` to keep track of your current location.
  Add a `MoveToANewLocation()` method that has a `Location` as its parameter. This method should first set `currentLocation` to the new location. Then it'll clear the combo box using its `Items.Clear()` method, and then add the name of each location in the `Exits[ ]` array using the combo box's `Items.Add()` method. Finally, reset the combo box so it displays the first item in the list by setting its `SelectedIndex` property to zero.

NOTE

— Here's where you'll set up what populates the ComboBox.

Set the text box so that it has the description of the current location.
Use the **`is`** keyword to check if the current location has a door. If it does, make the "Go through the door" button visible using its `Visible` property. If not, make it invisible.
If the "Go here:" button is clicked, move to the location selected in the combo box.

NOTE

— Hint: when you choose an item in the combo box, its selected index in the combo box will be the same as the index of the corresponding location in the Exits[] array.

If the “Go through the door” button is clicked, move to the location that the door connects to.

NOTE

— Another hint: your form’s currentLocation field is a Location reference. So even though it’s pointing to an object that implements IHasExteriorDoor, you can’t just type “currentLocation.DoorLocation” because DoorLocation isn’t a field in Location. You’ll need to downcast if you want to get the door location out of the object.

LONG EXERCISE SOLUTION

Here’s the code to model the house. We used classes to represent the rooms and locations, and an interface for any place that has a door.

```
interface IHasExteriorDoor {
    string DoorDescription { get; }
    Location DoorLocation { get; set; }
}

class Room : Location {
    private string decoration;

    public Room(string name, string decoration)
        : base(name) {
        this.decoration = decoration;
    }

    public override string Description {
        get {
            return base.Description + " You see " + decoration + ".";
        }
    }
}

class RoomWithDoor : Room, IHasExteriorDoor {
    public RoomWithDoor(string name, string decoration, string doorDescription)
        : base(name, decoration)
    {
        DoorDescription = doorDescription;
    }

    public string DoorDescription { get; private set; }

    public Location DoorLocation { get; set; }
}
```

Here's the IHasExteriorDoor interface.

The Room class inherits from Location and adds a private field for the decoration. Its constructor sets the field.

The RoomWithDoor class inherits from Room and implements IHasExteriorDoor. It does everything that the room does, but it adds a description of the exterior door to the constructor. It also adds DoorLocation, a reference to the location that the door leads to. DoorDescription and DoorLocation are required by IHasExteriorDoor.

Did you use backing fields instead of automatic properties? That's a perfectly valid solution, too.

```
class Outside : Location {
    private bool hot;

    public Outside(string name, bool hot)
        : base(name)
    {
        this.hot = hot;
    }

    public override string Description {
        get {
            string newDescription = base.Description;
            if (hot)
                newDescription += " It's very hot.";
            return newDescription;
        }
    }
}

class OutsideWithDoor : Outside, IHasExteriorDoor {
    public OutsideWithDoor(string name, bool hot, string doorDescription)
        : base(name, hot)
    {
        this.DoorDescription = doorDescription;
    }

    public string DoorDescription { get; private set; }

    public Location DoorLocation { get; set; }

    public override string Description {
        get {
            return base.Description + " You see " + DoorDescription + ".";
        }
    }
}
```

Outside is a lot like Room—it inherits from Location, and adds a private field for the Hot property, which is used in the Description() method extended from the base class.

OutsideWithDoor inherits from Outside and implements IHasExteriorDoor, and it looks a lot like RoomWithDoor.

The base class's Description property fills in whether or not the location is hot. And that relies on the original Location class's Description property to add the main description and exits.

We're not done yet — flip the page!

Here's the code for the form. It's all in the *Form1.cs* file, inside the `Form1` declaration.

```
public partial class Form1 : Form
{
    Location currentLocation;

    RoomWithDoor livingRoom;
    Room diningRoom;
    RoomWithDoor kitchen;

    OutsideWithDoor frontYard;
    OutsideWithDoor backYard;
    Outside garden;

    public Form1() {
        InitializeComponent();
        CreateObjects();
        MoveToANewLocation(livingRoom);
    }

    private void CreateObjects() {
        livingRoom = new RoomWithDoor("Living Room", "an antique carpet",
                "an oak door with a brass knob");
        diningRoom = new Room("Dining Room", "a crystal chandelier");
        kitchen = new RoomWithDoor("Kitchen", "stainless steel appliances", "a screen door");

        frontYard = new OutsideWithDoor("Front Yard", false, "an oak door with a brass knob");
        backYard = new OutsideWithDoor("Back Yard", true, "a screen door");
        garden = new Outside("Garden", false);

        diningRoom.Exits = new Location[] { livingRoom, kitchen };
        livingRoom.Exits = new Location[] { diningRoom };
        kitchen.Exits = new Location[] { diningRoom };
        frontYard.Exits = new Location[] { backYard, garden };
        backYard.Exits = new Location[] { frontYard, garden };
        garden.Exits = new Location[] { backYard, frontYard };

        livingRoom.DoorLocation = frontYard;
        frontYard.DoorLocation = livingRoom;

        kitchen.DoorLocation = backYard;
        backYard.DoorLocation = kitchen;
    }
```

This is how the form keeps track of which room is being displayed.

The form uses these reference variables to keep track of each of the rooms in the house.

We made `Exits` a public string array field in the `Location` class. This is not a great example of encapsulation! Another object could easily modify the `Exits` array. In the next chapter, you'll learn about a better way to expose a sequence of strings or other objects.

The form's constructor creates the objects and then uses the MoveToANewLocation method.

When the form creates the objects, first it needs to instantiate the classes and pass the right information to each one's constructor.

Here's where we pass the door description to the OutsideWithDoor constructors.

Here's where the Exits[] array for each instance is populated. We need to wait to do this until after all the instances are created, because otherwise we wouldn't have anything to put into each array!

For the IHasExteriorDoor objects, we need to set their door locations.

```
    private void MoveToANewLocation(Location newLocation) {
        currentLocation = newLocation;

        exits.Items.Clear();
        for (int i = 0; i < currentLocation.Exits.Length; i++)
            exits.Items.Add(currentLocation.Exits[i].Name);
        exits.SelectedIndex = 0;

        description.Text = currentLocation.Description;

        if (currentLocation is IHasExteriorDoor)
            goThroughTheDoor.Visible = true;
        else
            goThroughTheDoor.Visible = false;
    }

    private void goHere_Click(object sender, EventArgs e) {
        MoveToANewLocation(currentLocation.Exits[exits.SelectedIndex]);
    }

    private void goThroughTheDoor_Click(object sender, EventArgs e) {
        IHasExteriorDoor hasDoor = currentLocation as IHasExteriorDoor;
        MoveToANewLocation(hasDoor.DoorLocation);
    }
}
```

The MoveToANewLocation() method displays a new location in the form.

First we need to clear the combo box, and then we can add each of the locations' names to it. Finally, we set its selected index (or which line is highlighted) to zero so it shows the first item in the list. Don't forget to set the ComboBox's DropDownStyle property to DropDownList—that way, the user won't be able to type anything into the combo box.

This makes the "Go through the door" button invisible if the current location doesn't implement IHasExteriorDoor.

When the user clicks the "Go here:" button, it moves to the location selected in the combo box.

We need to use the **as** keyword in order to downcast currentLocation to an IHasExteriorDoor so we can get access to the DoorLocation field.

But we're not done yet!

It's fine to create a model of a house, but wouldn't it be cool to turn it into a game? Let's do it! You'll play Hide and Seek against the computer. We'll need to add an `Opponent` class and have him hide in a room. And we'll need to make the house a lot bigger. Oh, and he'll need someplace to hide! We'll add a new interface so that some rooms can have a hiding place. Finally, we'll update the form to let you check the hiding places, and keep track of how many moves you've made trying to find your opponent. Sound fun? Definitely!

Let's get started!

*Here's the biggest challenge we've given you so far. Read the instructions carefully! It's **not cheating** to peek at the solution.*

EXERCISE

Time for hide and seek! Build on your original house program to add more rooms, hiding places, and an opponent who hides from you.

> **NOTE**
>
> Create a new project, and use the IDE's Add Existing Item feature to add the classes from the first part of the exercise.

- ❶ **Add an `IHidingPlace` interface.**
  We don't need to do anything fancy here. Any `Location` subclass that implements `IHidingPlace` has a place for the opponent to

hide. It just needs a string to store the name of the hiding place ("in the closet", "under the bed", etc.). Give it a get accessor, but no set accessor — we'll set this in the constructor, since once a room has a hiding place we won't ever need to change it.

We didn't give you a class diagram this time, so you should grab a piece of paper and draw it yourself. That will help you understand the program you need to build.

- **❷ Add classes that implement `IHidingPlace`.**
  You'll need two more classes: `OutsideWithHidingPlace` (which inherits from `Outside`) and `RoomWithHidingPlace` (which inherits from `Room`). Also, let's make any room with a door have a hiding place, so it'll have to inherit from `RoomWithHidingPlace` instead of `Room`.

  NOTE

  So every room with an exterior door will also have a hiding place: the kitchen has a cabinet, and the living room has a closet.

- **❸ Add a class for your opponent.**
  The `Opponent` object will find a random hiding place in the house, and it's your job to find him.

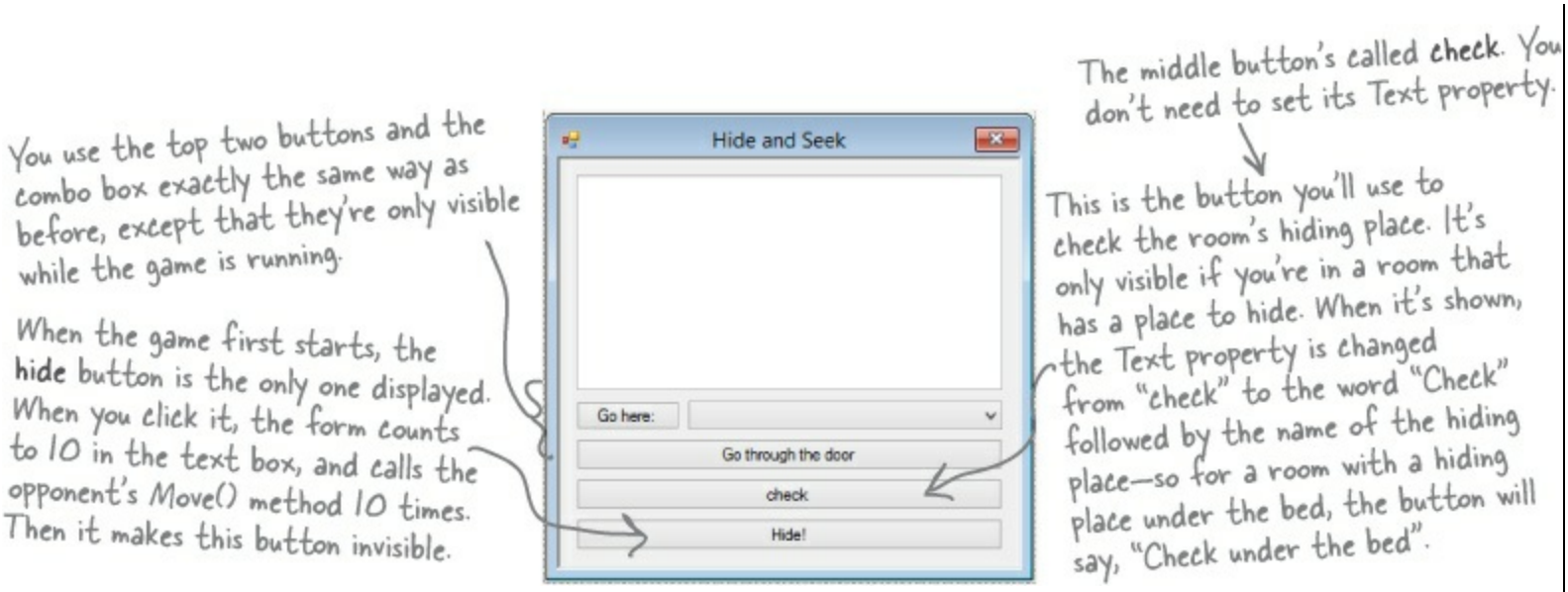
  He'll need a private `Location` field (`myLocation`) so he can keep track of where he is, and a private `Random` field (`random`) to use when he moves to a random hiding place.
  The constructor takes the starting location and sets `myLocation` to it, and sets `random` to a new instance of `Random`. He starts in the front yard (that'll be passed in by the form), and moves from hiding place to hiding place randomly. He moves 10 times when the game starts. When he encounters an exterior door, he flips a coin to figure out whether or not to go through it.
  Add a `Move()` method that moves the opponent from his current location to a new location. First, if he's in a room with a door, then he flips a coin to decide whether or not to go through the door, so if `random.Next(2)` is equal to 1, he goes through it. Then he chooses one of the exits from his current location at random and goes through it. If that location doesn't have a hiding place, then he'll do it again — he'll choose a random exit from his current location and go there, and he'll keep doing it over and over until he finds a place to hide.
  Add a `Check()` method that takes a location as a parameter and returns `true` if he's hiding in that location, or `false` otherwise.

- **❹ Add more rooms to the house.**
  Update your `CreateObjects()` method to add more rooms:

  Add **stairs** with a wooden bannister that connect the living room to the **upstairs hallway**, which has a picture of a dog and a closet to hide in.
  The upstairs hallway connects to three rooms: a **master bedroom** with a large bed, a **second bedroom** with a small bed, and a **bathroom** with a sink and a toilet. Someone could hide under the bed in either bedroom or in the shower.
  The front yard and back yard both connect to the **driveway**, where someone could hide in the garage. Also, someone could hide in the shed in the **garden**.

  Don't forget that there are many ways to solve any programming problem. If your solution is different than ours but it works, that's great!

- **❺ OK, it's time to update the form.**
  You'll need to add a few buttons to the form. And we'll get a little more intricate with making them visible or invisible, depending on the state of the game.



- **❻ Make the buttons work.**
  There are two new buttons to add to the form.

  The middle button checks the hiding place in the current room and is only visible when you're in a room with a place to hide using the opponent's `Check()` method. If you found him, then it resets the game.
  The bottom button is how you start the game. It counts to 10 by showing "1...", waiting 200 milliseconds, then showing "2...", then "3...", etc., in the text box. After each number, it tells the opponent to move by calling his `Move()` method. Then it shows, "Ready or not, here I come!" for half a second, and then the game starts.

> **NOTE**
>
> — Flip back to Chapter 2 for a refresher on DoEvents() and Sleep() — they'll come in handy.

- **❼ Add a method to redraw the form, and another one to reset the game.**
  Add a `RedrawForm()` method that puts the right text in the description text box, makes the buttons visible or invisible, and puts the correct label on the middle button. Then add a `ResetGame()` method that's run when you find your opponent. It resets the opponent object so that he starts in the front yard again — he'll hide when you click the "Hide!" button. It should leave the form with nothing but the text box and "Hide!" button visible. The text box should say where you found the opponent, and how many moves it took.
- **❽ Keep track of how many moves the player made.**
  Make sure the text box displays the number of times you checked a hiding place or moved between rooms. When you find the opponent, he should pop up a message box that says, "You found me in X moves!"
- **❾ Make it look right when you start the program.**
  When you first start the program, all you should see is an empty text box and the "Hide!" button. When you click the button, the fun begins!

EXERCISE SOLUTION

Build on your original house program to add more rooms, hiding places, and an opponent who hides from you.

Here's the new IHidingPlace interface. It just has one string field with a get accessor that returns the name of the hiding place.

```
interface IHidingPlace {
    string HidingPlaceName { get; }
}

class RoomWithHidingPlace : Room, IHidingPlace {
    public RoomWithHidingPlace(string name, string decoration, string hidingPlaceName)
        : base(name, decoration)
    {
        HidingPlaceName = hidingPlaceName;
    }
    public string HidingPlaceName { get; private set; }
    public override string Description {
        get {
            return base.Description + " Someone could hide " + HidingPlaceName + ".";
        }
    }
}
```

The RoomWithHidingPlace class inherits from Room and implements IHidingPlace by adding the HidingPlaceName property. The constructor sets its value.

```
class RoomWithDoor : RoomWithHidingPlace, IHasExteriorDoor {
    public RoomWithDoor(string name, string decoration,
                        string hidingPlaceName, string doorDescription)
        : base(name, decoration, hidingPlaceName)
    {
        DoorDescription = doorDescription;
    }

    public string DoorDescription { get; private set; }

    public Location DoorLocation { get; set; }
}
```

Since we decided every room with a door also needed a hiding place, we made RoomWithDoor inherit from RoomWithHidingPlace. The only change to it is that its constructor takes a hiding place name and sends it on to the RoomWithHidingPlace constructor.

You'll also need the OutsideWithDoor class, which is identical to the version from the "Explore the House" program.

```
class OutsideWithHidingPlace : Outside, IHidingPlace {
    public OutsideWithHidingPlace(string name, bool hot, string hidingPlaceName)
        : base(name, hot)
    {
       HidingPlaceName = hidingPlaceName;
    }

    public string HidingPlaceName { get; private set; }

    public override string Description {
        get {
            return base.Description + " Someone could hide " + HidingPlaceName + ".";
        }
    }
}
```

The OutsideWithHidingPlace class inherits from Outside and implements IHidingPlace just like RoomWithHidingPlace does.

```
class Opponent {
    private Random random;
    private Location myLocation;
    public Opponent(Location startingLocation) {
        myLocation = startingLocation;
        random = new Random();
    }
    public void Move() {
       bool hidden = false;
        while (!hidden) {
            if (myLocation is IHasExteriorDoor) {
                IHasExteriorDoor locationWithDoor =
                                myLocation as IHasExteriorDoor;
                if (random.Next(2) == 1)
                    myLocation = locationWithDoor.DoorLocation;
            }
            int rand = random.Next(myLocation.Exits.Length);
            myLocation = myLocation.Exits[rand];
            if (myLocation is IHidingPlace)
                hidden = true;
        }
    }
    public bool Check(Location locationToCheck) {
        if (locationToCheck != myLocation)
            return false;
        else
            return true;
    }
}
```

The Opponent class constructor takes a starting location. It creates a new instance of Random, which it uses to move randomly between rooms.

The Move() method first checks if the current room has a door using the **is** keyword—if so, it has a 50% chance of going through it. Then it moves to a random location, and keeps moving until it finds a hiding place.

The guts of the Move() method is this while loop. It keeps looping until the variable hidden is true—and it sets it to true when it finds a room with a hiding place.

The Check() method just checks the opponent's location against the location that was passed to it using a Location reference. If they point to the same object, then he's been found!

We're not done yet — flip the page!

Here's all the code for the form. The only things that stay the same are the `goHere_Click()` and `goThroughTheDoor_Click()` methods.

```
int Moves;

Location currentLocation;

RoomWithDoor livingRoom;
RoomWithHidingPlace diningRoom;
RoomWithDoor kitchen;
Room stairs;
RoomWithHidingPlace hallway;
RoomWithHidingPlace bathroom;
RoomWithHidingPlace masterBedroom;
RoomWithHidingPlace secondBedroom;

OutsideWithDoor frontYard;
OutsideWithDoor backYard;
OutsideWithHidingPlace garden;
OutsideWithHidingPlace driveway;

Opponent opponent;
```

Here are all the fields in the Form1 class. It uses them to keep track of the locations, the opponent, and the number of moves the player has made.

The Form1 constructor creates the objects, sets up the opponent, and then resets the game. We added a Boolean parameter to ResetGame() so that it only displays its message when you win, not when you first start up the program.

```
public Form1() {
    InitializeComponent();
    CreateObjects();
    opponent = new Opponent(frontYard);
    ResetGame(false);
}
private void MoveToANewLocation(Location newLocation) {
    Moves++;
    currentLocation = newLocation;
    RedrawForm();
}

private void RedrawForm() {
    exits.Items.Clear();
    for (int i = 0; i < currentLocation.Exits.Length; i++)
        exits.Items.Add(currentLocation.Exits[i].Name);
    exits.SelectedIndex = 0;
    description.Text = currentLocation.Description + "\r\n(move #" + Moves + ")";
    if (currentLocation is IHidingPlace) {
        IHidingPlace hidingPlace = currentLocation as IHidingPlace;
        check.Text = "Check " + hidingPlace.HidingPlaceName;
        check.Visible = true;
    }
    else
        check.Visible = false;
    if (currentLocation is IHasExteriorDoor)
        goThroughTheDoor.Visible = true;
    else
        goThroughTheDoor.Visible = false;
}
```

The MoveToANewLocation() method sets the new location and then redraws the form.

We need the hiding place name, but we've only got the CurrentLocation object, which doesn't have a HidingPlaceName property. So we can use **as** to copy the reference to an IHidingPlace variable.

RedrawForm() populates the combo box list, sets the text (adding the number of moves), and then makes the buttons visible or invisible depending on whether or not there's a door or the room has a hiding place.

Wow—you could add an entire wing onto the house just by adding a couple of lines! That's why well-encapsulated classes and objects are really useful.

```
private void CreateObjects() {
    livingRoom = new RoomWithDoor("Living Room", "an antique carpet",
              "inside the closet", "an oak door with a brass handle");
    diningRoom = new RoomWithHidingPlace("Dining Room", "a crystal chandelier",
               "in the tall armoire");
    kitchen = new RoomWithDoor("Kitchen", "stainless steel appliances",
              "in the cabinet", "a screen door");
    stairs = new Room("Stairs", "a wooden bannister");
    hallway = new RoomWithHidingPlace("Upstairs Hallway", "a picture of a dog",
              "in the closet");
    bathroom = new RoomWithHidingPlace("Bathroom", "a sink and a toilet",
              "in the shower");
    masterBedroom = new RoomWithHidingPlace("Master Bedroom", "a large bed",
              "under the bed");
    secondBedroom = new RoomWithHidingPlace("Second Bedroom", "a small bed",
              "under the bed");

    frontYard = new OutsideWithDoor("Front Yard", false, "a heavy-looking oak door");
    backYard = new OutsideWithDoor("Back Yard", true, "a screen door");
    garden = new OutsideWithHidingPlace("Garden", false, "inside the shed");
    driveway = new OutsideWithHidingPlace("Driveway", true, "in the garage");

    diningRoom.Exits = new Location[] { livingRoom, kitchen };
    livingRoom.Exits = new Location[] { diningRoom, stairs };
    kitchen.Exits = new Location[] { diningRoom };
    stairs.Exits = new Location[] { livingRoom, hallway };
    hallway.Exits = new Location[] { stairs, bathroom, masterBedroom, secondBedroom };
    bathroom.Exits = new Location[] { hallway };
    masterBedroom.Exits = new Location[] { hallway };
    secondBedroom.Exits = new Location[] { hallway };
    frontYard.Exits = new Location[] { backYard, garden, driveway };
    backYard.Exits = new Location[] { frontYard, garden, driveway };
    garden.Exits = new Location[] { backYard, frontYard };
    driveway.Exits = new Location[] { backYard, frontYard };

    livingRoom.DoorLocation = frontYard;
    frontYard.DoorLocation = livingRoom;

    kitchen.DoorLocation = backYard;
    backYard.DoorLocation = kitchen;
}
```

The new CreateObjects() method creates all the objects to build the house. It's a lot like the old one, but it has a whole lot more places to go.

We're still not done — flip the page!

Here's the rest of the code for the form. The gotHere and goThroughTheDoor button event handlers are identical to the ones in the first part of this exercise, so flip back a few pages to see them.

```
private void ResetGame(bool displayMessage) {
    if (displayMessage) {
        MessageBox.Show("You found me in " + Moves + " moves!");
        IHidingPlace foundLocation = currentLocation as IHidingPlace;
        description.Text = "You found your opponent in " + Moves
              + " moves! He was hiding " + foundLocation.HidingPlaceName + ".";
    }
    Moves = 0;
    hide.Visible = true;
    goHere.Visible = false;
    check.Visible = false;
    goThroughTheDoor.Visible = false;
    exits.Visible = false;
}

private void check_Click(object sender, EventArgs e) {
    Moves++;
    if (opponent.Check(currentLocation))
        ResetGame(true);
    else
        RedrawForm();
}

private void hide_Click(object sender, EventArgs e) {
    hide.Visible = false;

    for (int i = 1; i <= 10; i++) {
        opponent.Move();
        description.Text = i + "... ";
        Application.DoEvents();
        System.Threading.Thread.Sleep(200);
    }

    description.Text = "Ready or not, here I come!";
    Application.DoEvents();
    System.Threading.Thread.Sleep(500);

    goHere.Visible = true;
    exits.Visible = true;
    MoveToANewLocation(livingRoom);
}
```

The ResetGame() method resets the game. It displays the final message, then makes all the buttons except the "Hide!" one invisible.

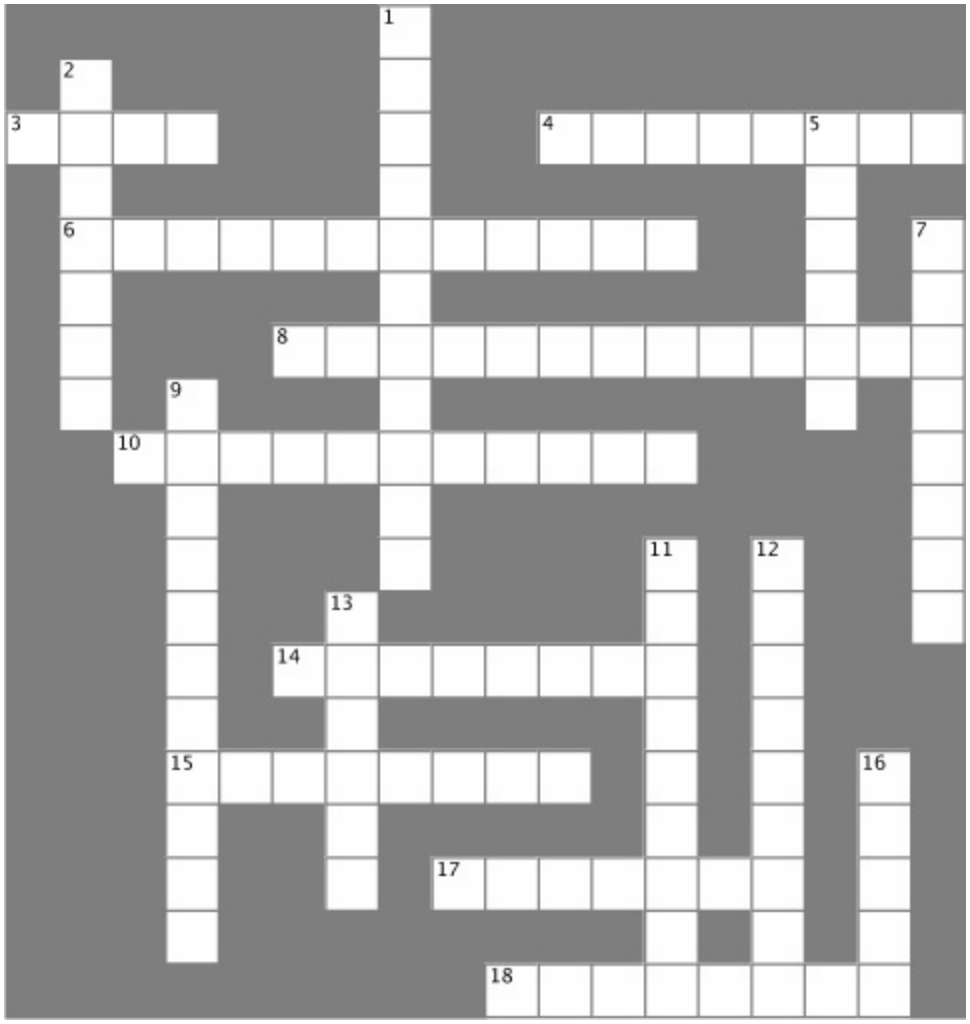
We want to display the name of the hiding place, but CurrentLocation is a Location reference, so it doesn't give us access to the HidingPlaceName field. Luckily, we can use the **as** keyword to downcast it to an IHidingPlace reference that points to the same object.

When you click the **check** button, it checks whether or not the opponent is hiding in the current room. If he is, it resets the game. If not, it redraws the form (to update the number of moves).

Remember DoEvents() from FlashyThing in Chapter 2? Without it, the text box doesn't refresh itself and the program looks frozen.

The **hide** button is the one that starts the game. The first thing it does is make itself invisible. Then it counts to 10 and tells the opponent to move. Finally, it makes the first button and the combo box visible, and then starts off the player in the living room. The MoveToANewLocation() method calls RedrawForm().

OOPCROSS



Across	Down
3. What an abstract method doesn't have 4. C# doesn't allow _____ inheritance 6. When you subclass to a method that expects its base class, you're using this OOP principle 8. The OOP principle where you hide private data and only expose those methods and fields that other classes need access to 10. One of the four principles of OOP that you implement using the colon operator 14. Every method in an interface is automatically _____ 15. If your class implements an interface that _____ from another interface, then you need to implement all of its members, too 17. An access modifier that's not valid for anything inside an interface 18. Object _____ Programming means creating programs that combine your data and code together into classes and objects	1. When you move common methods from specific classes to a more general class that they all inherit from, you're using this OOP principle 2. If a class that implements an interface doesn't implement all of its methods, getters, and setters, then the project won't _____ 5. Everything in an interface is automatically _____ 7. An abstract class can include both abstract and _____ methods 9. You can't _____ an abstract class 11. A class that implements this must include all of the methods, getters, and setters that it defines 12. What you do with an interface 13. The `is` keyword returns true if an _____ implements an interface 16. An interface can't technically include a _____, but it can define getters and setters that look just like one from the outside

POOL PUZZLE SOLUTION FROM 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.

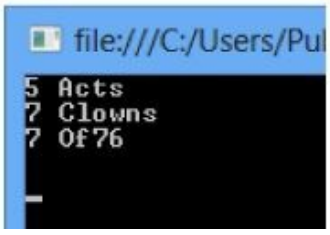
```
interface INose {
    int Ear() ;
    string Face { get; }
}

abstract class Picasso : INose {
    public virtual int Ear()
    {
        return 7;
    }
    public Picasso(string face)
    {
        this.face = face;
    }
    public virtual string Face {
        get { return face ; }
    }
    string face;
}

class Clowns : Picasso {
    public Clowns() : base("Clowns") { }
}
```

Properties can appear anywhere in the class! It's easier to read your code if they're at the top, but it's perfectly valid to have the face property at the bottom of the Picasso class.

Here's where the Acts class calls the constructor in Picasso, which it inherits from. It passes "Acts" into the constructor, which gets stored in the face property.

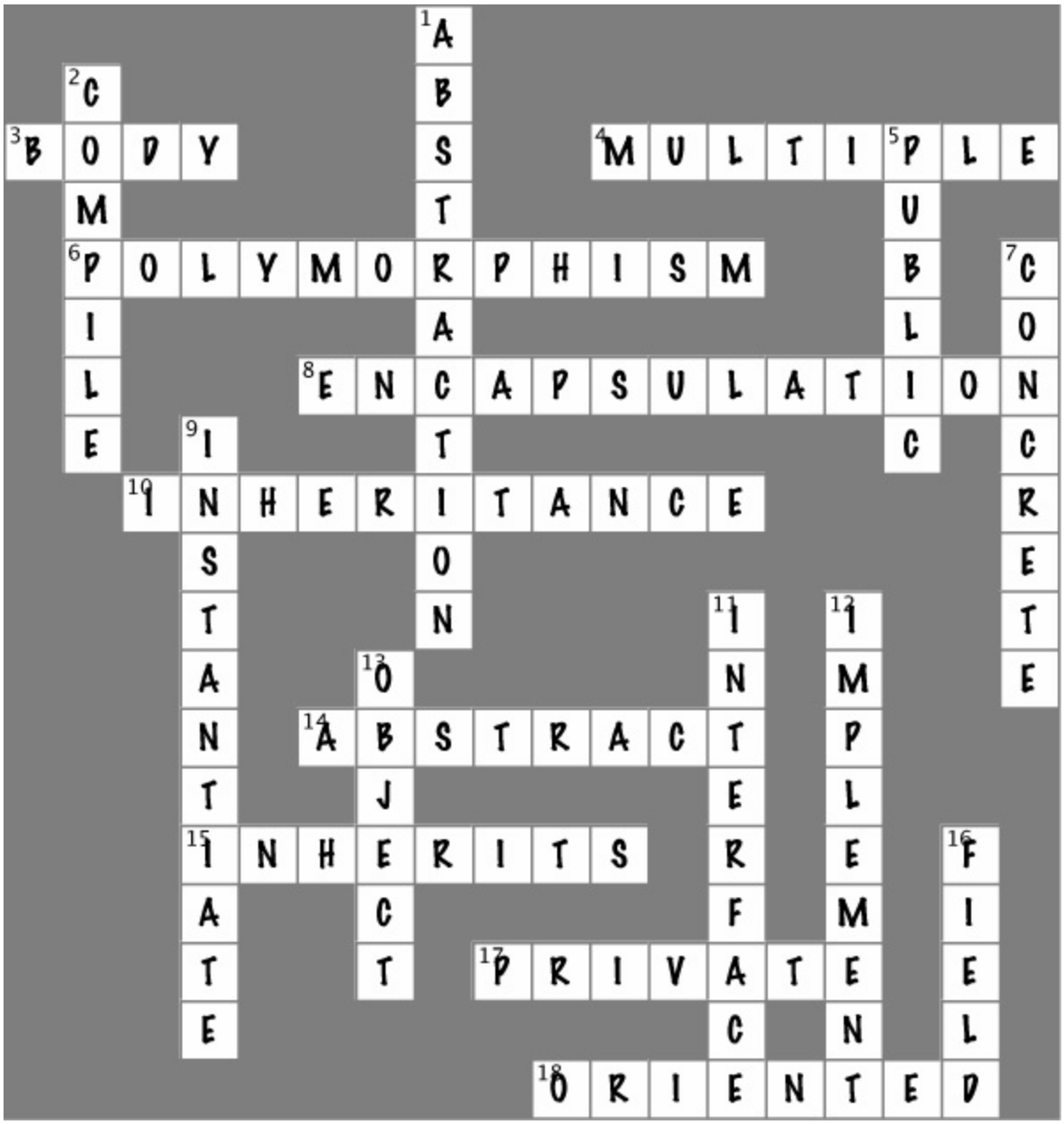
```
class Acts : Picasso {
    public Acts() : base("Acts") { }
    public override int Ear() {
        return 5;
    }
}

class Of76 : Clowns {
    public override string Face {
      get { return "Of76"; }
    }
    public static void Main(string[] args) {
        string result = "";
        INose[] i = new INose[3];
        i[0] = new Acts();
        i[1] = new Clowns();
        i[2] = new Of76();
        for (int x = 0; x < 3; x++) {
            result += ( i[x].Ear() + " "
                      + i[x].Face ) + "\n";
        }
        Console.WriteLine(result);
        Console.ReadKey();
    }
}
```

Face is a get accessor that returns the value of the face property. Both of them are defined in Picasso and inherited into the subclasses.



Output

OOPCROSS SOLUTION



Chapter 8. Enums and Collections: Storing lots of data

When it rains, it pours.

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

Suppose you have several worker bees, all represented by `Worker` classes. How would you write a constructor that took a job as a parameter? If you use a string for the job name, you might end up with code that looks like this:

Our bee management software kept track of each worker's job using a string like "Sting Patrol" or "Nectar Collector".

Our code would allow these values to be passed in a constructor even though the program only supports Sting Patrol, Nectar Collector, and other jobs that a bee does.

```
Worker buzz = new Worker("Attorney General");
Worker clover = new Worker("Dog Walker");
Worker gladys = new Worker("Newscaster");
```

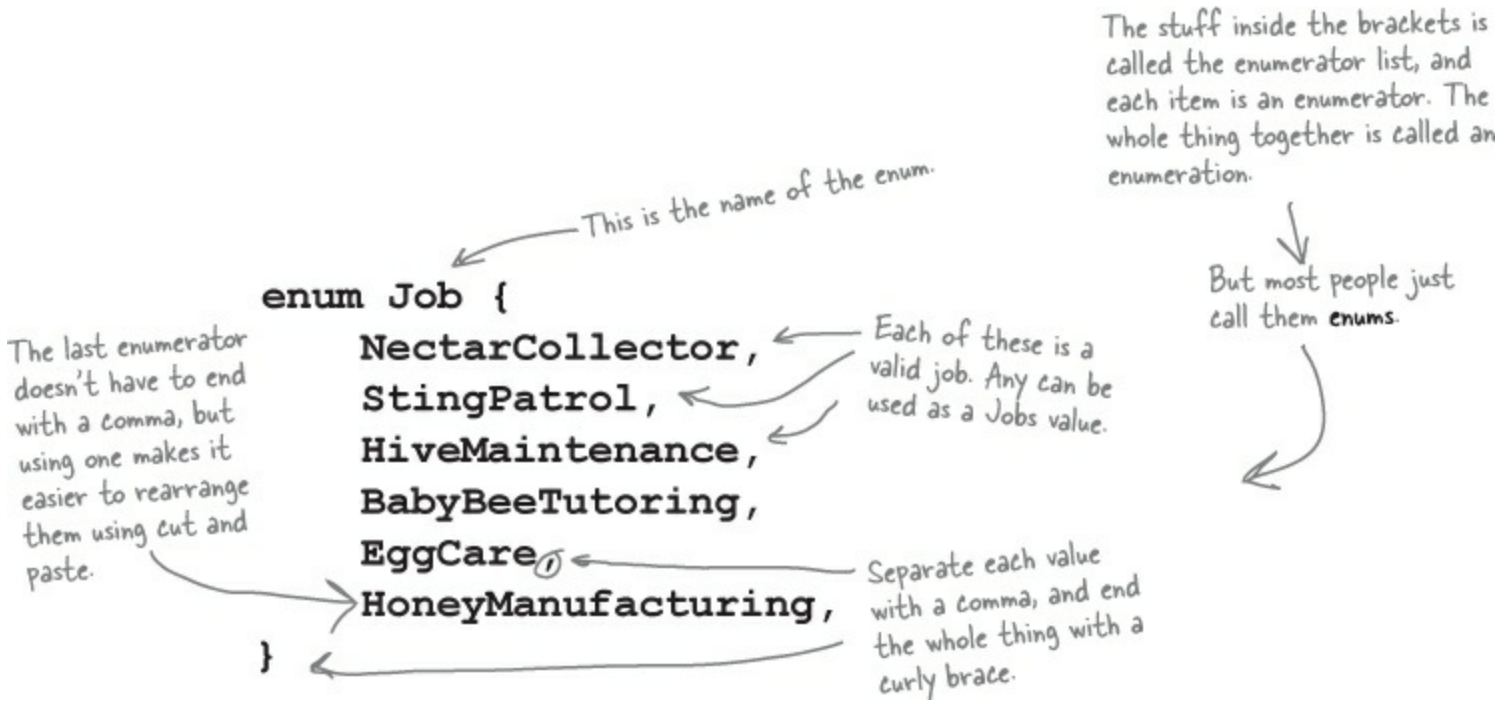
This code compiles, no problem. But these jobs don't make any sense for a bee. The Worker class really shouldn't allow these types as valid data.

You could probably add code to the `Worker` constructor to check each string and make sure it's a valid bee job. However, if you add new jobs that bees can do, you've got to change this code and recompile the `Worker` class. That's a pretty short-sighted solution. What if you have other classes that need to check for the types of worker bees they can be? Now you've got to duplicate code, and that's a bad path to go down.

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.

Enums let you work with a set of valid values

An **enum** is a data type that only allows certain values for that piece of data. So we could define an enum called `Job`, and define the allowed jobs:



Now, you can reference these with types like this:

But you can't just make up a new value for the enum! If you do, the program won't compile.

```
private void button1_Click(object sender EventArgs e)
{
      Worker buzz = new Worker(Job.AttorneyGeneral);
}
```

Here's the error you get from the compiler.

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,
}
```

These 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.

> **NOTE**
>
> 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`.

```
int value = (int)TrickScore.Fetch * 3;
MessageBox.Show(value.ToString());
TrickScore score = (TrickScore)value;
MessageBox.Show(score.ToString());
```

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.

Since Fetch has a value of 10, this statement sets the value to 30.

You can cast an int back to a TrickScore. Since value is equal to 30, score gets set to TrickScore.Speak. So when you call score.ToString(), it returns "Speak".

You can cast the enum as a number and do calculations with it, or you can use the `ToString()` method to treat the name as a string. 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 `:` operator, like this:

```
enum TrickScore : long {
        Sit = 7,
        Beg = 2500000000025
}
```

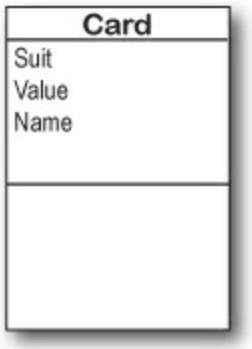
This tells the compiler to treat values in the TrickScore enum as longs, not ints.

If you tried to compile this code without specifying long as the type, you'd get this message:

```
Cannot implicitly convert type 'long' to 'int'.
```

Use what you've learned about enums to build a class that holds a playing card.

EXERCISE



- **❶ CREATE A NEW 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`).
- **❷ USE TWO ENUMS TO DEFINE THE SUITS AND VALUES.**
  Use the familiar Add→Class feature in the IDE to add them, replacing the word `class` with `enum` in the newly added files. Make sure that `(int)Suits.Spades` is equal to 0, followed by `Clubs` (equal to 1), `Diamonds` (2), and `Hearts` (3). 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.
- **❸ ADD A PROPERTY FOR THE NAME OF THE CARD.**
  `Name` should be a read-only property. The get accessor should return a string that describes the card. This code will run in a form that calls the `Name` property from the `card` class and displays it:

```
Card card = new Card(Suits.Spades, Values.Ace);
string cardName = card.Name;
```

To make this work, your Card class will need a constructor that takes two parameters.

  The value of `cardName` should be `Ace of Spades`.
- **❹ ADD A FORM BUTTON THAT POPS UP 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**. More on that later....

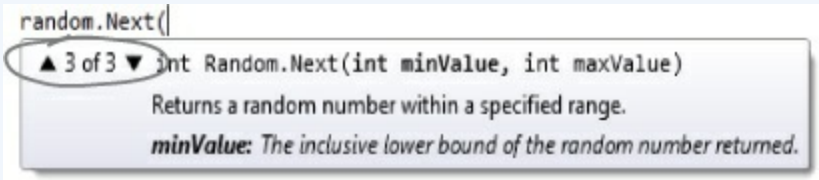
```
Random random = new Random();
int numberBetween0and3 = random.Next(4);
int numberBetween1and13 = random.Next(1, 14);
int anyRandomInteger = random.Next();
```

This tells Random to return a value at least 1 but under 14.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Hold on a second. When I was typing in that code, I noticed that an IntelliSense window popped up that said something about "3 of 3" when I used that `Random.Next()` method. What was that about?**

A: **A:** What you saw was a method that was **overloaded**. When a class has a method that you can call more than one way, it's called overloading. When you're using a class with an overloaded method, 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 where limiting values is important. Nobody wants to turn over their cards and be faced with a Joker of Clubs, or a 13 of Hearts. Here's how we wrote the `Card` class.

```
enum Suits {
    Spades,
    Clubs,
    Diamonds,
    Hearts
}

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
}

class Card {
    public Suits Suit { get; set; }
    public Values Value { get; set; }

    public Card(Suits suit, Values value) {
        this.Suit = suit;
        this.Value = value;
    }
    public string Name {
        get { return Value.ToString() + " of " + Suit.ToString(); }
    }
}

Random random = new Random();
private void button1_Click(object sender, EventArgs e) {
    Card card = new Card((Suits)random.Next(4), (Values)random.Next(1, 14));
    MessageBox.Show(card.Name);
}
```

When you don't specify values, the first item in the list is equal to 0, the second is 1, the third is 2, etc.

Here's where we set the value of Values.Ace to 1.

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?

The Card class has a Suit property of type Suits, and a Value property of type Values.

The get accessor for the Name property can take advantage of the way an enum's ToString() method returns its name converted to a string.

Here's where we use the overloaded Random.Next() method to generate a random number that we cast to the enum.

Here's the code for the button that pops up the name of a random card.

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.

This array declaration would continue all the way through the deck. It's just abbreviated here to save space.

```
class Deck {
    private Card[] cards =  {
        new Card(Suits.Spades, Values.Ace),
        new Card(Suits.Spades, Values.Two),
        new Card(Suits.Spades, Values.Three),
        // ...
        new Card(Suits.Diamonds, Values.Queen),
        new Card(Suits.Diamonds, Values.King),
    };

    public void PrintCards() {
        for (int i = 0; i < cards.Length; i++)
            Console.WriteLine(cards[i].Name());
    }
}
```

...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.

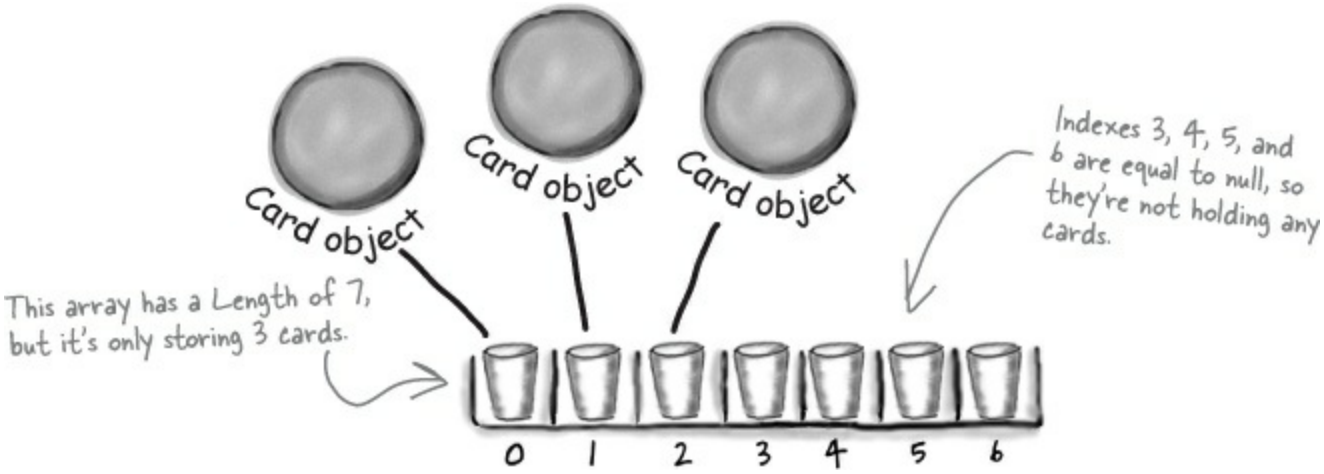
BRAIN POWER

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? How would you add a card to the deck?

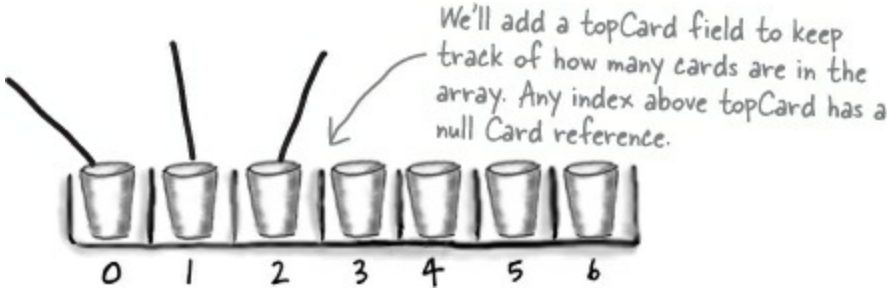
Arrays are hard to work with

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.

- ❶ 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:



- ❷ 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.



- ❸ 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!

> **NOTE**
>
> There's actually an Array.Resize() method built into the .NET Framework that does exactly that.

Lists make it easy to store collections of...anything

The .NET Framework has 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:

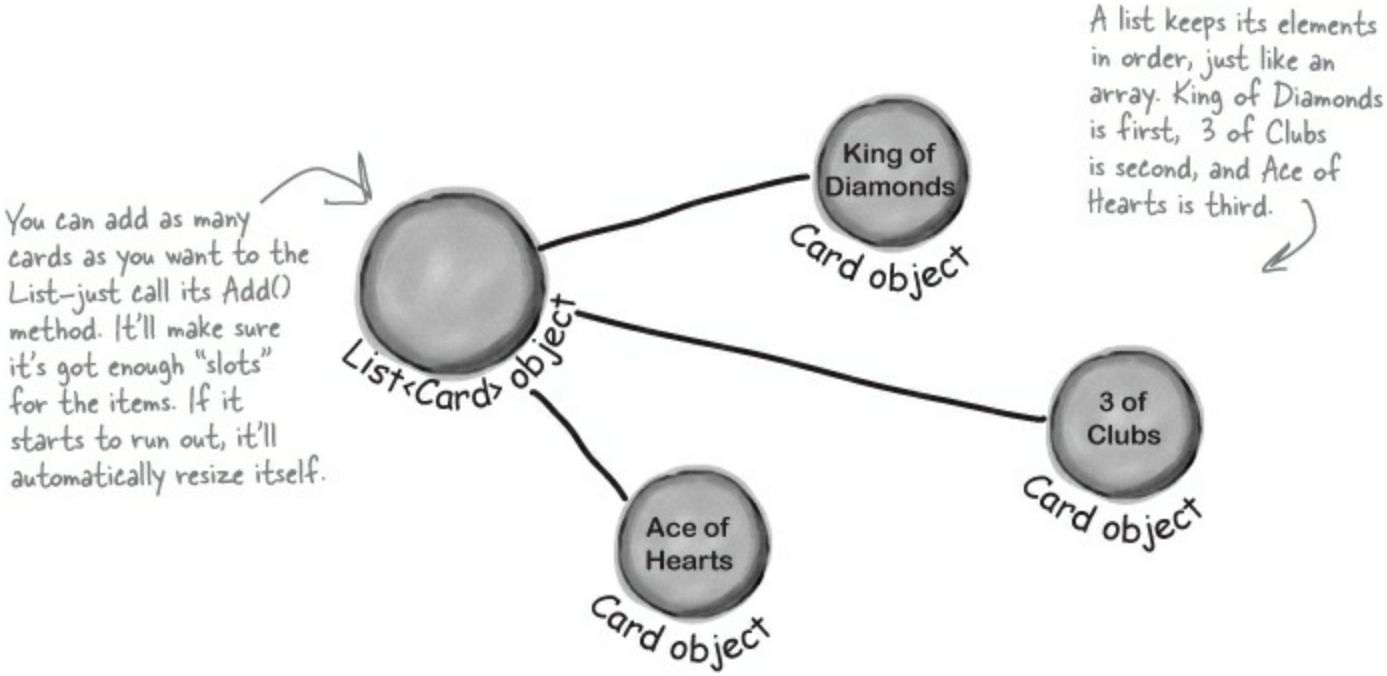
- Black **First you create a new instance of `List<T>`.**
  Every array has a type — you don't just have an array, you have an `int` array, a `Card` array, etc. Lists are the same way. You need to specify the type of object or value that the list will hold by putting it in angle brackets <> when you use the `new` keyword to create it.

```
List<Card> cards = new List<Card>();
```

- --------- **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>`).

```
cards.Add(new Card(Suits.Diamonds, Values.King));
cards.Add(new Card(Suits.Clubs, Values.Three));
cards.Add(new Card(Suits.Hearts, Values.Ace));
```

> **NOTE**
>
> Which means they're assignable to the type: interfaces, abstract classes, base classes, etc.



RELAX

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>!

The `<T>` at the end of `List<T>` means it's *generic*.

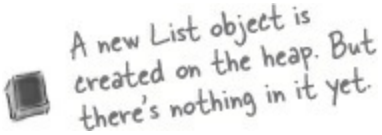
The T gets replaced with a type — so `List<int>` just means a `List` of `int`s. You'll get plenty of practice with generics over the next few pages.

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>`.

- Black **You can make one.**

```
List<Egg> myCarton = new List<Egg>();
```

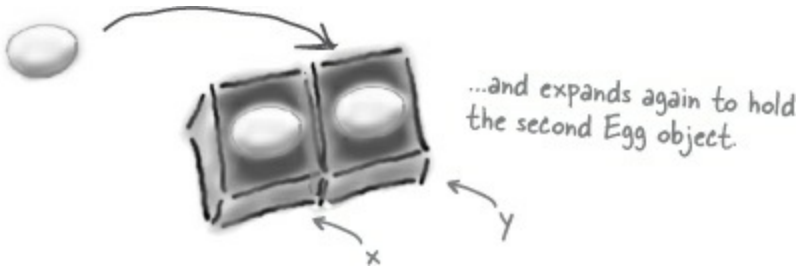


- **--------- Add something to it.**

```
Egg x = new Egg();
myCarton.Add(x);
```

- **❶ Add something else to it.**

```
Egg y = new Egg();
myCarton.Add(y);
```



- **❷ Find out how many things are in it.**

```
int theSize = myCarton.Count;
```

- **❸ Find out if it has something in particular in it.**

> **NOTE**
>
> Now you can search for any Egg inside the list. This would definitely come back true.

```
bool isIn = myCarton.Contains(x);
```

- **❹ Figure out where that thing is.**

```
int idx = myCarton.IndexOf(y);
```

The index for x would be 0 and the index for y would be 1.

- **❺ Take something out of it.**

```
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.

We filled in a couple for you....

List	Regular array
`List<String> myList = new List <String>();`	`String [] myList = new String[2];`
`String a = "Yay!";`	`String a = "Yay!";`
`myList.Add(a);`	
`String b = "Bummer";`	`String b = "Bummer";`
`myList.Add(b);`	
`int theSize = myList.Count;`	
`Guy o = guys[1];`	
`bool isIn = 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
`List<String> myList = new List <String>();`	`String[] myList = new String[2];`
`String a = "Yay!"`	`String a = "Yay!";`
`myList.Add(a);`	`myList[0] = a;`
`String b = "Bummer";`	`String b = "Bummer";`
`myList.Add(b);`	`myList[1] = b;`
`int theSize = myList.Count;`	`int theSize = myList.Length;`
`Guy o = guys[1];`	`Guy o = guys[1];`

`bool isIn = myList.Contains(b);`	`bool isIn = false;`
	`for (int i = 0; i < myList.Length; i++) {`
	`if (b == myList[i]) {`
	`isIn = true;`
	`}`
	`}`

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.

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 .NET Framework does have an Array class, which makes some of these things a little easier to do, but we're concentrating on List objects because they're a lot easier to use.

Lists shrink and grow dynamically

The great thing about a `List` is that you don't need to know how long it'll be when you create it. A `List` automatically grows and shrinks to fit its contents. Here's an example of a few of the methods that make working with `List`s a lot easier than arrays. **Create a new Console Application** and add this code to the `Main()` method. It *won't* print anything — **use the debugger** to step through the code and see what's going on.

DO THIS!

We're declaring a List of Shoe objects called ShoeCloset.

```
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" });

int numberOfShoes = shoeCloset.Count;
foreach (Shoe shoe in shoeCloset) {
    shoe.Style = Style.Flipflops;
    shoe.Color = "Orange";
}

shoeCloset.RemoveAt(4);

Shoe thirdShoe = shoeCloset[2];
Shoe secondShoe = shoeCloset[1];
shoeCloset.Clear();

shoeCloset.Add(thirdShoe);
if (shoeCloset.Contains(secondShoe))
    Console.WriteLine("That's surprising.");
```

You can use a new statement inside the List.Add() method.

This returns the total number of Shoe objects in the List.

This foreach loop goes through each of the shoes in the closet.

The Remove() method will remove the object by its reference; RemoveAt() does it by index number.

The Clear() method removes all of the objects in a List.

We saved references to two shoes before we cleared the list. We added one back, but the other's still missing.

This line will never run, because Contains() will return false. We only added thirdShoe into the cleared list, not secondShoe.

`foreach` is a special kind of loop for `List`s. It will execute a statement for each object in the `List`. This loop creates an identifier called `shoe`. As the loop goes through the items, it sets `shoe` equal to the first item in the list, then the second, then the third, until the loop is done.

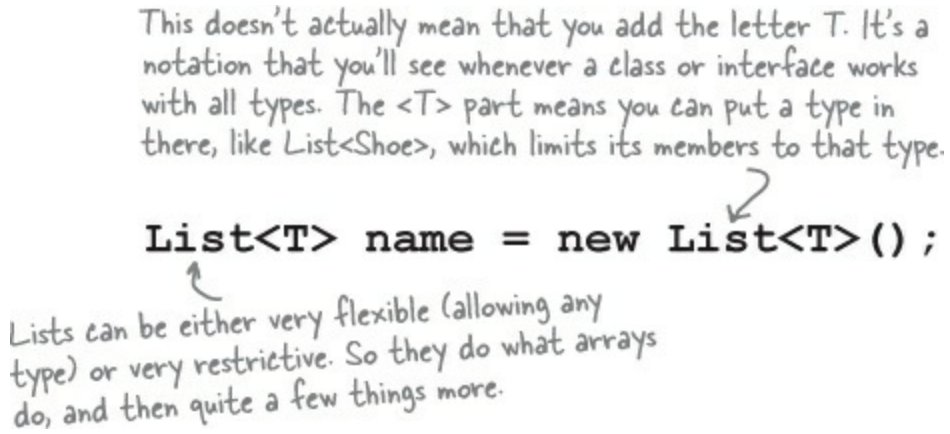
foreach loops work on arrays, too! In fact, they work on any collection.

Here's the Shoe class we're using, and the Style enum it uses.

```
class Shoe {
    public Style Style;
    public string Color;
}

enum Style {
    Sneakers,
    Loafers,
    Sandals,
    Flipflops,
    Wingtips,
    Clogs,
}
```

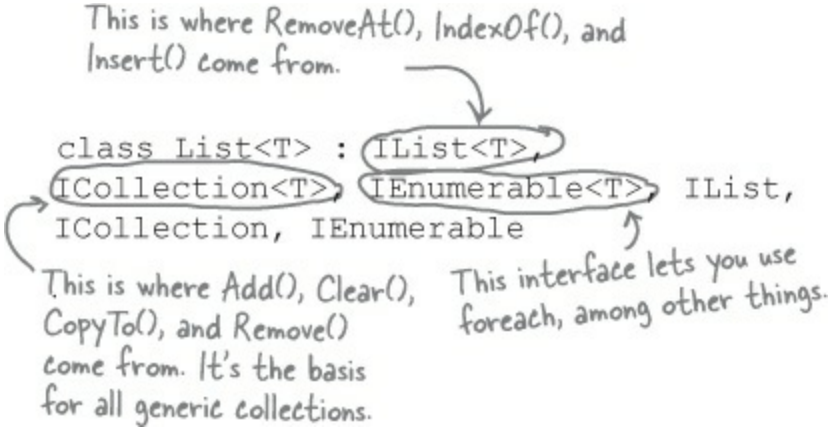
Generics can store <u>any</u> type

You've already seen that a `List` can store strings or `Shoe`s. You could also make `List`s 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 `List`s easy — once you've created your list, you always know the type of data that's inside it.



The .NET Framework comes with some generic interfaces that let the collections you're building work with any and all types. The `List` class implement those interfaces, and that's why you could create a `List` of integers and work with it in pretty much the same way that you would work with a `List` of `Shoe` objects.

Check it out for yourself. Type the word **`List`** into the IDE, and then right-click on it and select Go To Definition. That will take you to the declaration for the `List` class. It implements a few interfaces:



BULLET POINTS

- **`List`** is a class in the .NET Framework.
- 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 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 where something is (and if it is) in a `List`, use **`IndexOf()`**.
- To get the number of elements in a `List`, use the **`Count`** property.
- You can use the **`Contains()`** method to find out if a particular object is in a `List`.
- **`foreach`** is a special kind of loop that will iterate through all of the elements in a `List` and execute code on it. The syntax for a **`foreach`** loop is **`foreach (string s in StringList)`.** You don't have to tell the **`foreach`** loop to increment by one; it will go through the entire `List` all on its own.

WATCH IT!

You can't modify a collection while you're using `foreach` to iterate through it!

If you do, it will cause an error. Luckily, you can always make a copy of it. Every `IEnumerable` has a `ToList()` method that you can use to make a copy of it to safely iterate through.

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?

```
private void button1_Click(object sender,
EventArgs e){



}
```

```
a.RemoveAt(2);
```

```
List<string> a = new List<string>();
```

```
public void printL (List<string> a){
```

```
if (a.Contains("two")) {
    a.Add(twopointtwo);
}
```

```
a.Add(zilch);
a.Add(first);
a.Add(second);
a.Add(third);
```

```
}
```

```
string result = "";
```

```
if (a.Contains("three")){
      a.Add("four");
    }
```

```
foreach (string element in a)
{
    result += "\n" + element;
}
```

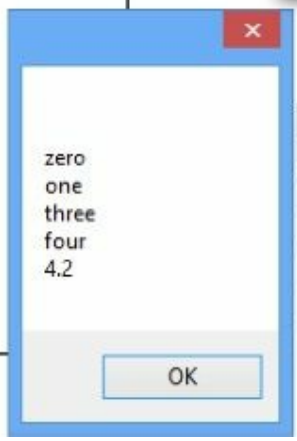
```
MessageBox.Show(result);
```

```
}
```

```
if (a.IndexOf("four") != 4) {
      a.Add(fourth);
}
```

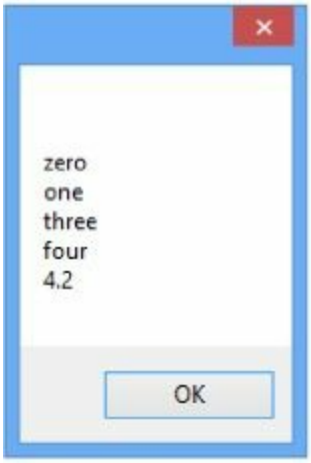
```
printL(a);
```

```
string zilch = "zero";
string first =  "one";
string second = "two";
string third = "three";
string fourth = "4.2";
string twopointtwo = "2.2";
```



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 way too easy. Just don't use cryptic names like `printL()` in real life!




```
private void button1_Click(object sender, EventArgs e)
{
    List<string> a = new List<string>();
    string zilch = "zero";
    string first =  "one";
    string second = "two";
    string third = "three";
    string fourth = "4.2";
    string twopointtwo = "2.2";
    a.Add(zilch);
    a.Add(first);
    a.Add(second);
    a.Add(third);
    if (a.Contains("three")){
        a.Add("four");
    }
    a.RemoveAt(2);
    if (a.IndexOf("four") != 4) {
        a.Add(fourth);
    }
    if (a.Contains("two")) {
        a.Add(twopointtwo);
    }
    printL(a);
}
public void printL (List<string> a){
    string result = "";
    foreach (string element in a)
    {
        result += "\n" + element;
    }
    MessageBox.Show(result);
}
}
```

Can you figure out why "2.2" never gets added to the list, even though it's declared here?

RemoveAt() removes the element at index #2—which is the third element in the list.

The printL() method uses a foreach loop to go through a list of strings, add each of them to one big string, and then show it in a message box.

The foreach loop goes through all of the elements in the list and prints them.

THERE ARE NO DUMB QUESTIONS

Q: **Q: So why would I ever use an enum instead of a `List`? Don't they solve the same problem?**

A: **A:** Enums are a little different than `List`s. First and foremost, enums are **types**, while `List`s 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 `List` can store just about anything. Since it's a list of ***objects***, each element in a list can have its own methods and properties. 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). So you can't store reference variables in them. 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: **Q: OK, it sounds like `List`s are pretty powerful. So why would I ever want to use an array?**

A: **A:** If you know that you have a fixed number of items to work with, or if you want a fixed sequence of values with a fixed length, then an array is perfect. Luckily, you can easily convert any list to an array using the `ToArray()` method...and you can convert an array to a list using one of the overloaded constructors for the `List<T>` object.

NOTE

Arrays also 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: **Q: I don't get the name "generic." Why is it called a generic collection? Why isn't an array generic?**

A: **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: **Q: OK, that explains the "collection" part. But what makes it "generic"?**

A: **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.

NOTE

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: **Q: Can I have a list that doesn't have a type?**

A: **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 `ArrayList`s 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.

You saw this code a few pages ago—it creates a new List<Shoe> and fills it with new Shoe objects.

```
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" });
```

The same code rewritten using a collection initializer

Notice how each Shoe object is initialized with its own object initializer? You can nest them inside a collection initializer, just like this.

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.

```
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.

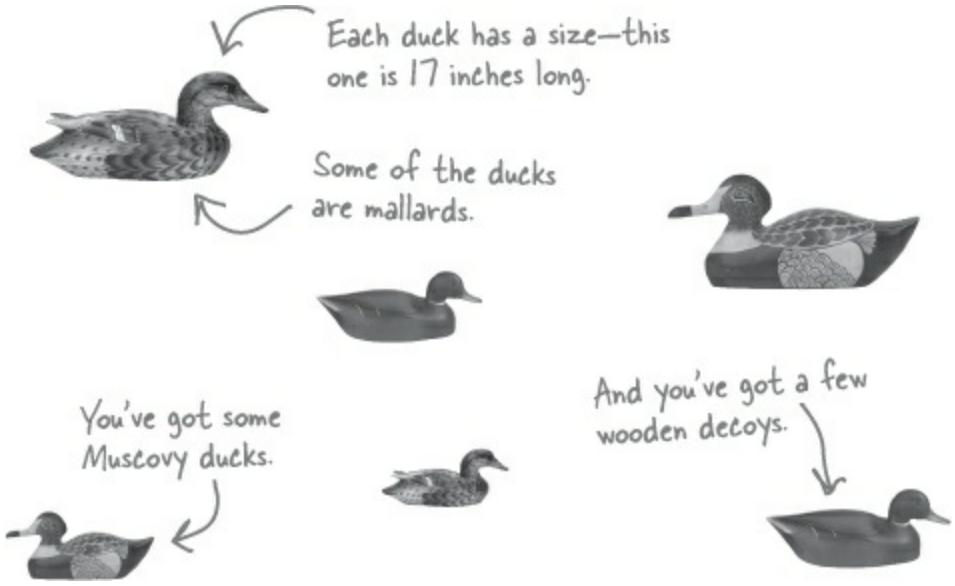
You're not limited to using "new" statements in the initializer—you can include variables, too.

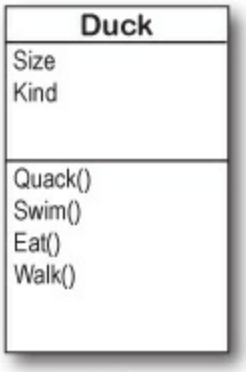
A collection initializer makes your code more compact by letting you combine creating a list with adding an initial set of items.

Let’s create a List of Ducks

DO THIS!

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.






```
class Duck {
    public int Size;
    public KindOfDuck Kind;
}

enum KindOfDuck {
    Mallard,
    Muscovy,
    Decoy,
}
```

The class has two public fields. It's also got some methods, which we're not showing here.

We'll use an enum called KindOfDuck to keep track of what sort of ducks are in your collection.

Add Duck and KindOfDuck to your project.

Here's the initializer for your List of Ducks

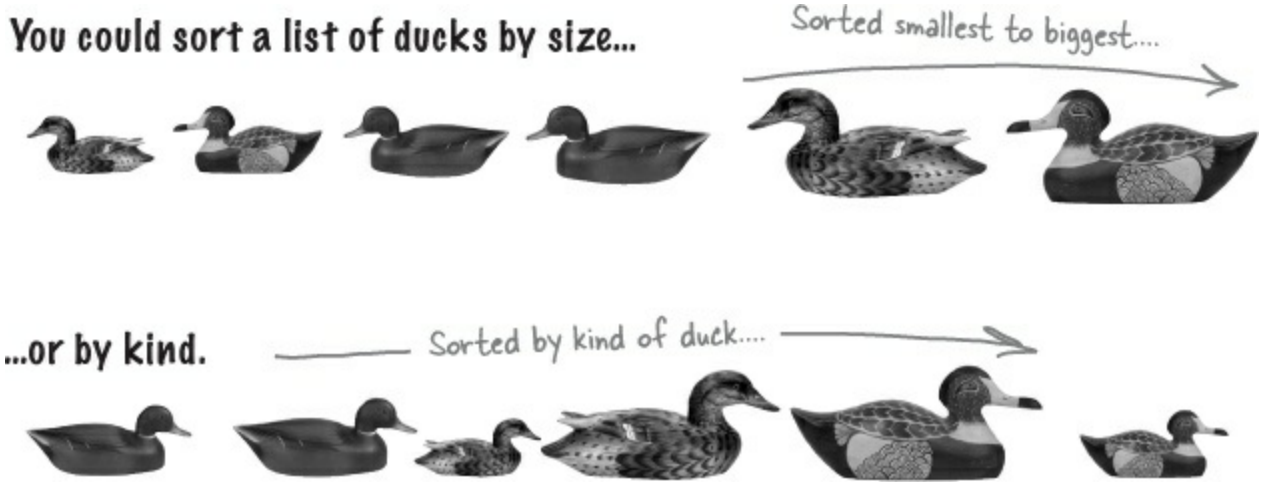
We've got six ducks, so we'll create a `List<Duck>` that has a collection initializer with six statements. Each statement in the initializer creates a new duck, using an object initializer to set each `Duck` object's `Size` and `Kind` field. **Add this code** to your `Main()` method in *Program.cs*:

```
List<Duck> ducks = new List<Duck>() {
    new Duck() { Kind = KindOfDuck.Mallard, Size = 17 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 18 },
    new Duck() { Kind = KindOfDuck.Decoy, Size = 14 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 11 },
    new Duck() { Kind = KindOfDuck.Mallard, Size = 14 },
    new Duck() { Kind = KindOfDuck.Decoy, Size = 13 },
};

// This keeps the output from disappearing before you can read it
Console.ReadKey();
```

You'll be adding code to your Main() method to print to the console. Make sure you keep this line at the end so the program stays open until you hit a key.

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.

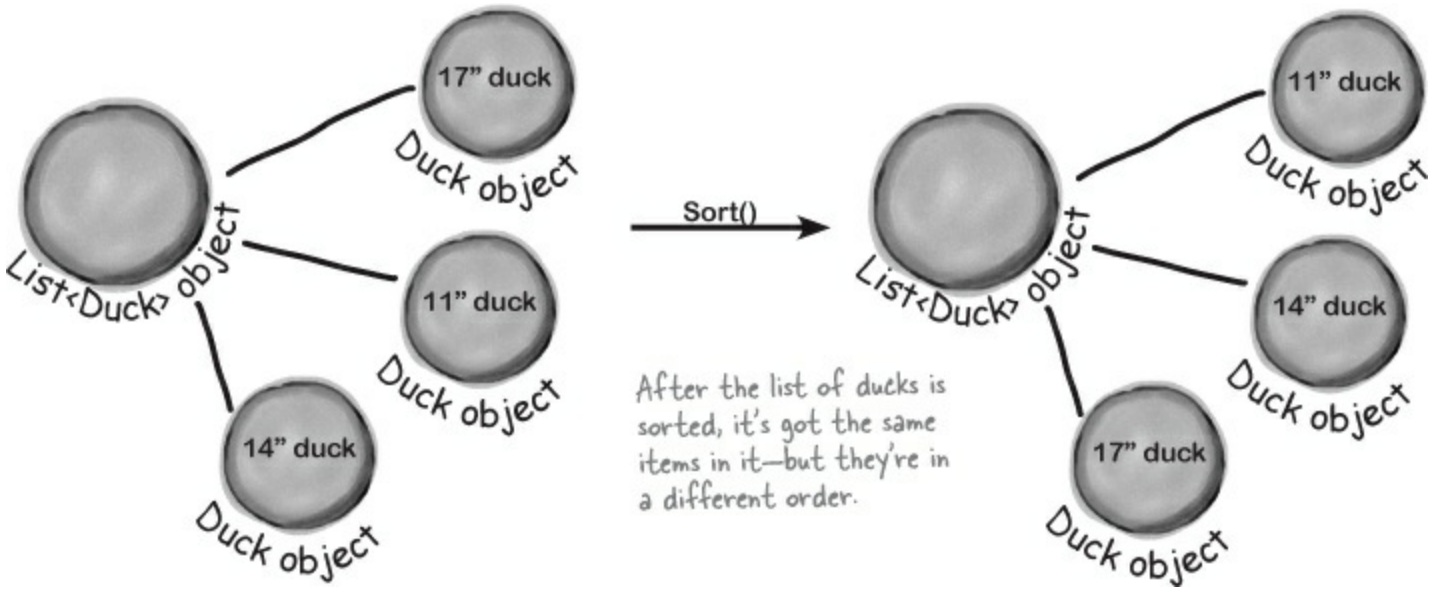


Lists know how to sort themselves

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.

NOTE

Technically, it's not the List<T> that knows how to sort itself. It depends on an IComparer<T> object, which you'll learn about in a minute.



IComparable<Duck> helps your list sort its ducks

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.

But sometimes you need to sort a list of objects that don't implement `IComparable<T>`, and .NET has another interface to help with that. You can pass `Sort()` an instance of a class that implements `IComparer<T>`. That interface also has one method. The `List` object's `Sort()` method uses the comparer object's `Compare()` method to compare pairs of objects, in order to figure out which one comes first in the sorted list.

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 is to modify the `Duck` class to implement `IComparable<Duck>`. To do that, we'd add 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;
    public KindOfDuck Kind;

    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 code to the end of your `Main()` method above the call to `Console.ReadKey()` to tell your list of ducks to sort itself. Use the debugger to see this at work by **putting a breakpoint** in the `CompareTo()` method.

```
ducks.Sort();
```

Use IComparer to tell your List how to sort

Lists have a special interface built into the .NET Framework that lets you build a separate class to help the `List<T>` sort out its members. By **implementing the `IComparer<T>` interface**, you can tell your `List` exactly how you want it to sort your objects. You do that by implementing the `Compare()` method in the `IComparer<T>` interface. 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.

Your List will sort differently depending on how you implement IComparer<T>.

Here's an example of how you'd declare a comparer class to compare `Duck` objects by size. Add it to your project as a new class:

This class implements IComparer, and specifies the type of object it can sort: Duck objects.

```
class DuckComparerBySize : IComparer<Duck>
{
    public int Compare(Duck x, Duck y)
    {
        if (x.Size < y.Size)
            return -1;
        if (x.Size > y.Size)
            return 1;
        return 0;
    }
}
```

These will always match: the same type in each.

The Compare() method returns an int, and has two parameters: both of the type you're sorting.

You can do whatever types of comparisons you want in the method.

Any negative number means object x should go before object y. x is "less than" y.

Any positive value means object x should go after object y. x is "greater than" y.

0 means that these two objects should be treated as the same (using this comparison calculation).

Here's a method to print the ducks in a List<Duck>.

Add this PrintDucks method to your Program class in your project so you can print the ducks in a list.

Update your Main() method to call it before and after you sort the list so you can see the results!

```
public static void PrintDucks(List<Duck> ducks)
{
    foreach (Duck duck in ducks)
        Console.WriteLine(duck.Size.ToString() + "-inch " + duck.Kind.ToString());
    Console.WriteLine("End of ducks!");
}
```

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.

We left out the code you already saw a few pages ago to initialize the list. Make sure you initialize your list before you try to sort it! If you don't, you'll get a null pointer exception.

```
DuckComparerBySize sizeComparer = new DuckComparerBySize();
ducks.Sort(sizeComparer);
PrintDucks(ducks);
```

You'll pass Sort() a reference to the new DuckComparerBySize object as its parameter.

Add this code to your program's Main() method to see how the ducks get sorted.

Sorted smallest to biggest....

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:

This comparer sorts by duck type. Remember, when you compare the enum Kind, you're comparing their **index** values.

So Mallard comes before Muscovy, which comes before Decoy.

```
class DuckComparerByKind : IComparer<Duck> {
    public int Compare(Duck x, Duck y) {
        if (x.Kind < y.Kind)
            return -1;
        if (x.Kind > y.Kind)
            return 1;
        else
            return 0;
        }
}
```

We compared the ducks' Kind properties, so the ducks are sorted based on the index value of the Kind property, a KindOfDuck enum.

Notice how "greater than" and "less than" have a different meaning here. We used < and > to compare enum index values, which lets us put the ducks in order.

Here's an example of how enums and Lists work together. Enums stand in for numbers, and are used in sorting of lists.

```
DuckComparerByKind kindComparer = new DuckComparerByKind();
ducks.Sort(kindComparer);
PrintDucks(ducks);
```

More duck sorting code for your Main() method.

Sorted by kind of duck...

IComparer can do complex comparisons

If you don't provide `Sort()` with an `IComparer<T>` object, it uses a default one that can sort value types or compare references. Flip to Leftover #6 in the Appendix to learn a little more about comparing objects.

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();

comparer.SortBy = SortCriteria.KindThenSize;
ducks.Sort(comparer);
PrintDucks(ducks);

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.

Here's how we'd use this comparer object. First we'd instantiate it as usual. Then we can set the object's SortBy field before calling ducks.Sort(). Now you can change the way the list sorts its ducks just by changing one field in the object. Add this code to the end of your Main() method. Now it sorts and re-sorts the list a bunch of times!

EXERCISE

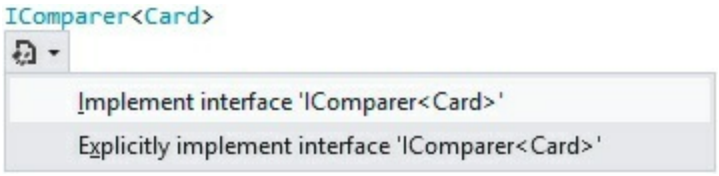
Create five random cards and then sort them.

- **❶ CREATE CODE TO MAKE A JUMBLED SET OF CARDS.**
  Create a new Console Application and add code to the `Main()` method that creates five random `Card` objects. After you create each object, use the built-in `Console.WriteLine()` method to write its name to the output. Use `Console.ReadKey()` at the end of the program to keep your window from disappearing when the program finishes.
- **❷ CREATE A CLASS THAT IMPLEMENTS ICOMPARER<CARD> TO SORT THE CARDS.**
  Here's a good chance to use that IDE shortcut to implement an interface:

```
class CardComparer_byValue : IComparer<Card>
```

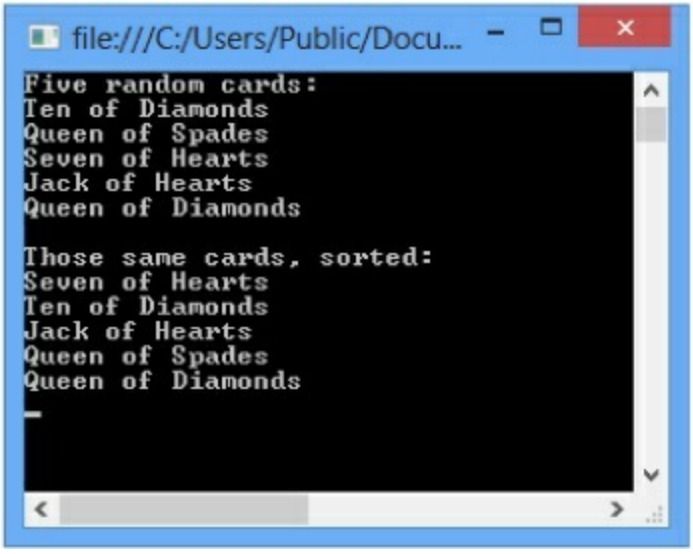
  Then click on `IComparer<Card>` and hover over the I. You'll see a box appear underneath it. When you click on the box, the IDE pops up its "Implement interface" window:

> **NOTE**
>
> Sometimes it's a little hard to get this box to pop up, so the IDE has a useful shortcut.: just press Ctrl-period.



  Click on "Implement interface IComparer<Card>" in the box to tell the IDE to automatically fill in all of the methods and properties that you need to implement. In this case, it creates an empty `Compare()` method to compare two cards, `x` and `y`. Write the method so that it returns 1 if `x` is bigger than `y`, –1 if it's smaller, and 0 if they're the same card. In this case, make sure that any King comes after any Jack, which comes after any 4, which comes after any Ace.
- **❸ MAKE SURE THE OUTPUT LOOKS RIGHT.**
  Here's what your output window should look like after you click the button.

When you use the built-in Console.WriteLine() method, it adds a line to this output. Console.ReadKey() waits for you to press a key before the program ends.



Your IComparer object needs to sort the cards by value, so the cards with the lowest values are first in the list.

EXERCISE SOLUTION

Create five random cards and then sort them.

```
class CardComparer_byValue : IComparer<Card> {
    public int Compare(Card x, Card y) {
        if (x.Value < y.Value) {
            return -1;
        }
        if (x.Value > y.Value) {
            return 1;
        }
        if (x.Suit < y.Suit) {
            return -1;
        }
        if (x.Suit > y.Suit) {
            return 1;
        }
        return 0;
    }
}
```

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 x has a bigger value, return 1. If x's value is smaller, return -1. Remember, both return statements end the method immediately.

These statements only get executed if x and y have the same value—that means the first two return statements weren't executed.

If none of the other four return statements were hit, the cards must be the same—so return zero.

```
static void Main(string[] args)
{
    Random random = new Random();
    Console.WriteLine("Five random cards:");
    List<Card> cards = new List<Card>();
    for (int i = 0; i < 5; i++)
    {
        cards.Add(new Card((Suits)random.Next(4),
                           (Values)random.Next(1, 14)));
        Console.WriteLine(cards[i].Name);
    }

    Console.WriteLine();
    Console.WriteLine("Those same cards, sorted:");
    cards.Sort(new CardComparer_byValue());
    foreach (Card card in cards)
    {
        Console.WriteLine(card.Name);
    }
    Console.ReadKey();
}
```

Here's 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.

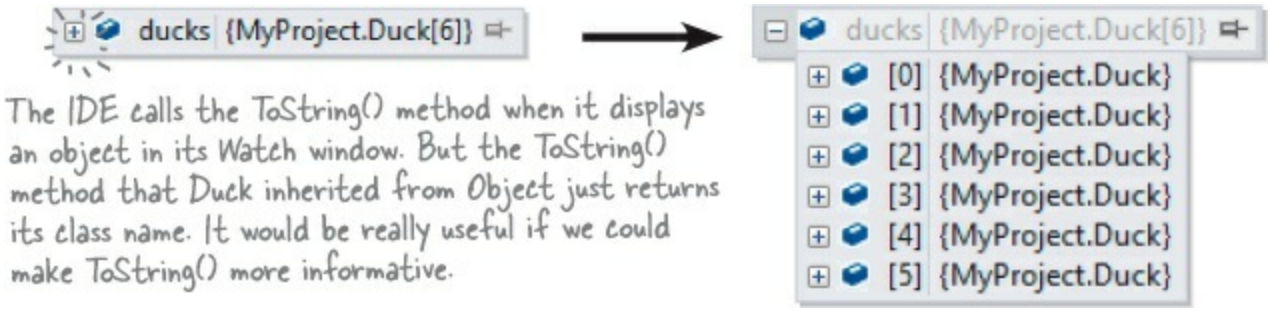
We're using `Console.ReadKey()` to keep console applications from exiting after they finish. This is great for learning, but not so great if you want to write real command-line applications. If you use Ctrl-F5 to start your program, the IDE runs it without debugging. When it finishes, it prints "Press any key to continue..." and waits for a keypress. But it doesn't debug your program (because it's running without debugging), so your breakpoints and watches won't work.

Overriding a ToString() method lets an object describe itself

Every .NET object has a **method called `ToString()` that converts it to a string**. By default, it just returns the name of your class (`MyProject.Duck`). The method is inherited from `Object` (remember, that's the base class for every object). This is a really useful method, and it's used a lot. For example, the + operator to concatenate strings **automatically calls an object's `ToString()` method**. And `Console.WriteLine()` or `String.Format()` will automatically call it when you pass objects to them, which can really come in handy when you want to turn an object into a string.

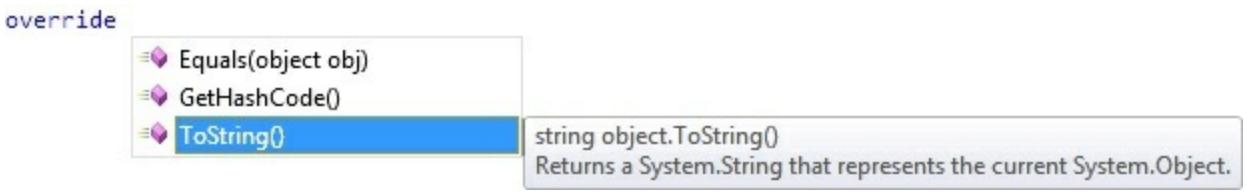
> **NOTE**
>
> So instead of passing a value to Console.WriteLine(), String.Format(), etc., you can pass an object — its ToString() method is called automatically. That also works with value types like ints and enums, too!

Go back to your duck sorting program. Put a breakpoint in the `Main()` method anywhere after the list is initialized and debug your program. Then **hover over any `ducks` variable** so it shows the value in a window. Any time you look at a variable in the debugger that's got a reference to a `List`, you can explore the contents of it by clicking the + button:



Hmm, that's not as useful as we'd hoped. You can see that there are six `Duck` objects in the list ("MyProject" is the namespace we used). If you click the + button next to 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?

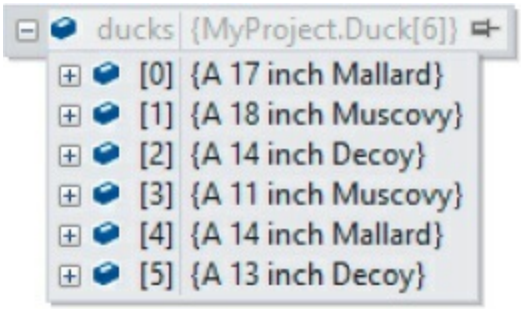
Luckily, `ToString()` is a `virtual` method on `Object`, the base class of every object. So all you need to do is **override the `ToString()` method** — and when you do, you'll see the results immediately in the IDE's Watch window! Open up your `Duck` class and start adding a new method by typing **`override`**. 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.ToString();
}
```

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 and shows you its contents.

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.ToString() + "-inch " + duck.Kind.ToString());
}
```

You can also leave off ".ToString()" and the + operator will call it automatically.

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:

```
public static void PrintDucks(List<Duck> ducks) {
    foreach (Duck duck in ducks) {
        Console.WriteLine(duck);
    }
    Console.WriteLine("End of ducks!");
}
```

If you pass Console.WriteLine() a reference to an object, it will call that object's ToString() method automatically.

Add this to your `Ducks` program and run it 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.ToString() + " of " + Suit.ToString(); }
}
```

You're still allowed to call ToString() like this, but now you know it's not necessary in this case, because + calls it automatically.

That's exactly what its `ToString()` method should do. So add a `ToString()` method to the `Card` class:

```
public override string ToString()
{
    return Name;
}
```

NOTE

ToString() is useful for a lot more than just making your objects easier to identify in the IDE. Keep your eyes open over the next few chapters, and you'll see how useful it is for every object to have a way to convert itself to a string. That's why every object has a ToString() method.

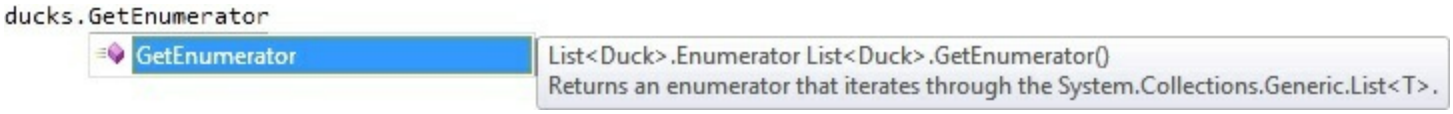
Now your programs that use `Card` objects will be easier to debug.

FOREACH LOOPS UP CLOSE

When you write a foreach loop, you're using IEnumerable<T>

> **NOTE**
>
> Collection initializers work with ANY IEnumerable<T> class — as long as it also has a method called Add().

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:



Add a line to create a new array of `Duck` objects:

```
Duck[] duckArray = new Duck[6];
```

Then type `duckArray.GetEnumerator` — the array also has a `GetEnumerator()` method. That's because all `List`s, and arrays implement an interface called **`IEnumerable<T>`**, which contains one method. That method, `GetEnumerator()`, returns an **Enumerator object**.

It's the `Enumerator` object that provides the machinery that lets you loop through a list in order. Here's a `foreach` loop that loops through a `List<Duck>` with a variable called `duck`:

```
foreach (Duck duck in ducks) {
   Console.WriteLine(duck);
}
```

And 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);
}
IDisposable disposable = enumerator as IDisposable;
if (disposable != null) disposable.Dispose();
```

(Don't worry about the last two lines for now. You'll learn about `IDisposable` in Chapter 9.)

> **NOTE**
>
> Technically, there's a little more than this, but you get the idea....

Those two loops print out the same ducks. You can see this for yourself by running both of them; they'll both have the same output.

> **When a collection implements IEnumerable<T>, it's giving you a way to write a loop that goes through its contents in order.**

Here's what's going on. 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!

What do you think would happen during a `foreach` loop if your `ToString()` method changes one of the object's fields?

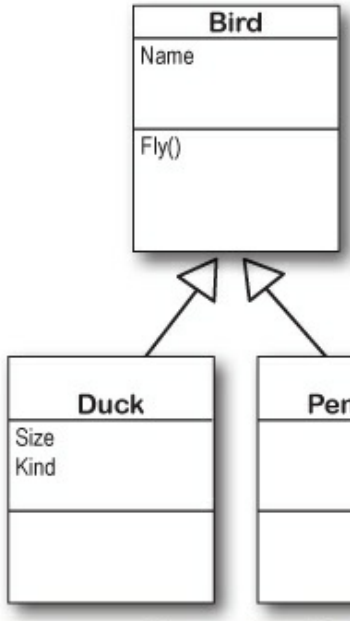
> **NOTE**
>
> Try experimenting with this by changing your Duck's ToString() to increment the Size property. Debug your program and hover over a Duck. Then do it again. Remember, each time you do it, the IDE calls its ToString() method.

You can upcast an entire list using IEnumerable

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.

Create a Console Application and 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.

DO THIS!




```
class Bird {
    public string Name { get; set; }
    public virtual void Fly() {
        Console.WriteLine("Flap, flap");
    }
    public override string ToString() {
        return "A bird named " + Name;
    }
}
```

```
class Penguin : Bird {
    public override void Fly() {
       Console.WriteLine("Penguins can't fly!");
    }
    public override string ToString() {
        return "A penguin named " + base.Name;
    }
}
```

Here's a Bird class, and a Penguin class that inherits from it. Add them to a new Console Application project, then copy your existing Duck class into it. Just change its declaration so that it extends Bird.

```
class Duck : Bird, IComparable<Duck> {
    // The rest of the class is the same
}
```

Here are the first few lines of your `Main()` method to initialize your list **and then upcast it**.

```
List<Duck> ducks = new List<Duck>() { /* initialize your list as usual */ }
IEnumerable<Bird> upcastDucks = ducks;
```

Take a close look at that last line of code. You're taking a reference to your `List<Duck>` and assigning it to an `IEnumerable<Bird>` interface variable. Debug through it, and you'll see it's pointing to the same object.

NOTE

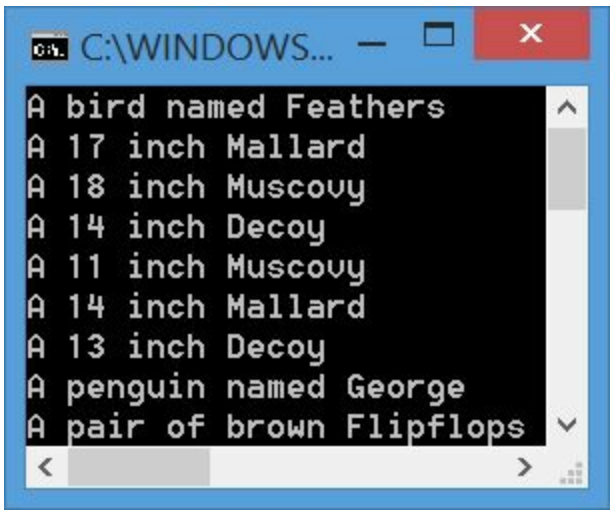
Copy the same collection initializer you've been using to initialize your List of ducks.

Combine your birds into a single list

Covariance is really useful when you want to take a collection of objects and add them to a more general list. Here's an example: if you have a list of `Bird` obects, you can add your `Duck` collection to it in one easy step. Here's an example that uses the `List.AddRange()` method, which you can use to add the contents of one list into another.

```
List<Bird> birds = new List<Bird>();

birds.Add(new Bird() { Name = "Feathers" });
birds.AddRange(ducks);
birds.Add(new Penguin() { Name = "George" });

foreach (Bird bird in birds) {
    Console.WriteLine(bird);
}
```

Next, change `birds` to a `List<Object>` variable — you'll also need to change `Bird` to `Object` in the List constructor and the foreach loop. Now you can any type of object to your list! **See if you can figure out** how to get your program to match this screenshot.



You'll need to add the Shoe class and Style enum from earlier in the chapter and override the Shoe.ToString() method.

You can build your own overloaded methods

You've been using **overloaded methods** and even an overloaded constructor that were part of the built-in .NET Framework classes and objects, so you can already see how useful they are. Wouldn't it be cool if you could build overloaded methods into your own classes? Well, you can — and it's easy! All you need to do is write two or more methods that have the same name but take different parameters.

NOTE

You can also use a using statement instead of changing the namespace. If you want to learn more about namespaces, take a minute and flip to Leftover #3 in the appendix.

DO THIS!

- **❶ Create a new Console Application project and add the `Card` class to it.**
  You can do this easily by right-clicking on the project in the Solution Explorer and selecting Existing Item from the Add menu. The IDE will make a copy of the class and add it to the project. The file will **still have the namespace from the old project**, so go to the top of the *Card.cs* file and change the `namespace` line to match the name of the new project you created. Then do the same for the `Values` and `Suits` enums.

> **NOTE**
>
> If you don't do this, you'll only be able to access the Card class by specifying its namespace (like oldnamespace.Card).

- **❷ Add some new overloaded methods to the `Card` class.**
  Create two `static DoesCardMatch()` methods. The first one should check a card's suit. The second should check its value. Both return `true` only if the card matches.

```
public static bool DoesCardMatch(Card cardToCheck, Suits suit) {
   if (cardToCheck.Suit == suit) {
      return true;
   } else {
      return false;
   }
}
public static bool DoesCardMatch(Card cardToCheck, Values value) {
   if (cardToCheck.Value == value) {
      return true;
   } else {
      return false;
   }
}
```

> **NOTE**
>
> Overloaded methods don't have to be static, but it's good to get a little more practice writing static methods.

- **❸ Add code to `Main()` to use the new methods.**
  Add this code to the `Main()` method in *Program.cs*:

```
Card cardToCheck = new Card(Suits.Clubs, Values.Three);
bool doesItMatch = Card.DoesCardMatch(cardToCheck, Suits.Hearts);
Console.WriteLine(doesItMatch);
```

  As soon as you type `DoesCardMatch(` the IDE will show you that you really did build an overloaded method:

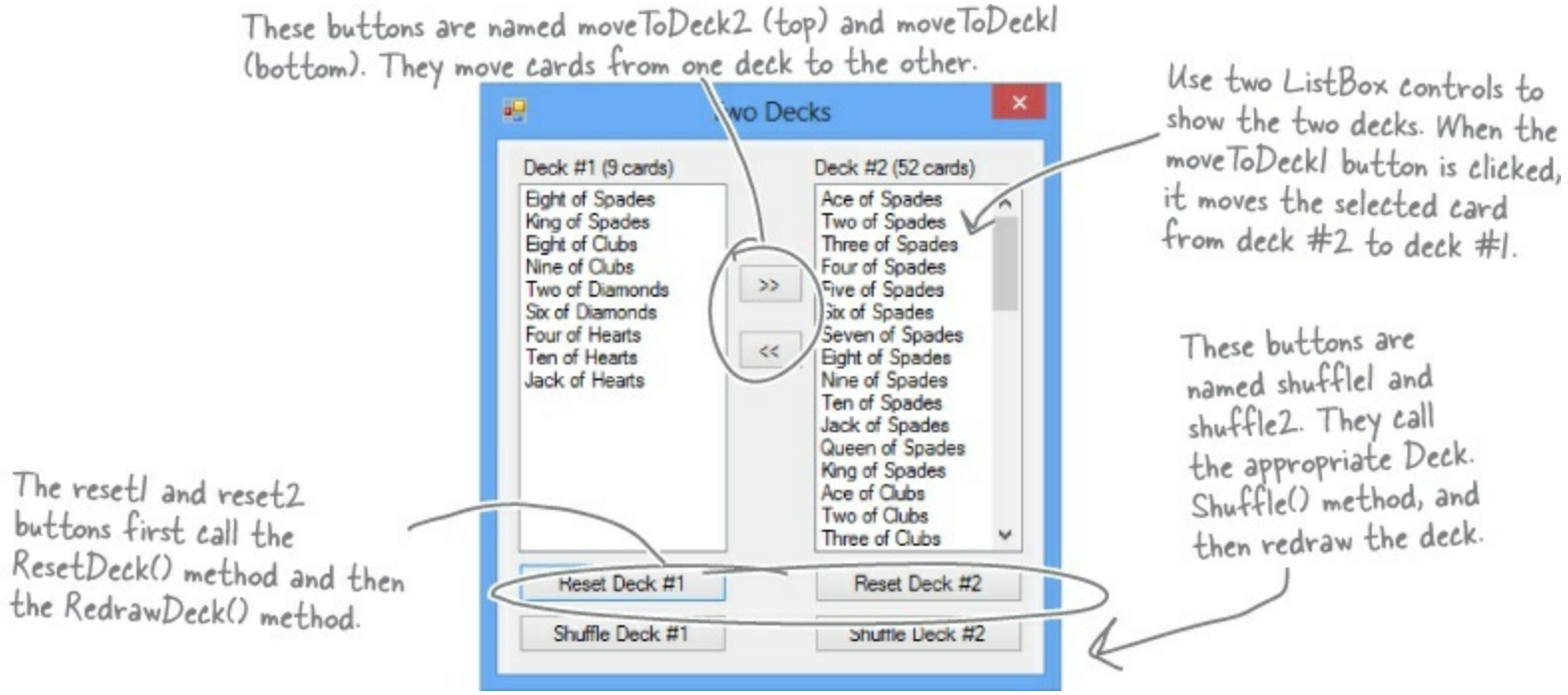
  Take a minute and play around with the two methods so you can get used to overloading.

EXERCISE

Get some practice using `Lists` by building a class to store a deck of cards, along with a form that uses it.

- **❶ BUILD A FORM THAT LETS YOU MOVE CARDS BETWEEN TWO DECKS.**
  You've built a `Card` class already. Now it's time to build a class to hold any number of cards, which we'll call `Deck`. A real-life deck has 52 cards, but the `Deck` class can hold any number of cards — or no cards at all.
  Then you'll build a form that shows you the contents of two `Deck` objects. When you first start the program, deck #1 has up to 10 random cards, and deck #2 is a complete deck of 52 cards, both sorted by suit and then value — and you can reset either deck to

its initial state using two Reset buttons. The form also has buttons (labeled "<<" and ">>") to move cards between the decks.



Remember, you can use a control's `Name` property to give it a name to make your code easier to read. Then when you double-click on the button, its event handler is given a matching name.

In addition to the event handlers for the six buttons, you'll need to add two methods for the form. First add a `ResetDeck()` method, which resets a deck to its initial state. It takes an int as a parameter: if it's passed 1, it resets the first `Deck` object by reinitializing it to an empty deck and a random number of up to 10 random cards; if it's passed 2, it resets the second `Deck` object so that it contains a full 52-card deck. Then add this method:

```
private void RedrawDeck(int DeckNumber) {
    if (DeckNumber == 1) {
        listBox1.Items.Clear();
        foreach (string cardName in deck1.GetCardNames())
            listBox1.Items.Add(cardName);
        label1.Text = "Deck #1 (" + deck1.Count + " cards)";
    } else {
        listBox2.Items.Clear();
        foreach (string cardName in deck2.GetCardNames())
            listBox2.Items.Add(cardName);
        label2.Text = "Deck #2 (" + deck2.Count + " cards)";
    }
}
```

The RedrawDeck() method shuffles the deck, draws random cards from it, and updates the two listbox controls with whatever happens to be in the two Deck objects.

Take a look at how we used the **foreach** loop to add each of the cards in the deck to the listbox.

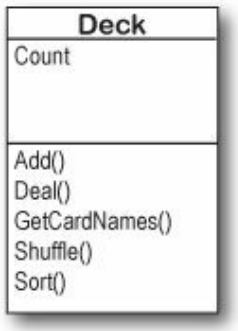
- **❷ BUILD THE DECK CLASS.**

Here's the skeleton for the `Deck` class. We've filled in several of the methods for you. You'll need to finish it by writing the `Shuffle()` and `GetCardNames()` methods, and you'll have to get the `Sort()` method to work. We also added two useful **overloaded constructors**: one that creates a complete deck of 52 cards, and another that takes an array of `Card` objects and loads them into the deck.

NOTE

When you have the declarations for a class without the implementation, it's called a "skeleton."

Deck
Count
Add() Deal() GetCardNames() Shuffle() Sort()



The Deck stores its cards in a List—but it keeps it private to make sure it's well encapsulated.

If you don't pass parameters into the constructor, it creates a complete deck of 52 cards.

The parameter has the type IEnumerable<Card>, which lets you pass any collection into the constructor, not just a List<T> or an array.

```
class Deck {
    private List<Card> cards;
    private Random random = new Random();

    public Deck() {
        cards = new List<Card>();
        for (int suit = 0; suit <= 3; suit++)
            for (int value = 1; value <= 13; value++)
                cards.Add(new Card((Suits)suit, (Values)value));
    }

    public Deck(IEnumerable<Card> initialCards) {
        cards = new List<Card>(initialCards);
    }

    public int Count { get { return cards.Count; } }

    public void Add(Card cardToAdd) {
        cards.Add(cardToAdd);
    }

    public Card Deal(int index) {
        Card CardToDeal = cards[index];
        cards.RemoveAt(index);
        return CardToDeal;
    }

    public void Shuffle() {
        // this method shuffles the cards by rearranging them in a random order
    }

    public IEnumerable<string> GetCardNames() {
        // this method returns a string array that contains each card's name
    }

    public void Sort() {
        cards.Sort(new CardComparer_bySuit());
    }
}
```

This overloaded constructor takes one parameter—an array of cards, which it loads as the initial deck.

Hint: the ListBox control's SelectedIndex property will be the same as the index of the card in the list. You can pass it directly to the Deal() method. If no card is selected, it'll be less than zero. In that case, the moveToDeck button should do nothing.

The Deal method deals one card out of the deck—it removes the Card object from the deck and returns a reference to it. You can deal from the top of the deck by passing it 0, or deal from the middle by passing it the index of the card to deal.

Again, even though GetCardNames() returns an array, we expose IEnumerable<string>.

You'll need to write the Shuffle() method and the GetCardNames() method, and add a class that implements IComparer to make the Sort() method work. And you'll need to add the Card class you already wrote. If you use Add Existing Item to add it, don't forget to change its namespace.

Another hint: the form makes it really easy to test your Shuffle() method. Keep clicking the "Reset Deck #1" button until you get a three-card deck. That'll make it easy to see if your shuffling code works.

EXERCISE SOLUTION

Build a class to store a deck of cards, along with a form that uses it.

```
class Deck {
    private List<Card> cards;
    private Random random = new Random();
    public Deck() {
        cards = new List<Card>();
        for (int suit = 0; suit <= 3; suit++)
            for (int value = 1; value <= 13; value++)
                cards.Add(new Card((Suits)suit, (Values)value));
    }
    public Deck(IEnumerable<Card> initialCards) {
        cards = new List<Card>(initialCards);
    }
    public int Count { get { return cards.Count; } }
    public void Add(Card cardToAdd) {
        cards.Add(cardToAdd);
    }
    public Card Deal(int index) {
        Card CardToDeal = cards[index];
        cards.RemoveAt(index);
        return CardToDeal;
    }
    public void Shuffle() {
        List<Card> newCards = new List<Card>();
        while (cards.Count > 0) {
            int CardToMove = random.Next(cards.Count);
            newCards.Add(cards[CardToMove]);
            cards.RemoveAt(CardToMove);
        }
        cards = newCards;
    }
    public IEnumerable<string> GetCardNames() {
        string[] CardNames = new string[cards.Count];
        for (int i = 0; i < cards.Count; i++)
            CardNames[i] = cards[i].Name;
        return CardNames;
    }
    public void Sort() {
        cards.Sort(new CardComparer_bySuit());
    }
}
```

Here's the constructor that creates a complete deck of 52 cards. It uses a nested for loop. The outside one loops through the four suits. That means the inside loop that goes through the 13 values runs four separate times, once per suit.

Here's the other constructor—this class has two overloaded constructors, each with different parameters.

The Add and Deal methods are pretty straightforward—they use the methods for the Cards list. The Deal method removes a card from the list, and the Add method adds a card to the list.

The Shuffle() method creates a new instance of List<Card> called newCards. Then it pulls random cards out of the cards field and sticks them in newCards until cards is empty. Once it's done, it resets the cards field to point to the new instance. The old instance won't have any more references pointing to it, so it'll get collected by the garbage collector.

Your GetCardNames() method needs to create an array that's big enough to hold all the card names. This one uses a for loop, but it could also use foreach.

```
class CardComparer_bySuit : IComparer<Card>
{
    public int Compare(Card x, Card y)
    {
        if (x.Suit > y.Suit)
            return 1;
        if (x.Suit < y.Suit)
            return -1;
        if (x.Value > y.Value)
            return 1;
        if (x.Value < y.Value)
            return -1;
        return 0;
    }
}

public partial class Form1 : Form
{
    Deck deck1;
    Deck deck2;
    Random random = new Random();

    public Form1() {
        InitializeComponent();
        ResetDeck(1);
        ResetDeck(2);
        RedrawDeck(1);
        RedrawDeck(2);
    }

    private void ResetDeck(int deckNumber) {
        if (deckNumber == 1) {
            int numberOfCards = random.Next(1, 11);
            deck1 = new Deck(new Card[] { });
            for (int i = 0; i < numberOfCards; i++)
                deck1.Add(new Card((Suits)random.Next(4),
                                   (Values)random.Next(1, 14)));
            deck1.Sort();
        } else
            deck2 = new Deck();
    }
```

Sorting by suit is a lot like sorting by value. The only difference is that in this case the suits are compared first, and then the values are compared only if the suits match.

Instead of using if/else if, we used a series of if statements. This works because each if statement only executes if the previous one didn't—otherwise, the previous one would have returned.

The form's constructor needs to reset the two decks, and then it draws them.

To reset deck #1, this method first uses random.Next() to pick how many cards will go into the deck, and then creates a new empty deck. It uses a for loop to add that many random cards. It finishes off by sorting the deck. Resetting deck #2 is easy—just create a new instance of Deck().

You've already got the form's RedrawDeck() method from the exercise instructions.

We're not done yet — flip the page!

Here's the rest of the code for the form.

Naming your controls makes it a lot easier to read your code. If these were called button1_Click, button2_Click, etc., you wouldn't know which button's code you were looking at!

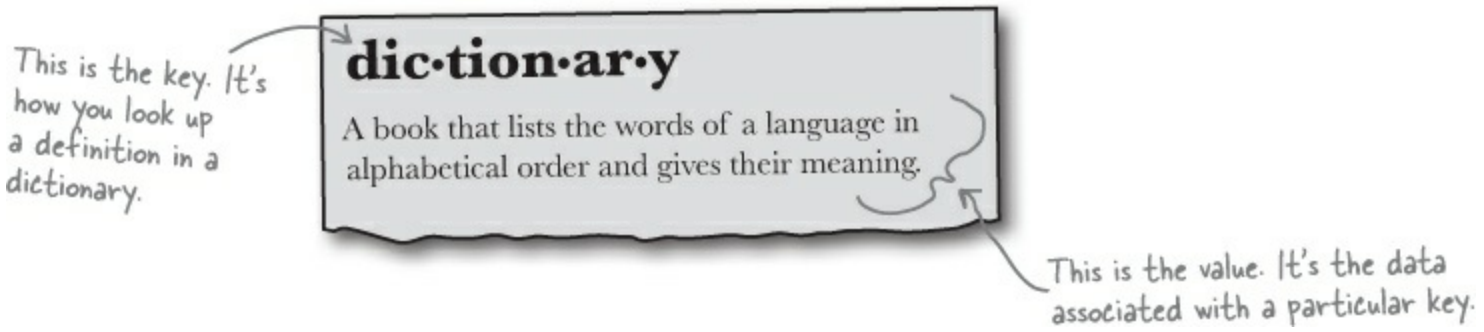
```
    private void reset1_Click(object sender, EventArgs e) {
        ResetDeck(1);
        RedrawDeck(1);
    }

    private void reset2_Click(object sender, EventArgs e) {
        ResetDeck(2);
        RedrawDeck(2);
    }

    private void shuffle1_Click(object sender, EventArgs e) {
        deck1.Shuffle();
        RedrawDeck(1);
    }

    private void shuffle2_Click(object sender, EventArgs e) {
        deck2.Shuffle();
        RedrawDeck(2);
    }

    private void moveToDeck1_Click(object sender, EventArgs e) {
        if (listBox2.SelectedIndex >= 0)
            if (deck2.Count > 0) {
                deck1.Add(deck2.Deal(listBox2.SelectedIndex));
            }
        RedrawDeck(1);
        RedrawDeck(2);
    }

    private void moveToDeck2_Click(object sender, EventArgs e) {
        if (listBox1.SelectedIndex >= 0)
            if (deck1.Count > 0)
                deck2.Add(deck1.Deal(listBox1.SelectedIndex));
        RedrawDeck(1);
        RedrawDeck(2);
    }
}
```

These buttons are pretty simple—first reset or shuffle the deck, then redraw it.

You can use the ListBox control's SelectedIndex property to figure out which card the user selected and then move it from one deck to the other. (If it's less than zero, no card was selected, so the button does nothing.) Once the card's moved, both decks need to be redrawn.

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> kv = new Dictionary <TKey, TValue>();
```

These are like List<T>. The <T> means a type goes in there. So you can declare one type for the key, and another type for the value.

These represent types. The first type in the angle brackets is always the key, and the second is always the data.

And here's a `Dictionary` in action:

```
private void button1_Click(object sender, EventArgs e)
{
    Dictionary<string, string> wordDefinition =
        new Dictionary<string, string>();

    wordDefinition.Add ("Dictionary", "A book that lists the words of a "
                       + "language in alphabetical order and gives their meaning");
    wordDefinition.Add ("Key", "A thing that provides a means of gaining access to "
                      + "our understanding something.");
    wordDefinition.Add ("Value", "A quantity, number, string, or reference.");

    if (wordDefinition.ContainsKey("Key")){
        MessageBox.Show(wordDefinition["Key"]);
    }

}
```

This dictionary has string values for keys, and strings as the value. It's like a real dictionary: term, and definition.

The Add() method is how you add keys and values to the dictionary.

Add() takes a key, and then the value.

ContainsKey() tells you if a key is in the dictionary. Handy, huh?

Here's how you get the value for a key. It looks kind of like an array index–get the value for the key at this index.

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 by passing a key and a value to 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 — which makes sense, because you stored those values in a dictionary so you could look them up using their unique keys. For this `Dictionary<string, string>`, you'll look up values using a string key, and it'll return a string.

```
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

Dictionaries are really versatile and can hold just about anything, from strings to numbers and even objects. 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> duckDictionary = new Dictionary<int, Duck>();
duckDictionary.Add(376, new Duck()
                { Kind = KindOfDuck.Mallard, Size = 15 });
```

NOTE

It's common to see a dictionary that maps integers to objects when you're assigning unique ID numbers to objects.

Build a program that uses a dictionary

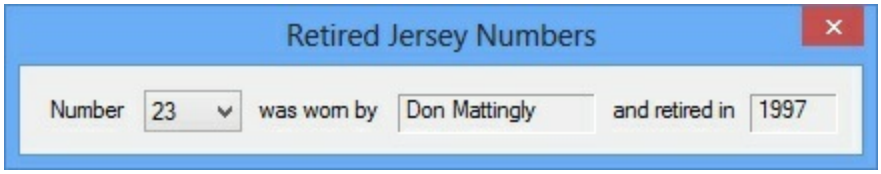
Here's a quick program that any New York baseball fan will like. When an important player retires, the team retires the player's jersey number. Let's build a program that looks up who wore famous numbers and when those numbers were retired. Here's a class to keep track of a jersey number:

DO THIS!

```
class JerseyNumber {
    public string Player { get; private set; }
    public int YearRetired { get; private set; }

    public JerseyNumber(string player, int numberRetired) {
        Player = player;
        YearRetired = numberRetired;
    }
}
```

Yogi Berra was #8 for one team and Cal Ripken, Jr., was #8 for another. 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?

Here's the form:



And here's all of the code for the form:

```
public partial class Form1 : Form {
   Dictionary<int, JerseyNumber> retiredNumbers = new Dictionary<int, JerseyNumber>() {
        {3, new JerseyNumber("Babe Ruth", 1948)},
        {4, new JerseyNumber("Lou Gehrig", 1939)},
        {5, new JerseyNumber("Joe DiMaggio", 1952)},
        {7, new JerseyNumber("Mickey Mantle", 1969)},
        {8, new JerseyNumber("Yogi Berra", 1972)},
        {10, new JerseyNumber("Phil Rizzuto", 1985)},
        {23, new JerseyNumber("Don Mattingly", 1997)},
        {42, new JerseyNumber("Jackie Robinson", 1993)},
        {44, new JerseyNumber("Reggie Jackson", 1993)},
   };

   public Form1() {
        InitializeComponent();
        foreach (int key in retiredNumbers.Keys) {
            number.Items.Add(key);
        }
   }

   private void number_SelectedIndexChanged(object sender, EventArgs e) {
        JerseyNumber jerseyNumber = retiredNumbers[(int)number.SelectedItem];
        nameLabel.Text = jerseyNumber.Player;
        yearLabel.Text = jerseyNumber.YearRetired.ToString();
   }
}
```

Use a collection initializer to populate your Dictionary with JerseyNumber objects.

Add each key from the dictionary to the ComboBox's Items collection.

The ComboBox's SelectedItem property is an Object. Since the Dictionary key is an int, we need to cast it to an int value before doing the lookup in the Dictionary.

Use the ComboBox's SelectedIndexChanged event to update the two labels on the form with the values from the JerseyNumber object retrieved from the Dictionary.

LONG EXERCISE

Build a game of **Go Fish!** that you can play against the computer.

This exercise is a little different....

There's a good chance that you're learning C# because you want a job as a professional developer. That's why we modeled this exercise after a professional assignment. When you're working as a programmer on a team, you don't usually build a complete program from start to finish. Instead, you'll build a *piece* of a bigger program. So we're going to give you a puzzle that's got some of the pieces already filled in. The code for the form is given to you in step #3. You just have to type it in — which may seem like a great head start, but it means that your classes ***have to work with that code***. And that can be a challenge!

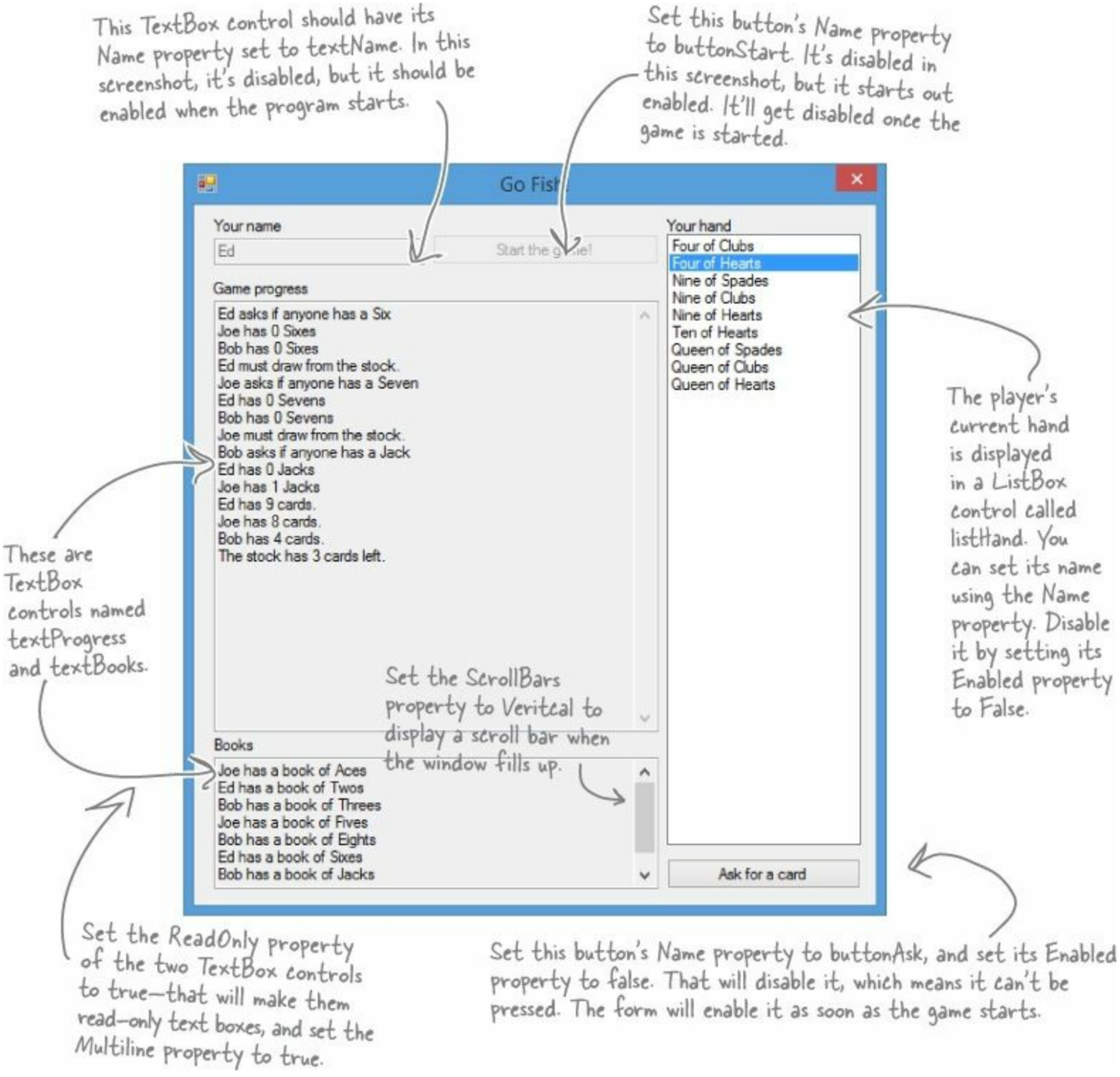
- **❶ START WITH THE SPEC.**
  Many professional software projects start with a specification, and this one is no exception. You'll be building a game of the classic card game ***Go Fish!*** Different people play the game by slightly different rules, so here's a recap of the rules you'll be using:

  The game starts with a deck of 52 cards. Five cards are dealt to each player. The pile of cards that's left after everyone's dealt a hand is called the **stock**. Each player takes turns asking for a value ("Do you have any sevens?"). Any other player holding cards with that value must hand them over. If nobody has a card with that value, then the player must "go fish" by taking a card from the stock.
  The goal of the game is to make books, where a book is the complete set of all four cards that have the same value. The player with the most books at the end of the game is the winner. As soon as a player collects a book, he places it face-up on the table so all the other players can see what books everyone else has.
  When placing a book on the table causes a player to run out of cards, then he has to draw five more cards from the stock. If there are fewer than five cards left in the stock, he takes all of them. The game is over as soon as the stock is out of cards. The winner is then chosen based on whoever has the most books.
  For this computer version of *Go Fish*, there are two computer players and one human player. Every round starts with the human player selecting one of the cards in his hand, which is displayed at all times. He does this by choosing one of the cards and indicating that he will ask for a card. Then the two computer players will ask for their cards. The results of each round will be

displayed. This will repeat until there's a winner.
The game will take care of all of the trading of cards and pulling out of books automatically. Once there's a winner, the game is over. The game displays the name of the winner (or winners, in case of a tie). No other action can be taken — the player will have to restart the program in order to start a new game.

If you don't know what you're building before you start, then how would you know when you're done? That's why many professional software projects start with a specification that tells you what you're going to build.

- **❷ BUILD THE FORM.**
Build the form for the *Go Fish!* game. It should have a `ListBox` control for the player's hand, two `TextBox` controls for the progress of the game, and a button to let the player ask for a card. To play the game, the user will select one of the cards from the hand and click the button to ask the computer players if they have that card.



We're not done yet — flip the page!

- **❸ HERE'S THE CODE FOR THE FORM.**
Enter it exactly like you see here. The rest of the code that you write will have to work with it.

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
    }

    private Game game;

    private void buttonStart_Click(object sender, EventArgs e) {
        if (String.IsNullOrEmpty(textName.Text)){
            MessageBox.Show("Please enter your name", "Can't start the game yet");
            return;
        }
        game = new Game(textName.Text, new List<string> { "Joe", "Bob" }, textProgress);
        buttonStart.Enabled = false;
        textName.Enabled = false;
        buttonAsk.Enabled = true;
        listHand.Enabled = true;
        UpdateForm();
    }

    private void UpdateForm() {
        listHand.Items.Clear();
        foreach (String cardName in game.GetPlayerCardNames())
            listHand.Items.Add(cardName);
        textBooks.Text = game.DescribeBooks();
        textProgress.Text += game.DescribePlayerHands();
        textProgress.SelectionStart = textProgress.Text.Length;
        textProgress.ScrollToCaret();
    }

    private void buttonAsk_Click(object sender, EventArgs e) {
        textProgress.Text = "";
        if (listHand.SelectedIndex < 0) {
            MessageBox.Show("Please select a card");
            return;
        }
        if (game.PlayOneRound(listHand.SelectedIndex)) {
            textProgress.Text += "The winner is... " + game.GetWinnerName();
            textBooks.Text = game.DescribeBooks();
            buttonAsk.Enabled = false;
            listHand.Enabled = false;
        } else
            UpdateForm();
    }
}
```

This is the only class that the form interacts with. It runs the whole game.

The Enabled property enables or disables a control on the form.

When you start a new game, it creates a new instance of the Game class, enables the Ask button, disables the Start Game button, and then redraws the form.

This method clears and repopulates the ListBox that holds the player's hand, and then updates the text boxes.

Using SelectionStart and ScrollToCaret() like this scrolls the text box to the end, so if there's too much text to display at once it scrolls down to the bottom.

The SelectionStart line moves the flashing text box cursor to the end, and once it's moved, the ScrollToCaret() method scrolls the text box down to the cursor.

The player selects one of the cards and clicks the Ask button to see if any of the other players have a card that matches its value. The Game class plays a round using the PlayOneRound() method.

- **❹ YOU'LL NEED THIS CODE, TOO.**
  You'll need the code you wrote before for the `Card` class, the `Suits` and `Values` enums, the `Deck` class, and the `CardComparer_byValue` class. But you'll need to add a few more methods to the `Deck` class...and you'll need to understand them in order to use them.

```
public Card Peek(int cardNumber) {
    return cards[cardNumber];
}
```

The Peek() method lets you take a peek at one of the cards in the deck without dealing it.

```
public Card Deal() {
    return Deal(0);
}
```

Someone overloaded Deal() to make it a little easier to read. If you don't pass it any parameters, it deals a card off the top of the deck.

```
public bool ContainsValue(Values value) {
    foreach (Card card in cards)
        if (card.Value == value)
            return true;
    return false;
}
```

The ContainsValue() method searches through the entire deck for cards with a certain value, and returns true if it finds any. Can you guess how you'll use this in the Go Fish game?

```
public Deck PullOutValues(Values value) {
    Deck deckToReturn = new Deck(new Card[] { });
    for (int i = cards.Count - 1; i >= 0; i--)
        if (cards[i].Value == value)
            deckToReturn.Add(Deal(i));
    return deckToReturn;
}
```

You'll use the PullOutValues() method when you build the code to get a book of cards from the deck. It looks for any cards that match a value, pulls them out of the deck, and returns a new deck with those cards in it.

```
public bool HasBook(Values value) {
    int NumberOfCards = 0;
    foreach (Card card in cards)
        if (card.Value == value)
            NumberOfCards++;
    if (NumberOfCards == 4)
        return true;
    else
        return false;
}
```

The HasBook() method checks a deck to see if it contains a book of four cards of whatever value was passed as the parameter. It returns true if there's a book in the deck, false otherwise.

```
public void SortByValue() {
    cards.Sort(new CardComparer_byValue());
}
```

The SortByValue() method sorts the deck using a CardComparer_byValue object.

Still not done — flip the page!

- **❺ NOW COMES THE HARD PART: BUILD THE PLAYER CLASS.**

There's an instance of the `Player` class for each of the three players in the game. They get created by the `buttonStart` button's event handler.

```
class Player
{
    private string name;
    public string Name { get { return name; } }
    private Random random;
    private Deck cards;
    private TextBox textBoxOnForm;

    public Player(String name, Random random, TextBox textBoxOnForm) {
        // The constructor for the Player class initializes four private fields, and then
        // adds a line to the TextBox control on the form that says, "Joe has just
        // joined the game"—but use the name in the private field, and don't forget to
        // add a line break at the end of every line you add to the TextBox.
    }

    public IEnumerable<Values> PullOutBooks() { } // see the facing page for the code

    public Values GetRandomValue() {
        // This method gets a random value—but it has to be a value that's in the deck!
    }

    public Deck DoYouHaveAny(Values value) {
        // This is where an opponent asks if I have any cards of a certain value
        // Use Deck.PullOutValues() to pull out the values. Add a line to the TextBox
        // that says, "Joe has 3 sixes"—use the new Card.Plural() static method
    }

    public void AskForACard(List<Player> players, int myIndex, Deck stock) {
        // Here's an overloaded version of AskForACard()—choose a random value
        // from the deck using GetRandomValue() and ask for it using AskForACard()
    }

    public void AskForACard(List<Player> players, int myIndex, Deck stock, Values value) {
        // Ask the other players for a value. First add a line to the TextBox: "Joe asks
        // if anyone has a Queen". Then go through the list of players that was passed in
        // as a parameter and ask each player if he has any of the value (using his
        // DoYouHaveAny() method). He'll pass you a deck of cards—add them to my deck.
        // Keep track of how many cards were added. If there weren't any, you'll need
        // to deal yourself a card from the stock (which was also passed as a parameter),
        // and you'll have to add a line to the TextBox: "Joe had to draw from the stock"
    }

    // Here's a property and a few short methods that were already written for you
    public int CardCount { get { return cards.Count; } }
    public void TakeCard(Card card) { cards.Add(card); }
    public IEnumerable<string> GetCardNames() { return cards.GetCardNames(); }
    public Card Peek(int cardNumber) { return cards.Peek(cardNumber); }
    public void SortHand() { cards.SortByValue(); }
}
```

Look closely at each of the comments—they tell you what the methods are supposed to do. Your job is to fill in the methods.

There's a rare case when an opponent's last card was taken by another player, so he has no cards left when AskForACard() is called. Can you figure out how to deal with this case?

- **❻ YOU'LL NEED TO ADD THIS METHOD TO THE PLAYER CLASS.**

  Here's the `PullOutBooks()` method for the `Player` class. It loops through each of the 13 card values. For each of the values, it counts all of the cards in the player's `cards` field that match the value. If the player has all four cards with that value, that's a complete book — it adds the value to the books variable to be returned, and it removes the book from the player's cards.

```
public IEnumerable<Values> PullOutBooks() {
    List<Values> books = new List<Values>();
    for (int i = 1; i <= 13; i++) {
        Values value = (Values)i;
        int howMany = 0;
        for (int card = 0; card < cards.Count; card++)
            if (cards.Peek(card).Value == value)
```

```
                howMany++;
            if (howMany == 4) {
                books.Add(value);
                cards.PullOutValues(value);
            }
        }
        return books;
    }
```

NOTE

A couple more things to think about

That Peek() method we added to the Deck class will come in handy. It lets the program look at one of the cards in the deck by giving its index number, but unlike Deal() it doesn't remove the card.

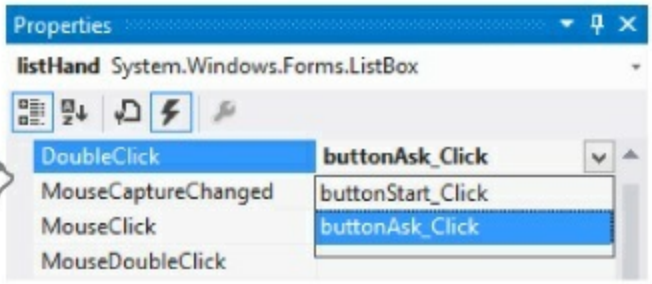
And you'll have to build TWO overloaded versions of theAskForACard() method. The first one is used by the opponents when they ask for cards — it'll look through their hands and find a card to ask for. The second one is used when the player asks for the card. Both of them ask EVERY other player (both computer and human) for any cards that match the value.

- **❼ YOU'LL NEED TO ADD THIS METHOD TO THE CARD CLASS.**
It's a static method to take a value and return its plural — that way a 10 will return "Tens" but a 6 will return "Sixes" (with "es" on the end). Since it's static, you call it with the class name — `Card.Plural()` — and not from an instance.

```
public partial class Card {
    public static string Plural(Values value) {
        if (value == Values.Six)
            return "Sixes";
        else
            return value.ToString() + "s";
    }
}
```

We used a partial class to add this static method to Card to make it easy for you to see what's going on. But you don't need to use a partial class—if you want, you can just add it straight into the existing Card class.

Once you build the "Ask for a card" button's event handler, you can also hook the ListBox's DoubleClick event up to it so you can also double-click on a card to ask for it.



Nearly there — keep flipping!

- **❽ THE REST OF THE JOB: BUILD THE GAME CLASS.**
The form keeps one instance of `Game`. It manages the game play. Look closely at how it's used in the form.

The Player and Game classes both use a reference to the multiline TextBox on the form to print messages for the user to read. Make sure you add "using System.Windows.Forms;" to the top of their files.

```
class Game {
    private List<Player> players;
    private Dictionary<Values, Player> books;
    private Deck stock;
    private TextBox textBoxOnForm;

    public Game(string playerName, IEnumerable<string> opponentNames, TextBox textBoxOnForm) {
        Random random = new Random();
        this.textBoxOnForm = textBoxOnForm;
        players = new List<Player>();
        players.Add(new Player(playerName, random, textBoxOnForm));
        foreach (string player in opponentNames)
            players.Add(new Player(player, random, textBoxOnForm));
        books = new Dictionary<Values, Player>();
        stock = new Deck();
        Deal();
        players[0].SortHand();
    }
    private void Deal() {
       // This is where the game starts—this method's only called at the beginning
       // of the game. Shuffle the stock, deal five cards to each player, then use a
       // foreach loop to call each player's PullOutBooks() method.
    }
    public bool PlayOneRound(int selectedPlayerCard) {
       // Play one round of the game. The parameter is the card the player selected
       // from his hand—get its value. Then go through all of the players and call
       // each one's AskForACard() methods, starting with the human player (who's
       // at index zero in the Players list—make sure he asks for the selected
       // card's value). Then call PullOutBooks()—if it returns true, then the
       // player ran out of cards and needs to draw a new hand. After all the players
       // have gone, sort the human player's hand (so it looks nice in the form).
       // Then check the stock to see if it's out of cards. If it is, reset the
       // TextBox on the form to say, "The stock is out of cards. Game over!" and return
       // true. Otherwise, the game isn't over yet, so return false.
    }
    public bool PullOutBooks(Player player) {
       // Pull out a player's books. Return true if the player ran out of cards, otherwise
       // return false. Each book is added to the Books dictionary. A player runs out of
       // cards when he''s used all of his cards to make books—and he wins the game.
    }
    public string DescribeBooks() {
       // Return a long string that describes everyone's books by looking at the Books
       // dictionary: "Joe has a book of sixes. (line break) Ed has a book of Aces."
    }
}
```

It's great for encapsulation, too. If you expose an IEnumerable<T> instead of, say, a List<T>, then you can't accidentally write code that modifies it.

Using `IEnumerable<T>` in public class members is a great way to make your classes more flexible, and that's something you need to think about when your code needs to be reused. Now someone else can use a `string[]`, `List<string>`, or something else entirely to instantiate the Game class.

NOTE

Here's a hint for writing the GetWinnerName() method: you'll need to create a new Dictionary<string, int> called winners at the top of the method. The winners dictionary will let you use each player's name to look up the number of books he made during the game. First you'll use a foreach loop to go through the books that the players made and build the dictionary. Then you'll use another foreach loop to find the highest number of books associated with any player. But there might be a tie — more than one player might have the most books! So you'll need one more foreach loop to look for all the players in winners that have the number of books that you found in the second loop and build a string that says who won.

```
public string GetWinnerName() {
       // This method is called at the end of the game. It uses its own dictionary
       // (Dictionary<string, int> winners) to keep track of how many books each player
       // ended up with in the books dictionary. First it uses a foreach loop
       // on books.Keys — foreach (Values value in books.Keys) — to populate
       // its winners dictionary with the number of books each player ended up with.
       // Then it loops through that dictionary to find the largest number of books
       // any winner has. And finally it makes one last pass through winners to come
       // up with a list of winners in a string ("Joe and Ed"). If there's one winner,
       // it returns a string like this: "Ed with 3 books". Otherwise, it returns a
       // string like this: "A tie between Joe and Bob with 2 books."
}

// Here are a couple of short methods that were already written for you:

public IEnumerable<string> GetPlayerCardNames() {
    return players[0].GetCardNames();
}

public string DescribePlayerHands() {
    string description = "";
    for (int i = 0; i < players.Count; i++) {
        description += players[i].Name + " has " + players[i].CardCount;
        if (players[i].CardCount == 1)
            description += " card." + Environment.NewLine;
        else
            description += " cards." + Environment.NewLine;
    }
    description += "The stock has " + stock.Count + " cards left.";
    return description;
}
```

USE ENVIRONMENT.NEWLINE TO ADD LINE BREAKS

You've been using \n throughout the book to add line breaks to message boxes. .NET also gives you a convenient constant for adding line breaks: Environment.NewLine. It always contains the constant value "\r\n". If you actually look at the characters that make up a Windows-formatted text file, at the end of every line you'll see two characters: '\r' and '\n'. Other operating systems (like Unix) only use a '\n' to indicate the end of each line. The MessageBox.Show() method is smart enough to automatically convert '\n' characters to line breaks, but your code can be easier to read if you use Environment.NewLine instead of escape characters. Also, Environment.NewLine is what gets appended to the end of each line when you use Console.WriteLine().

> Go to the Watch window and type (int)'\r' to cast the character \r to a number. It turns into 13. '\n' turns into 10. Every char turns into its own unique number called its Unicode value. You'll learn more about that in the next chapter.

LONG EXERCISE SOLUTION

Here are the filled-in methods in the `Game` class.

```
private void Deal() {
    stock.Shuffle();
    for (int i = 0; i < 5; i++)
        foreach (Player player in players)
            player.TakeCard(stock.Deal());
    foreach (Player player in players)
        PullOutBooks(player);
}

public bool PlayOneRound(int selectedPlayerCard) {
    Values cardToAskFor = players[0].Peek(selectedPlayerCard).Value;
    for (int i = 0; i < players.Count; i++) {
        if (i == 0)
            players[0].AskForACard(players, 0, stock, cardToAskFor);
        else
            players[i].AskForACard(players, i, stock);
        if (PullOutBooks(players[i])) {
            textBoxOnForm.Text += players[i].Name
                                + " drew a new hand" + Environment.NewLine;
            int card = 1;
            while (card <= 5 && stock.Count > 0) {
                players[i].TakeCard(stock.Deal());
                card++;
            }
        }
        players[0].SortHand();
        if (stock.Count == 0) {
            textBoxOnForm.Text =
                 "The stock is out of cards. Game over!" + Environment.NewLine;
            return true;
        }
    }
    return false;
}

public bool PullOutBooks(Player player)
{
    IEnumerable<Values> booksPulled = player.PullOutBooks();
    foreach (Values value in booksPulled)
        books.Add(value, player);
    if (player.CardCount == 0)
        return true;
    return false;
}
```

The Deal() method gets called when the game first starts—it shuffles the deck and then deals five cards to each player. Then it pulls out any books that the players happened to have been dealt.

After the player or opponent asks for a card, the game pulls out any books that he made. If a player's out of cards, he draws a new hand by dealing up to 5 cards from the stock.

As soon as the player clicks the "Ask for a card" button, the game calls AskForACard() with that card. Then it calls AskForACard() for each opponent.

After the round is played, the game sorts the player's hand to make sure it's displayed in order on the form. Then it checks to see if the game's over. If it is, PlayOneRound() returns true.

PullOutBooks() looks through a player's cards to see if he's got four cards with the same value. If he does, they get added to his books dictionary. And if he's got no cards left afterward, it returns true.

Bonus mini-exercise: When the game is done, the Start button stays disabled and the player just sees the results of the game. Can you figure out how to reenable the Start button when the game is over and have it start up a new game?

The form needs to display a list of books, so it uses DescribeBooks() to turn the player's books dictionary into words.

```
public string DescribeBooks() {
    string whoHasWhichBooks = "";
    foreach (Values value in books.Keys)
        whoHasWhichBooks += books[value].Name + " has a book of "
            + Card.Plural(value) + Environment.NewLine;
    return whoHasWhichBooks;
}

public string GetWinnerName() {
    Dictionary<string, int> winners = new Dictionary<string, int>();
    foreach (Values value in books.Keys) {
        string name = books[value].Name;
        if (winners.ContainsKey(name))
            winners[name]++;
        else
            winners.Add(name, 1);
    }
    int mostBooks = 0;
    foreach (string name in winners.Keys)
        if (winners[name] > mostBooks)
            mostBooks = winners[name];
    bool tie = false;
    string winnerList = "";
    foreach (string name in winners.Keys)
        if (winners[name] == mostBooks)
        {
            if (!String.IsNullOrEmpty(winnerList))
            {
                winnerList += " and ";
                tie = true;
            }
            winnerList += name;
        }
    winnerList += " with " + mostBooks + " books";
    if (tie)
        return "A tie between " + winnerList;
    else
        return winnerList;
}
```

Once the last card's been picked up, the game needs to figure out who won. That's what the GetWinnerName() does. And it'll use a dictionary called winners to do it. Each player's name is a key in the dictionary; its value is the number of books that player got during the game.

Next the game looks through the dictionary to figure the number of books that the player with the most books has. It puts that value in a variable called mostBooks.

Now that we know which player has the most books, the method can come up with a string that lists the winner (or winners).

We're not done yet — flip the page!

Here are the filled-in methods in the `Player` class.

```
public Player(String name, Random random, TextBox textBoxOnForm) {
    this.name = name;
    this.random = random;
    this.textBoxOnForm = textBoxOnForm;
    this.cards = new Deck( new Card[] {} );
    textBoxOnForm.Text += name +
                " has just joined the game" + Environment.NewLine;
}

public Values GetRandomValue() {
    Card randomCard = cards.Peek(random.Next(cards.Count));
    return randomCard.Value;
}

public Deck DoYouHaveAny(Values value) {
    Deck cardsIHave = cards.PullOutValues(value);
    textBoxOnForm.Text += Name + " has " + cardsIHave.Count + " "
          + Card.Plural(value) + Environment.NewLine;
    return cardsIHave;
}

public void AskForACard(List<Player> players, int myIndex, Deck stock) {
    if (stock.Count > 0) {
        if (cards.Count == 0)
            cards.Add(stock.Deal());
        Values randomValue = GetRandomValue();
        AskForACard(players, myIndex, stock, randomValue);
        if (stock.Count > 0 && players[0].CardCount == 0)
            players[0].cards.Add(stock.Deal());
    }
}

public void AskForACard(List<Player> players, int myIndex,
      Deck stock, Values value) {
    textBoxOnForm.Text += Name + " asks if anyone has a "
                                + value  + Environment.NewLine;
    int totalCardsGiven = 0;
    for (int i = 0; i < players.Count; i++) {
        if (i != myIndex) {
            Player player = players[i];
            Deck CardsGiven = player.DoYouHaveAny(value);
            totalCardsGiven += CardsGiven.Count;
            while (CardsGiven.Count > 0)
                cards.Add(CardsGiven.Deal());
        }
    }
    if (totalCardsGiven == 0 && stock.Count > 0) {
        textBoxOnForm.Text += Name +
              " must draw from the stock." + Environment.NewLine;
        cards.Add(stock.Deal());
    }
}
```

Here's the constructor for the Player class. It sets its private fields and adds a line to the progress text box saying who joined.

The GetRandomValue() method uses Peek() to look at a random card in the player's hand.

DoYouHaveAny() uses the PullOutValues() method to pull out and return all cards that match the parameter.

If an opponent gave up his last card, GetRandomValue() will try to call Deal() on an empty deck. These if statements prevent that from happening.

This handles an edge case where the human player's last card is taken by an opponent.

There are two overloaded AskForACard() methods. The first one is used by the opponents—it gets a random card from the hand and calls the other AskForACard().

This AskForACard() method looks through every player (except for the one asking), calls its DoYouHaveAny() method, and adds any cards handed over to the hand.

If no cards were handed over, the player draws from the stock using its Deal() method.

Bonus mini-exercise: Can you figure out a way to improve encapsulation and design in your `Player` class by replacing `List<Player>` with `IEnumerable<Player>` in the two `AskForACard()` methods without changing the way the software works? Flip to Leftover #8 in the Appendix for a useful tool to help with that.

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.

> **NOTE**
>
> 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

> **NOTE**
>
> 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!

> **NOTE**
>
> 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 collections are an important part of the .NET Framework

They're really useful — so much that the IDE automatically adds this statement to the top of every class you add to your project:

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

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.

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.

NOTE

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 new queue of strings.

```
Queue<string> myQueue = new Queue<string>();
myQueue.Enqueue("first in line");
myQueue.Enqueue("second in line");
myQueue.Enqueue("third in line");
myQueue.Enqueue("last in line");
```

Here's where we add four items to the queue. When we pull them out of the queue, they'll come out in the same order they went in.

Peek() lets you take a "look" at the first item in the queue without removing it.

```
string takeALook = myQueue.Peek();①
string getFirst = myQueue.Dequeue();②
string getNext = myQueue.Dequeue();③
int howMany = myQueue.Count;④
```

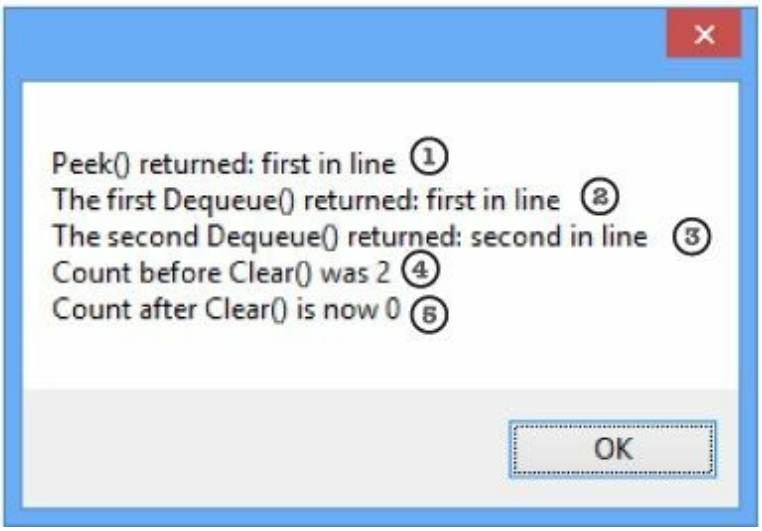
The first Dequeue() pulls the first item out of the queue. Then the second one shifts up into the first place—the next call to Dequeue() pulls that one out next.

The Clear() method **removes** all objects from the queue.

```
myQueue.Clear();
MessageBox.Show("Peek() returned: " + takeALook + "\n"
    + "The first Dequeue() returned: " + getFirst + "\n"
    + "The second Dequeue() returned: " + getNext + "\n"
    + "Count before Clear() was " + howMany + "\n"
    + "Count after Clear() is now " + myQueue.Count);⑤
```

The queue's Count property returns the number of items in the queue.

Objects in a queue need to wait their turn. The first one in the queue is the first one to come out of it.



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.

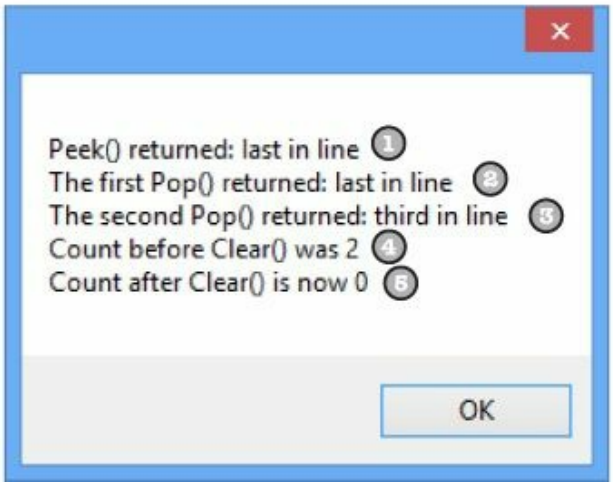
Creating a stack is just like creating any other generic collection.

When you push an item onto a stack, it pushes the other items back one notch and sits on top.

```
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");
1 string takeALook = myStack.Peek();
2 string getFirst = myStack.Pop();
3 string getNext = myStack.Pop();
4 int howMany = myStack.Count;
myStack.Clear();
MessageBox.Show("Peek() returned: " + takeALook + "\n"
    + "The first Pop() returned: " + getFirst + "\n"
    + "The second Pop() returned: " + getNext + "\n"
    + "Count before Clear() was " + howMany + "\n"
    + "Count after Clear() is now " + myStack.Count);
                                                    6
```

When you pop an item off the stack, you get the most recent item that was added.

You can also use Environment.NewLine instead of \n here, but we wanted the code to be easier to read.



The last object you put on a stack is the first object that you pull off of it.

Don't worry — 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`** 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.)

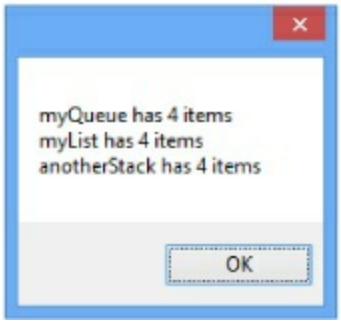
Let's set up a stack with four items—in this case, a stack of strings.

```
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);
MessageBox.Show("myQueue has " + myQueue.Count + " items\n"
    + "myList has " + myList.Count + " items\n"
    + "anotherStack has " + anotherStack.Count + " items\n");
```

All four items were copied into the new collections.



...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. Start with the `Lumberjack` class, filling in the missing code. Then design the form, and add the button event handlers to it.

- ❶ Here's the **`Lumberjack`** class. Fill in the get accessor for `FlapjackCount` and the `TakeFlapjacks` and `EatFlapjacks` methods.

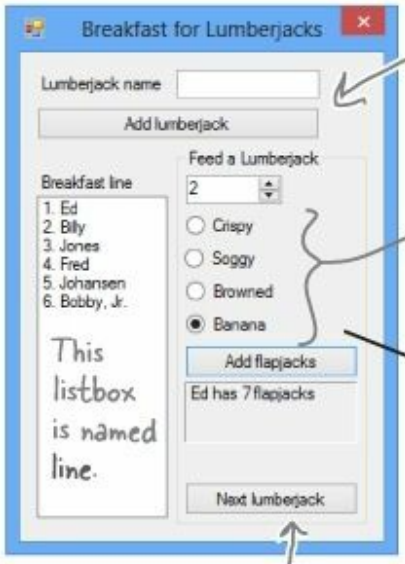
```
class Lumberjack {
    private string name;
    public string Name { get { return name; } }
    private Stack<Flapjack> meal;
    public Lumberjack(string name) {
        this.name = name;
        meal = new Stack<Flapjack>();
    }
    public int FlapjackCount { get { // return the count } }
    public void TakeFlapjacks(Flapjack food, int howMany) {
       // Add some number of flapjacks to the Meal stack
    }
    public void EatFlapjacks() {
       // Write this output to the console
    }
}
```

```
enum Flapjack {
    Crispy,
    Soggy,
    Browned,
    Banana
}
```

```
Output
Ed's eating flapjacks
Ed ate a banana flapjack
Ed ate a banana flapjack
Ed ate a crispy flapjack
Ed ate a soggy flapjack
Ed ate a soggy flapjack
Ed ate a soggy flapjack
Ed ate a browned flapjack
```

Notice how the `Flapjack` enum uses uppercase letters ("Soggy"), but the output has lowercase letters ("soggy")? Here's a hint to help you get the output right. `ToString()` returns a string object, and one of its public members is a method called `ToLower()` that returns a lowercase version of the string.

- ❷ Build this form. It lets you enter the names of lumberjacks into a text box so they get in the breakfast line. You can give the lumberjack at the front of the line a plate of flapjacks, and then tell him to move on to eat them using the "Next lumberjack" button. We've given you the click event handler for the "Add flapjacks" button. Use a queue called **`breakfastLine`** to keep track of the lumberjacks.



When the user clicks "Add lumberjack", add the name in the name text box to the breakfastLine queue.

When you drag these RadioButton controls into the group box, the form automatically links them and only allows the user to check one of them at a time. Look at the addFlapjacks_Click method to figure out what they should be named.

This button should dequeue the next lumberjack, call his EatFlapjacks(), then redraw the list box.

You'll need to add a RedrawList() method to update the list box with the contents of the queue. All three buttons will call it. **Here's a hint**: it uses a foreach loop.

```
private void addFlapjacks_Click(...) {
    if (breakfastLine.Count == 0) return;
    Flapjack food;
    if (crispy.Checked == true)
        food = Flapjack.Crispy;
    else if (soggy.Checked == true)
        food = Flapjack.Soggy;
    else if (browned.Checked == true)
        food = Flapjack.Browned;
    else
        food = Flapjack.Banana;

    Lumberjack currentLumberjack = breakfastLine.Peek();
    currentLumberjack.TakeFlapjacks(food,
                                    (int)howMany.Value);
    RedrawList();
}
```

Note the special "else if" syntax.

Peek() returns a reference to the first lumberjack in the queue.

The NumericUpDown control is called howMany, and the label is called nextInLine.

This program just prints lines to the console, so you need to open the Output window in the IDE to see the output.

EXERCISE SOLUTION

```
private Queue<Lumberjack> breakfastLine = new Queue<Lumberjack>();
private void addLumberjack_Click(object sender, EventArgs e) {
    if (String.IsNullOrEmpty(name.Text)) return;
    breakfastLine.Enqueue(new Lumberjack(name.Text));
    name.Text = "";
    RedrawList();
}
private void RedrawList() {
    int number = 1;
    line.Items.Clear();
    foreach (Lumberjack lumberjack in breakfastLine) {
        line.Items.Add(number + ". " + lumberjack.Name);
        number++;
    }
    if (breakfastLine.Count == 0) {
        groupBox1.Enabled = false;
        nextInLine.Text = "";
    } else {
        groupBox1.Enabled = true;
        Lumberjack currentLumberjack = breakfastLine.Peek();
        nextInLine.Text = currentLumberjack.Name + " has "
            + currentLumberjack.FlapjackCount + " flapjacks";
}   }
private void nextLumberjack_Click(object sender, EventArgs e) {
    if (breakfastLine.Count == 0) return;
    Lumberjack nextLumberjack = breakfastLine.Dequeue();
    nextLumberjack.EatFlapjacks();
    nextInLine.Text = "";
    RedrawList();
}

class Lumberjack {
    private string name;
    public string Name { get { return name; } }
    private Stack<Flapjack> meal;

    public Lumberjack(string name) {
        this.name = name;
        meal = new Stack<Flapjack>();
    }

    public int FlapjackCount { get { return meal.Count; } }

    public void TakeFlapjacks(Flapjack food, int howMany) {
        for (int i = 0; i < howMany; i++) {
            meal.Push(food);
    }   }

    public void EatFlapjacks() {
        Console.WriteLine(name + "'s eating flapjacks");
        while (meal.Count > 0) {
            Console.WriteLine(name + " ate a "
                + meal.Pop().ToString().ToLower() + " flapjack");
    }   }
}
```

We called the list box "line", and the label between the two buttons "nextInLine".

The RedrawList() method uses a foreach loop to pull the lumberjacks out of their queue and add each of them to the list box.

This if statement updates the label with information about the first lumberjack in the queue.

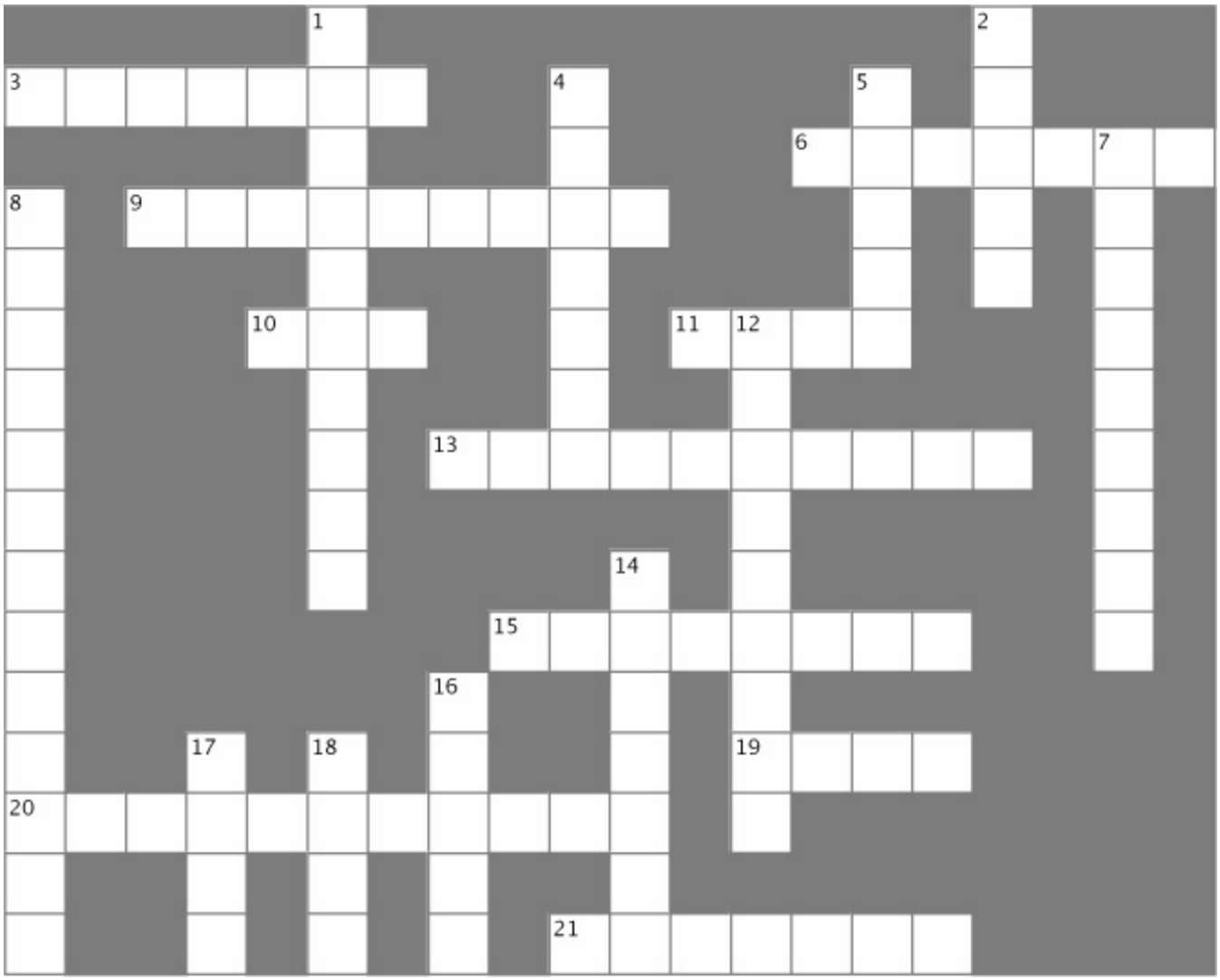
The TakeFlapjacks method updates the meal stack.

Here's where the Flapjack enum is made lowercase. Take a minute and figure out what's going on.

The EatFlapjacks method uses a while loop to print out the lumberjack's meal.

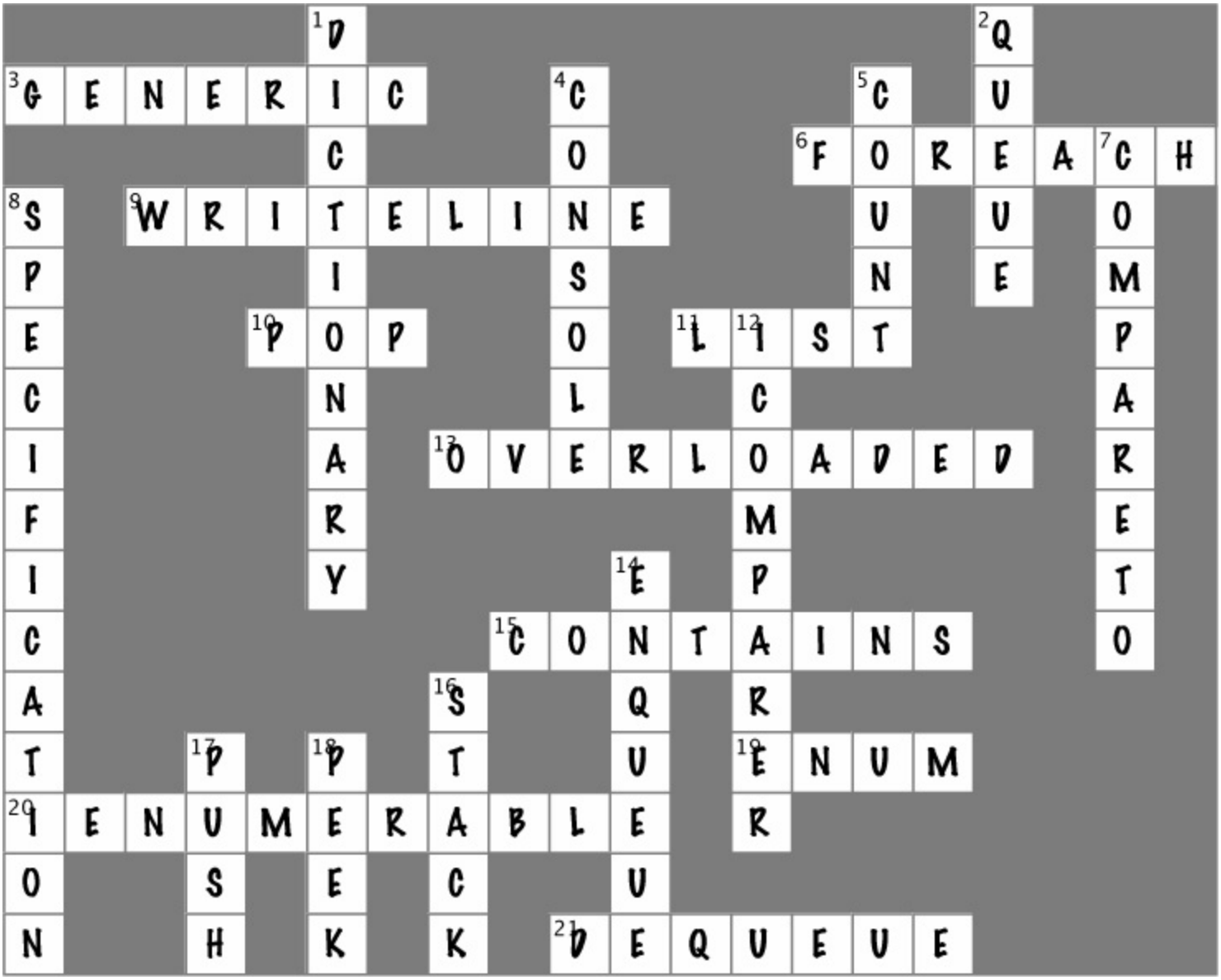
meal.Pop() returns an enum, whose ToString() method is called to return a string object, whose ToLower() method is called to return another string object.

COLLECTIONCROSS



Across	Down
3. An instance of a _____ collection only works with one specific type	1. The generic collection that lets you map keys to values
6. A special kind of loop that works on `IEnumerable<T>`	2. This collection is first-in, first-out
9. The name of the method you use to send a string to the output	4. The built-in class that lets your program write text to the output
10. How you remove something from a stack	5. A method to find out how many things are in a collection
11. An object that's like an array but more flexible	7. The only method in the IComparable interface
13. Two methods in a class with the same name but different parameters are _____	8. Most professional projects start with this
15. A method to figure out if a certain object is in a collection	12. An object that implements this interface helps your list sort its contents
19. An easy way to keep track of categories	14. How you add something to a queue
20. All generic collections implement this interface	16. This collection is first-in, last-out
21. How you remove something from a queue	17. How you add something to a stack
	18. This method returns the next object to come off of a stack or queue

COLLECTIONCROSS SOLUTION



Chapter 9. Reading and Writing Files: Save the last byte for me!

Sometimes it pays to be a little persistent.

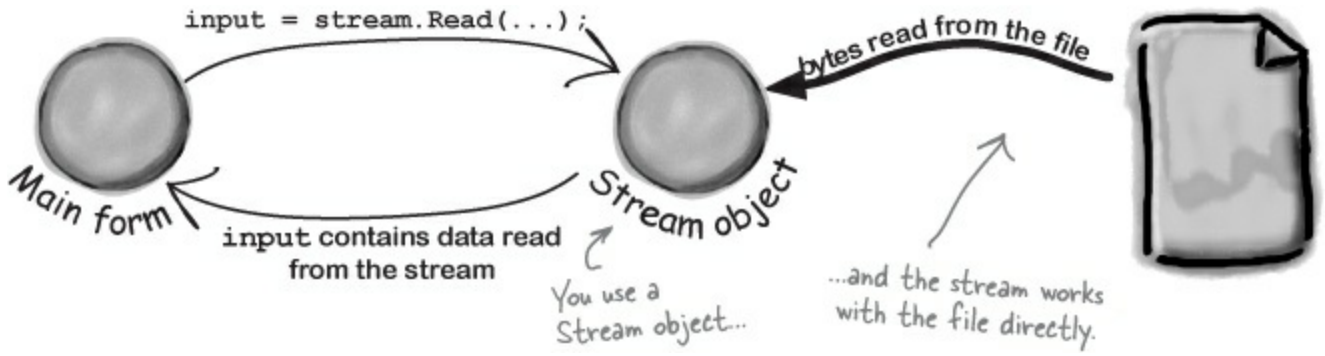
So far, all of your programs have been pretty short-lived. They fire up, run for a while, and shut down. But that's not always enough, especially when you're dealing with important information. You need to be able to **save your work**. In this chapter, we'll look at how to **write data to a file**,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

.NET uses streams to read and write data

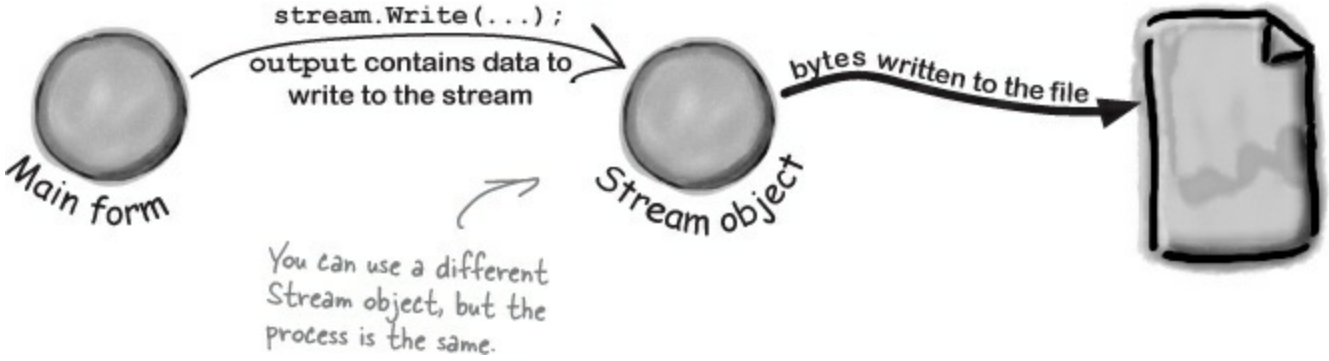
A **stream** is the .NET Framework's way of getting data in and out of your program. Any time your program reads or writes a file, connects to another computer over a network, or generally does anything where it **sends or receives bytes** from one place to another, you're using streams.

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 program — a form with an event handler that needs to read data from a file. You'll use a `Stream` object to do it.

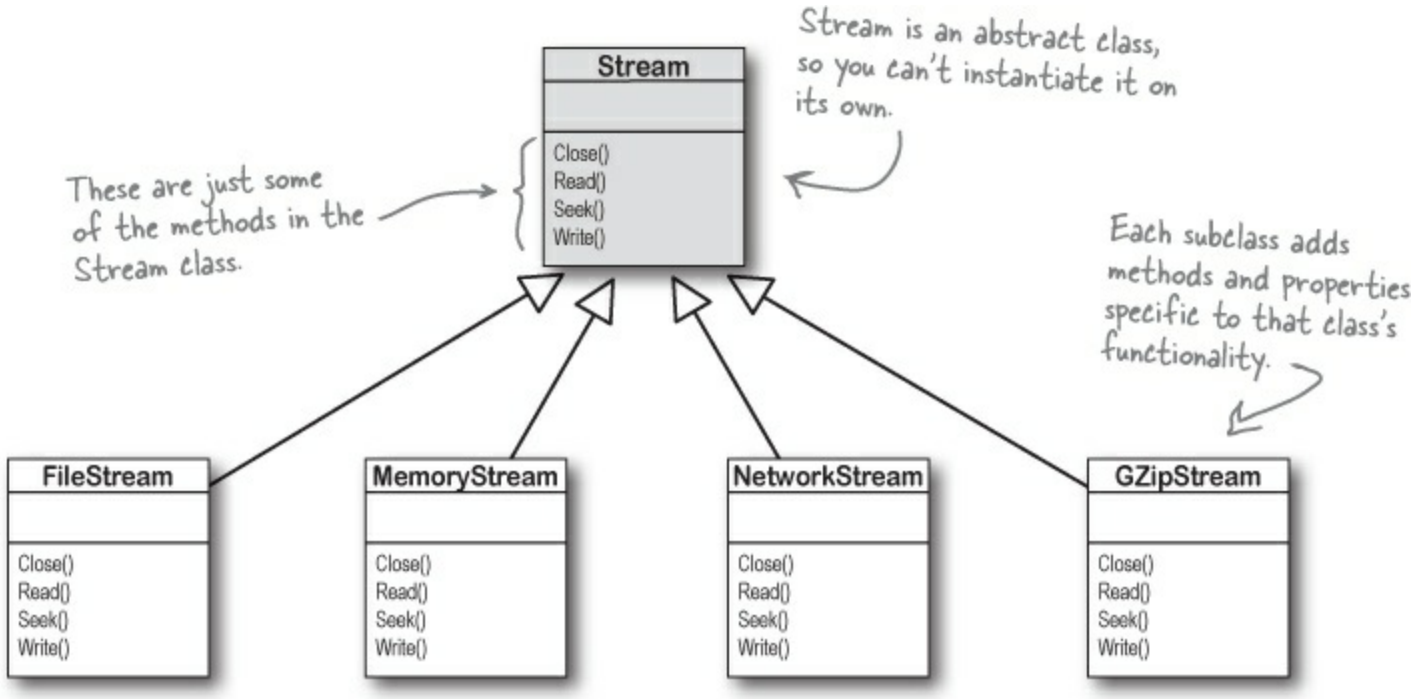


And if your program needs to write data out to the file, it can use another `Stream` object.



Different streams read and write different things

Every stream is a subclass of the abstract **`Stream`** class, and there are a bunch of built-in stream classes to do different things. We'll be concentrating on reading and writing regular files, but everything you learn in this chapter will just as easily apply to compressed or encrypted files, or network streams that don't use files at all.



A `FileStream` object lets you read from and write to files.

A `MemoryStream` object lets you read from and write data to chunks of memory.

A `NetworkStream` object lets you read and write data to other computers or devices on a network.

A `GZipStream` object lets you compress data so that it takes up less space and is easier to download and store.

Things you can do with a stream:

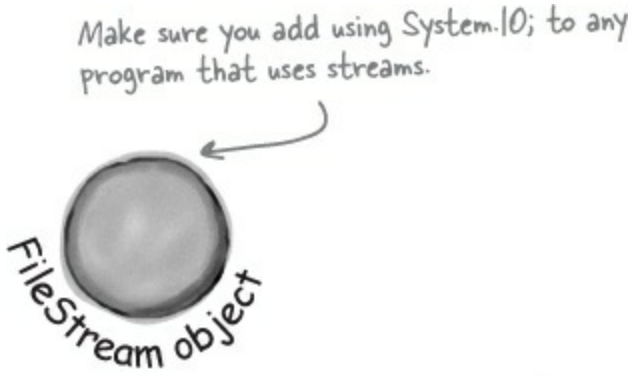
Streams let you read and write data. Use the right kind of stream for the data you're working with.

- **❶ WRITE TO THE STREAM.**
  You can write your data to a stream through a stream's `Write()` method.
- **❷ 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.
- **❸ 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.

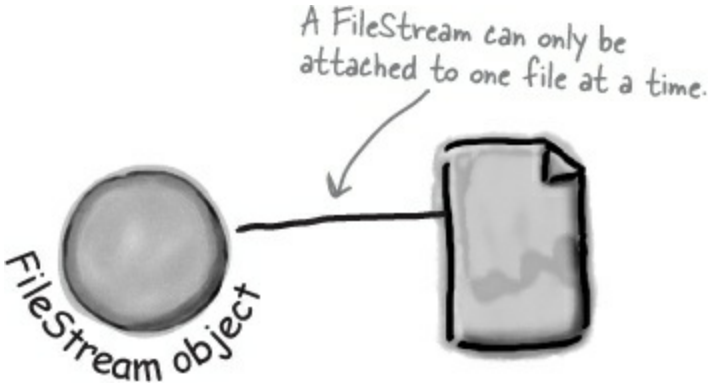
A FileStream reads and writes bytes to a file

When your program needs to write a few lines of text to a file, there are a lot of things that have to happen:

- ❶ Create a new `FileStream` object and tell it to write to the file.

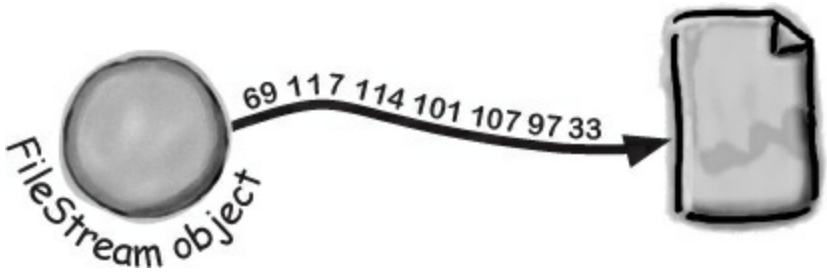


- ❷ The `FileStream` attaches itself to a file.



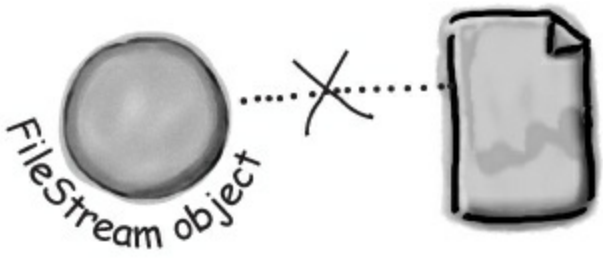
- ❸ Streams write bytes to files, so you'll need to convert the string that you want to write to an array of `bytes`.

- ❹ Call the stream's `Write()` method and pass it the `byte` array.



- ❺ Close the stream so other programs can access the file.

Forgetting to close a stream is a big deal. Otherwise, the file will be locked, and other programs won't be able to use it until you close your stream.



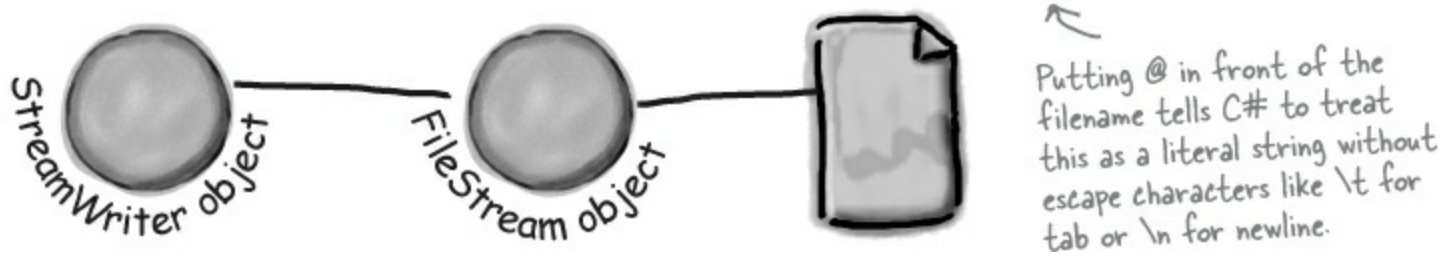
Write text to a file in three simple steps

C# comes with a convenient class called **`StreamWriter`** that 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.

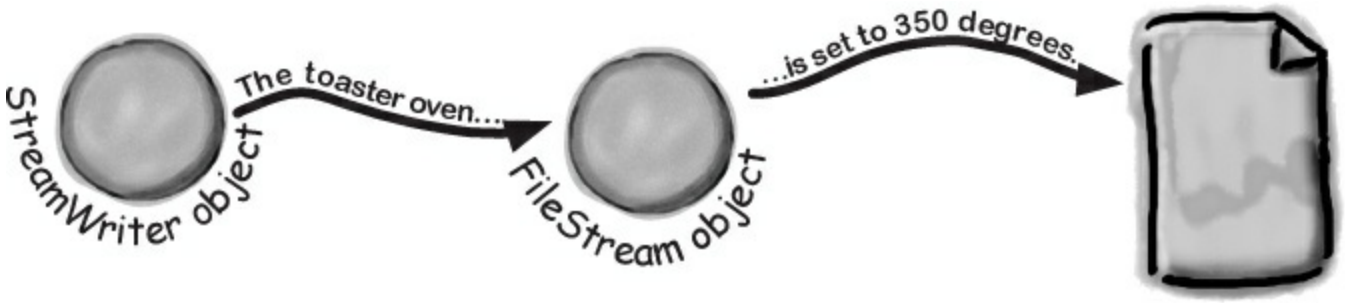
- **❶ Use the `StreamWriter`'s constructor to open or create a file.**
  You can pass a filename to the `StreamWriter()` constructor. When you do, the writer automatically opens the file. `StreamWriter` also has an overloaded constructor that lets you specify its *append* mode: passing it `true` tells 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.

```
StreamWriter writer = new StreamWriter(@"C:\newfiles\toaster oven.txt", true);
```



- **❷ 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. If you include "{0}", "{1}", "{2}", etc., inside the string you're writing, the methods include parameters in the strings being written: "{0}" is replaced with the first parameter after the string being written, "{1}" is replaced with the second, etc.

```
writer.WriteLine("The {0} is set to {1} degrees.", appliance, temp);
```



- **❸ 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. 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 to the `Main()` method:

It's probably not a good idea to write to your root folder, and your OS might not even let you do it. So pick another directory you want to write to.

The path starts with an @ sign so that the StreamWriter doesn't interpret the "\" as the start of an escape sequence.

This line creates the StreamWriter object and tells it where the file will be.

```
StreamWriter sw = new StreamWriter(@"C:\secret_plan.txt");
sw.WriteLine("How I'll defeat Captain Amazing");
sw.WriteLine("Another genius secret plan by The Swindler");
sw.Write("I'll create an army of clones and ");
sw.WriteLine("unleash them upon the citizens of Objectville.");
string location = "the mall";
for (int number = 0; number <= 6; number++){
    sw.WriteLine("Clone #{0} attacks {1}", number, location);
    if (location == "the mall") { location = "downtown"; }
    else { location = "the mall"; }
}
sw.Close();
```

WriteLine() adds a new line after writing. Write() sends just the text, with no extra line feeds at the end.

Can you figure out what's going on with the location variable in this code?

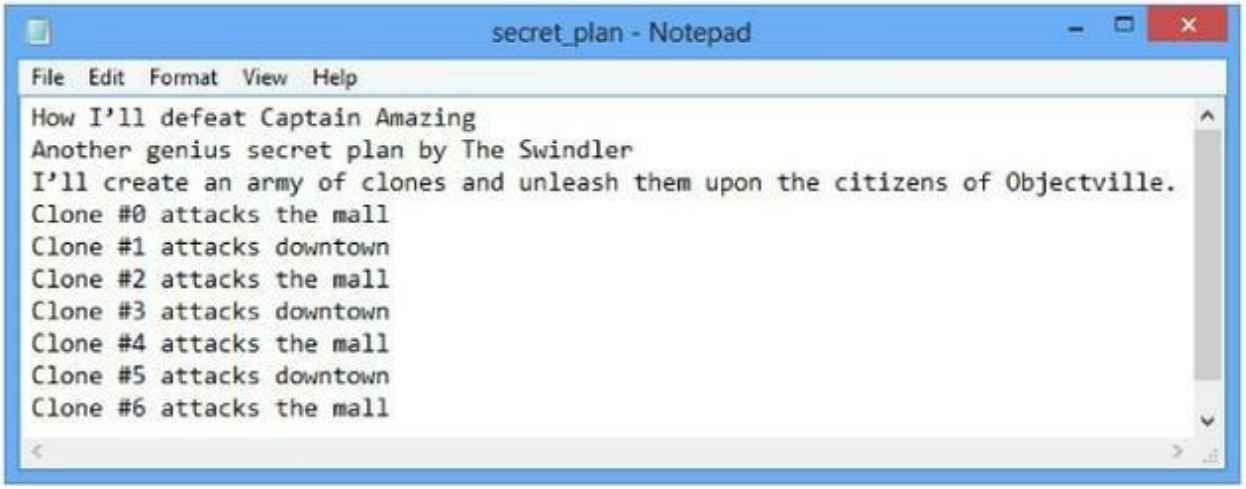
You can use the {} within the text to pass in variables to the string being written. {0} is replaced by the first parameter after the string, {1} by the second, and so on.

Close() frees up any connections to the file and any resources the StreamWriter is using. The text doesn't get written if you don't close the stream.

This is important!

StreamWriter is in the System.IO namespace, so make sure you add `using System.IO;` to the top of your program.

This is what the above code produces.



STREAMWRITER MAGNETS

Suppose you have the code for `button1_Click()` shown below. Your job is to use the magnets to build code for the `Flobbo` class so that when the event handler is called, it produces the output shown at the bottom of the page. Good luck!

```
static void Main(string[] args) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

```
sw.WriteLine(zap);
zap = "red orange";
return true;
```

```
}
```

```
sw.WriteLine(zap);
sw.Close();
return false;
```

```
public bool Blobbo
    (bool Already, StreamWriter sw) {
```

```
public bool Blobbo(StreamWriter sw) {
```

```
sw.WriteLine(zap);
zap = "green purple";
return false;
```

```
}
```

```
return new
    StreamWriter("macaw.txt");
```

```
}
```

```
}
```

```
}
```

```
private string zap;

public Flobbo(string zap) {
    this.zap = zap;
}
```

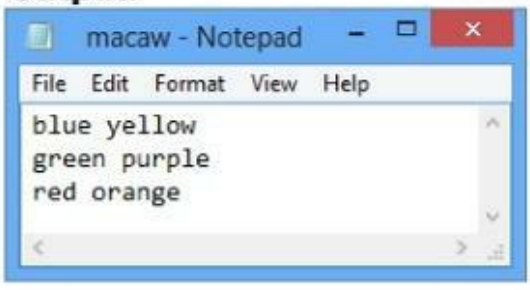
```
class Flobbo {
```

```
if (Already) {
```

```
} else {
```

```
public StreamWriter Snobbo() {
```

Output:



Assume all code files have `using System.IO;` at the top.

STREAMWRITER MAGNETS SOLUTION

NOTE

If you type this into the IDE, *macaw.txt* will be written to the *bin\Debug* folder inside your project folder, because that's where the executable is running.

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);
}
```

```
class Flobbo {
    private string zap;

    public Flobbo(string zap) {
        this.zap = zap;
    }

    public StreamWriter Snobbo() {
        return new
            StreamWriter("macaw.txt");
    }

    public bool Blobbo(StreamWriter sw) {
        sw.WriteLine(zap);
        zap = "green purple";
        return false;
    }

    public bool Blobbo
        (bool Already, StreamWriter sw) {
        if (Already) {
            sw.WriteLine(zap);
            sw.Close();
            return false;
        } else {
            sw.WriteLine(zap);
            zap = "red orange";
            return true;
        }
    }
}
```

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?

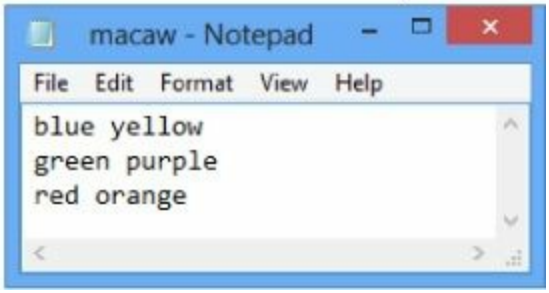
Assume all code files have `using System.IO;` at the top.

The Blobbo() method is overloaded—it's got two declarations with different parameters.

Make sure you close files when you're done with them.

If you run this code from the IDE, it creates macaw.txt in the bin\Debug folder.

Output:



NOTE

Sometimes people play a little fast and loose with the word "stream." A `StreamReader` (which inherits from `TextReader`) is a class that reads characters from streams. It's not a stream itself. When you pass a filename into its constructor, it creates a stream for you, and closes it when you call its `Close()` method. It's also got an overloaded constructor that takes a `Stream`. See how that works?

Reading and writing using two objects

Let's read Swindler's secret plans with a `StreamReader`. `StreamReader` works just like `StreamWriter`, except instead of writing a file you give the reader the name of the file to read in its constructor. The `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:

```
string folder =
    Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
StreamReader reader =
    new StreamReader(folder + @"\secret_plan.txt");
StreamWriter writer =
    new StreamWriter(folder + @"\emailToCaptainAmazing.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 plan:");
while (!reader.EndOfStream) {
    string lineFromThePlan = reader.ReadLine();
    writer.WriteLine("The plan -> " + lineFromThePlan);
}
writer.WriteLine();
writer.WriteLine("Can you help us?");
writer.Close();
reader.Close();
```

This returns the path of the user's My Documents folder. Check out the SpecialFolder enum to see what other folders you can find.

Pass the file you want to read from into the StreamReader's constructor. This time we're not writing to the C:\ folder! (Obviously, this file needs to exist in the My Documents folder.)

This program uses a StreamReader to read the Swindler's plan, and a StreamWriter to write a file that will get emailed to Captain Amazing.

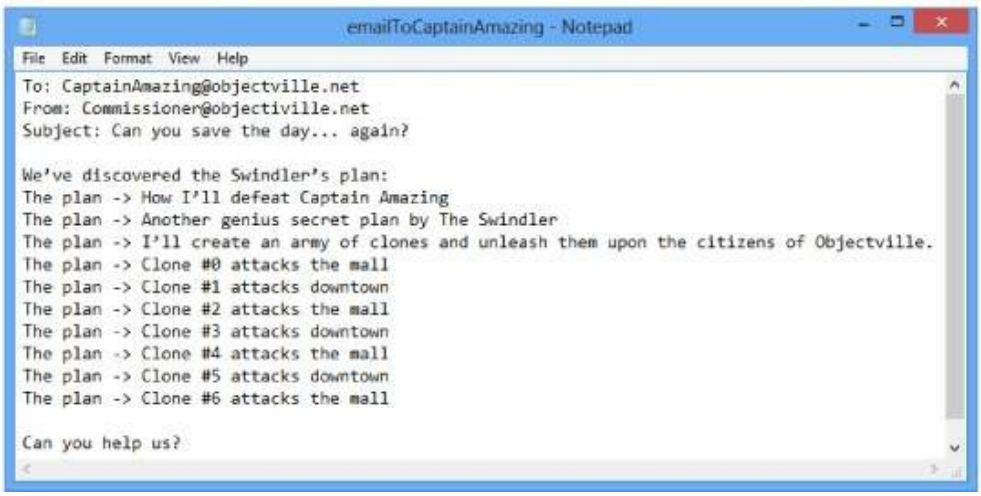
An empty WriteLine() method writes a blank line.

EndOfStream is the property that tells you if there's no data left unread in the file.

This loop reads a line from the reader and writes it out to the writer.

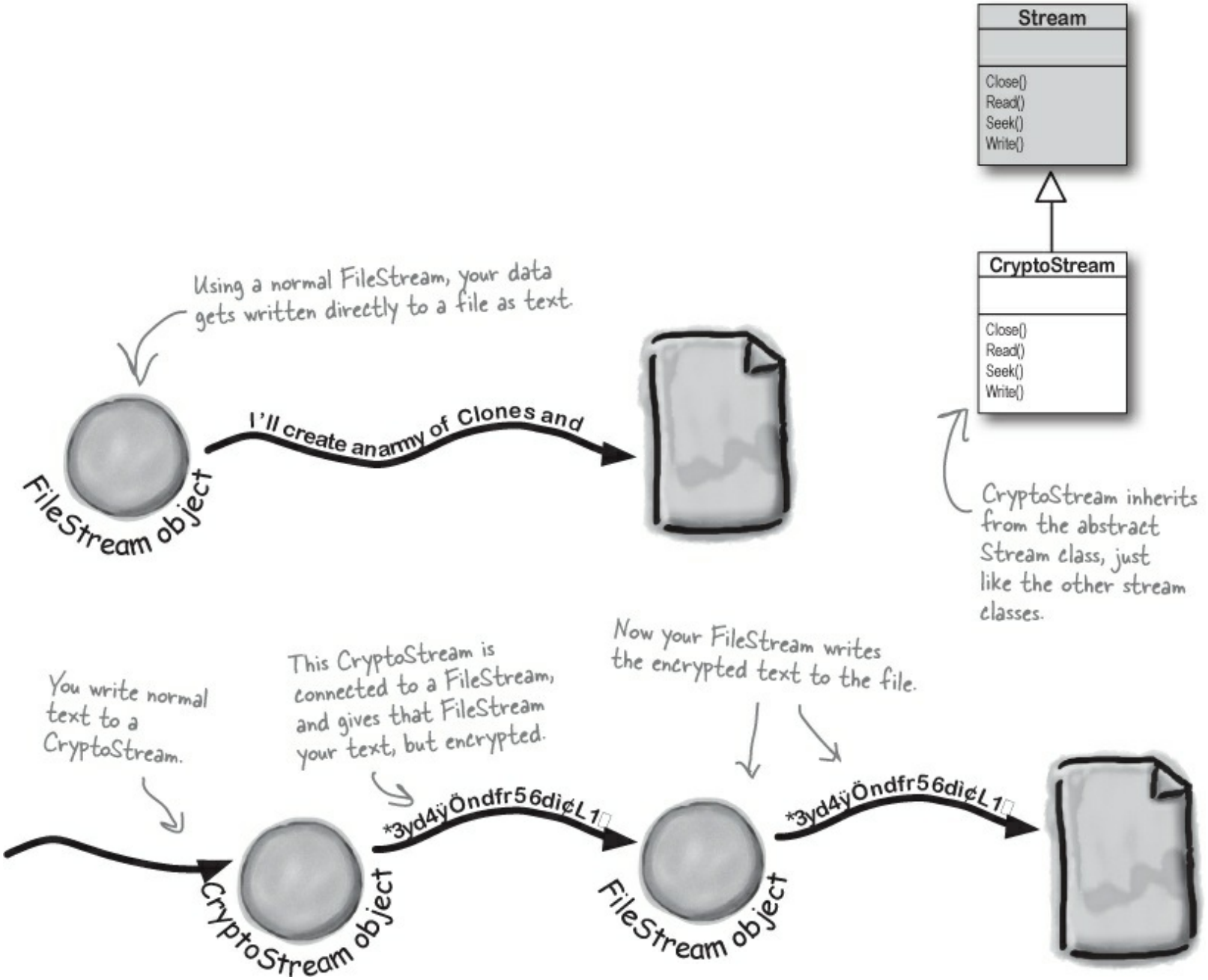
Make sure to close every stream that you open, even if you're just reading a file.

The StreamReader and StreamWriter opened up their own streams when you instantiated them. Calling their Close() methods tells them to close those streams.



Data can go through more than one stream

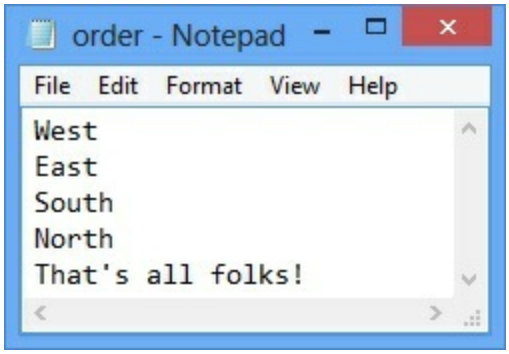
One big advantage to working with streams in .NET is that you can have your data go through more than one stream on its way to its final destination. One of the many types of streams that .NET ships with is the `CryptoStream` class. This lets you encrypt your data before you do anything else with it:



You can CHAIN streams. One stream can write to another stream, which writes to another stream...often ending with a network or file stream.

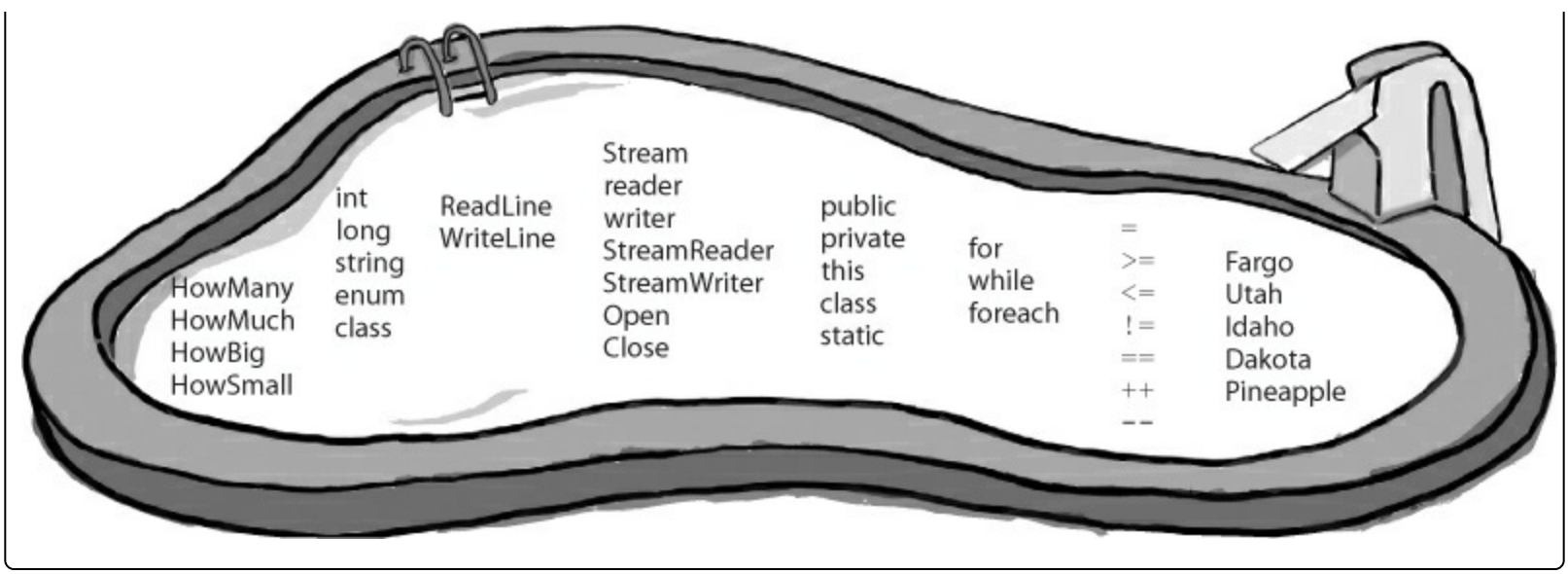
POOL PUZZLE

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.




```
class Pineapple {
  const _______ d = "delivery.txt";
  public ______  _______
        { North, South, East, West, Flamingo }
  public static void Main(string[] args) {
   __________ o = new _____________("order.txt");
   Pizza pz = new Pizza(new ___________(d, true));
   pz._________(Fargo.Flamingo);
   for (_____ w = 3; w >= 0; w--) {
     Pizza i = new Pizza
            (new ____________(d, false));
     i.Idaho((Fargo)w);
     Party p = new Party(new ___________(d));
     p._____________(o);
   }
   o._____________("That's all folks!");
   o._____________();
 }
}

class Pizza {
  private _____________ _______;
  public Pizza(___________  _______) {
    ______.writer = writer;
  }
  public void ______(______.Fargo f) {
    writer._________(f);
    writer._________();
  }
}

class Party {
  private _____________ reader;
  public Party(_____________ reader) {
    __________.reader = reader;
  }
  public void HowMuch(__________ q) {
    q._________(reader._________());
    reader._________();
  }
}
```

Note: each snippet from the pool can be used more than once!


HowMany
HowMuch
HowBig
HowSmall
int
long
string
enum
class
ReadLine
WriteLine
Stream
reader
writer
StreamReader
StreamWriter
Open
Close
public
private
this
class
static
for
while
foreach
=
>=
<=
!=
==
++
--
Fargo
Utah
Idaho
Dakota
Pineapple

POOL PUZZLE SOLUTION

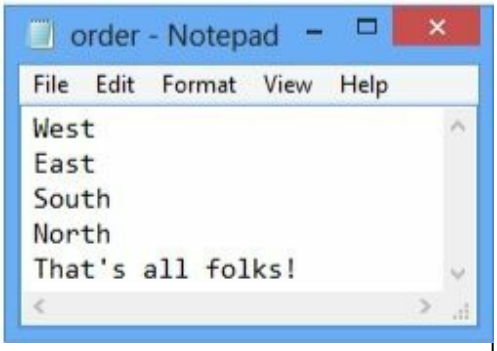
This enum (specifically, its ToString() method) is used to print a lot of the output.

```
class Pineapple {
    const string d = "delivery.txt";
    public enum Fargo { North, South, East, West, Flamingo }

    public static void Main(string[] args) {
        StreamWriter o = new StreamWriter("order.txt");
        Pizza pz = new Pizza(new StreamWriter(d, true));
        pz.Idaho(Fargo.Flamingo);
        for (int w = 3; w >= 0; w--) {
            Pizza 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();
    }
}

class Pizza {
    private StreamWriter writer;
    public Pizza(StreamWriter writer) {
        this.writer = writer;
    }
    public void Idaho(Pineapple.Fargo f) {
        writer.WriteLine(f);
        writer.Close();
    }
}

class Party {
    private StreamReader reader;
    public Party(StreamReader reader) {
        this.reader = reader;
    }
    public void HowMuch(StreamWriter q) {
        q.WriteLine(reader.ReadLine());
        reader.Close();
    }
}
```

Here's the 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.

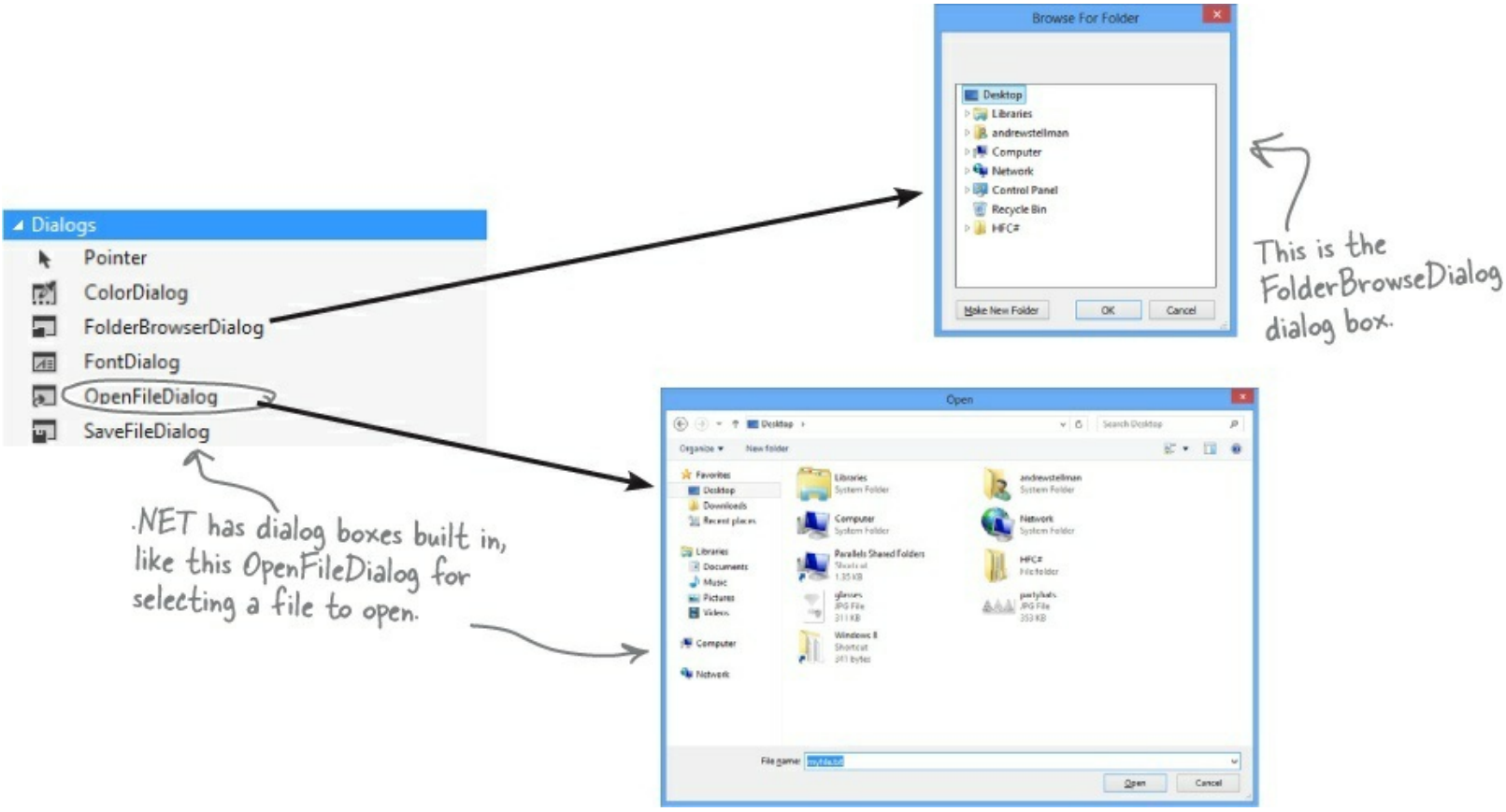
The Pizza class keeps a StreamWriter as a private field, and its Idaho() method writes Fargo enums to the file using their ToString() methods, which WriteLine() calls automatically.

The Party class has a StreamReader field, and its HowMuch() method reads a line from that StreamReader and writes it to a StreamWriter.



Use built-in objects to pop up standard dialog boxes

When you're working on a program that reads and writes files, there's a good chance that you'll need to pop up a dialog box at some point to prompt the user for a filename. That's why .NET for Windows Desktop includes objects to pop up the standard desktop file dialog boxes.



ShowDialog() pops up a dialog box

Displaying a dialog box is easy. Here's all you need to do:

- ❶ Create an instance of the dialog box object. You can do this in code using `new`, or you can drag it out of the toolbox and onto your form.

> **NOTE**
>
> We'll walk you through these steps in a minute.

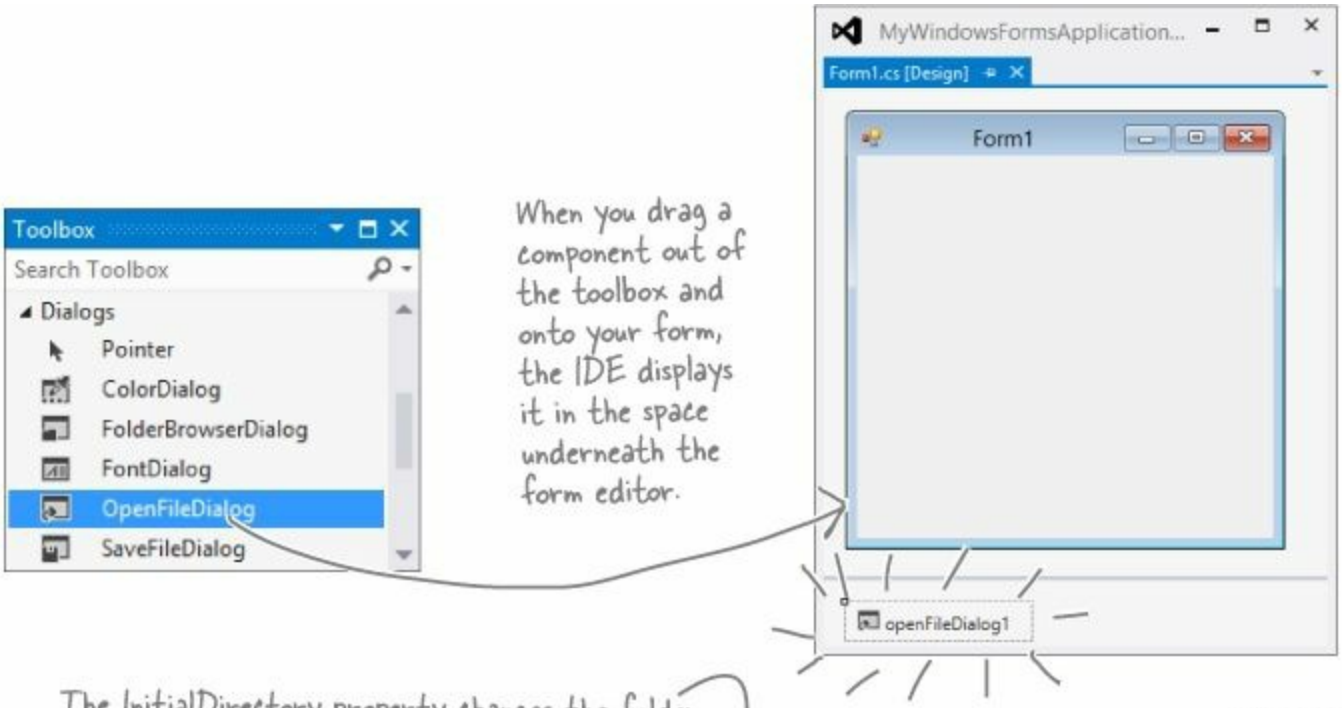
- ❷ Set the dialog box object's properties. A few useful ones include `Title` (which sets the text in the title bar), `InitialDirectory` (which tells it which directory to open first), and `FileName` (for Open and Save dialog boxes).
- ❸ Call the object's `ShowDialog()` method. That pops up the dialog box, and doesn't return until the user clicks the OK or Cancel button, or closes the window.
- ❹ The `ShowDialog()` method returns a `DialogResult`, which is an enum. Some of its members are `OK` (which means the user clicked OK), `Cancel`, `Yes`, and `No` (for Yes/No dialog boxes).

Dialog boxes are just another WinForms control

You can add Windows standard file dialog boxes to your program by dragging them to your form — just **drag an `OpenFileDialog` control out of the toolbox** and drop it onto your form. Instead of it showing up as a visual control, you'll see it appear in the space below your form. That's because it's a **component**, which is a special kind of **nonvisual toolbox control** that doesn't appear directly on the form, but which you can still use in your form's code just like you use any other control.

NOTE

"Nonvisual" just means it doesn't appear on your form when you drag it out of the toolbox.



The InitialDirectory property changes the folder that's first displayed when the dialog opens.

The Filter property lets you change the filters that show up on the bottom of the dialog box, such as what types of files to show.

```
openFileDialog1.InitialDirectory = @"c:\MyFolder\Default\";
openFileDialog1.Filter = "Text Files (*.txt)|*.txt|"
  + "Comma-Delimited Files (*.csv)|*.csv|All Files (*.*)|*.*";
openFileDialog1.FileName = "default_file.txt";
openFileDialog1.CheckFileExists = true;
openFileDialog1.CheckPathExists = false;
DialogResult result = openFileDialog1.ShowDialog();
if (result == DialogResult.OK){
    OpenSomeFile(openFileDialog1.FileName);
}
```

These properties tell the dialog box to display an error message if the user tries to open up a file or path that doesn't exist on the drive.

Display the dialog box using its ShowDialog() method, which returns a DialogResult. That's an enum that you can use to check whether or not the user hit the OK button. It'll be set to DialogResult.OK if the user clicked OK, and DialogResult.Cancel if he hit Cancel.

Dialog boxes are objects, too

An **`OpenFileDialog`** object shows the standard Windows Open window, and the **`SaveFileDialog`** shows the Save window. You can display them by creating a `new` instance, setting the properties on the object, and calling its `ShowDialog()` method. The `ShowDialog()` method returns a `DialogResult` enum (because some dialog boxes have more than two buttons or results, so a simple `bool` wouldn't be enough).

When you drag a save dialog object out of the toolbox and onto your form, the IDE just adds a line like this to your form's InitializeComponent() method.

```
saveFileDialog1 = new SaveFileDialog();
saveFileDialog1.InitialDirectory = @"c:\MyFolder\Default\";
saveFileDialog1.Filter = "Text Files (*.txt)|*.txt|"
  + "Comma-Delimited Files (*.csv)|*.csv|All Files (*.*)|*.*";
DialogResult result = saveFileDialog1.ShowDialog();
if (result == DialogResult.OK){
    SaveTheFile(saveFileDialog1.FileName);
}
```

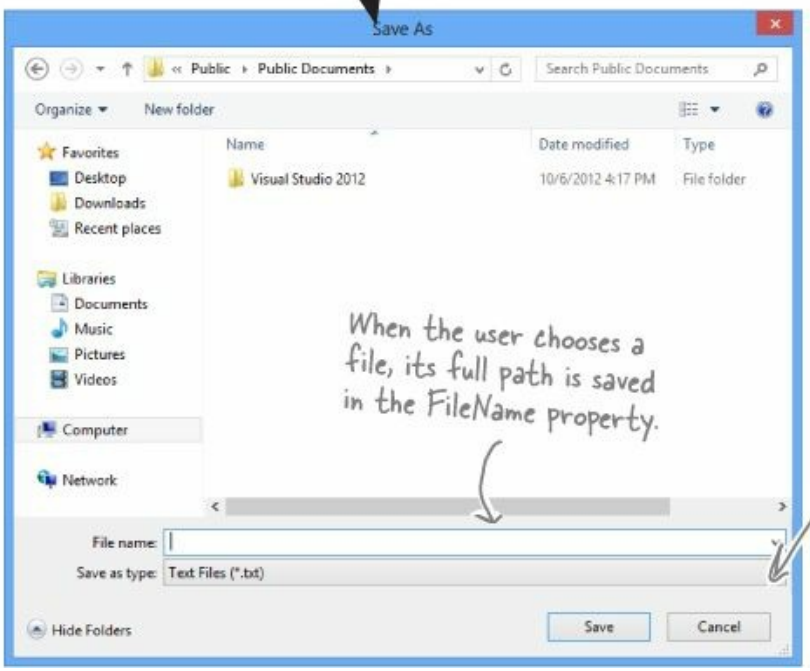
The Filter property isn't hard to figure out. Just compare what's between the | characters in the string with what shows up in the window.

The ShowDialog() and FileName properties work exactly the same as on the OpenFileDialog object.

This assumes that there's a method in the program called SaveTheFile() that takes a filename as a parameter.

The SaveFileDialog object pops up the standard Windows "Save as..." dialog box.

The Title property lets you change this text.



The ShowDialog() method pops up the dialog box and opens the folder specified in the InitialDirectory property.

When the user chooses a file, its full path is saved in the FileName property.

Change the "Save as type" list using the **Filter** property.

The DialogResult returned by the ShowDialog() method lets you figure out which button the user clicked.

Use the built-in 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 `FileStream`s first. `Directory` objects let you work with whole directories full of files.

Things you can do with File:

- **❶ 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.
- **❷ 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.
- **❸ 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.
- **❹ GET INFORMATION ABOUT THE FILE.**
  The `GetLastAccessTime()` and `GetLastWriteTime()` methods return the date and time when the file was last accessed and modified.

FILEINFO WORKS JUST LIKE FILE

If you're going to be doing a lot of work with a file, you might want to create an instance of the FileInfo class instead of using the File class's static methods.

The FileInfo class does just about everything the File class does, except you have to instantiate it to use it. You can create a new instance of FileInfo and access its Exists() method or its OpenRead() method in just the same way.

The only difference is that the File class is faster for a small number of actions, and FileInfo is better suited for big jobs.

NOTE

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.

Things you can do with Directory:

- ❶ **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.
- ❷ **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.
- ❸ **DELETE A DIRECTORY.**
  Deleting a directory is really simple too. Just use the `Delete()` method.

THERE ARE NO DUMB QUESTIONS

Q: **Q: I still don't get that {0} and {1} thing that was part of the `StreamWriter`.**

A: **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 take advantage of {0} and {1}:

```
writer.WriteLine(
  "My name is {0} and my age is {1}",
   name, age);
```

It's a lot easier to read that code, especially when many variables are included in the same line.

Q: **Q: Why did you put an @ in front of the string that contained the filename?**

A: **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:

```
string twoLine = @"this is a string
that spans two lines.";
```

Q: **Q: And what do `\n` and `\t` mean again?**

A: **A:** Those are escape sequences. `\n` is a line feed and `\t` is a tab. `\r` is a return character, or half of a Windows return — in Windows text files, lines have to end with `\r\n` (like we talked about when we introduced `Environment.NewLine` from Chapter 8). 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: **Q: What was that in the beginning about converting a string to a byte array? How would that even work?**

A: **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: **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: **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: **Q: What if I want to create my own dialog boxes? Can I do that?**

A: **A:** Yes, you definitely can. You can add a new form to your project and design it to look exactly how you want. Then you can create a new instance of it with `new` (just like you created an `OpenFileDialog` object). Then you can add a public `ShowDialog()` method, and it'll work just like any other dialog box.

Q: **Q: Why do I need to worry about closing streams after I'm done with them?**

A: **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.

SHARPEN YOUR PENCIL

.NET has two built-in classes with a bunch of static methods for working with files and folders. 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.

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(1976, 09, 25));`	
`string[] files = Directory.GetFiles(@"C:\Windows\",` `  "*.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");` See if you can guess what this one does—you haven't seen it yet.	
`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");`	

Use file dialogs to open and save files (all with just a few lines of code)

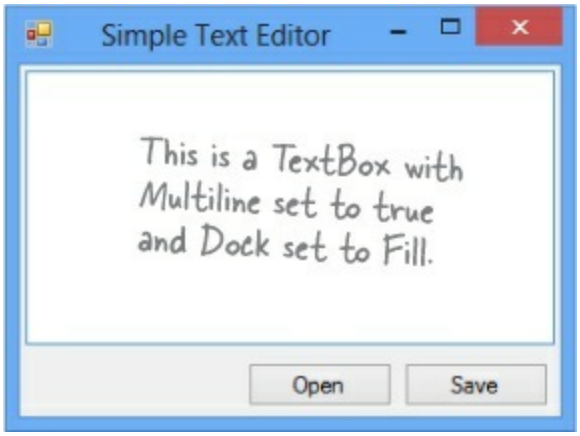
You can build a program that opens a text file. It'll let you make changes to the file and save your changes, with very little code, all using standard .NET controls. Here's how:

NOTE

Here's a trick to make your TextBox fill up the form. Drag a TableLayoutPanel from the toolbox (in Containers) onto the form, set its `Dock` property to `Fill`, and use its Rows and Columns property editors to give it two rows and one column. Drag the TextBox into the top cell and set its `Dock` property to `Fill`. Then drag a FlowLayoutPanel out of the toolbox into the bottom cell, set its `Dock` to `Fill`, set its `FlowDirection` property to `RightToLeft`, and drag the two buttons onto it. Set the size of the bottom row in the TableLayoutPanel to `AutoSize` and the top row to 100%, and resize the bottom row so that the two buttons just fit. Now your editor will resize smoothly!

DO THIS

- **❶ Build a simple form.**
  All you need is a `TextBox` and two `Button`s. Drop the `OpenFileDialog` and `SaveFileDialog` controls onto the form, too. Double-click on the buttons to create their event handlers and **add a private string field called `name` to the form**. Don't forget to put a `using` statement up top for the `System.IO` namespace.



- **❷ Hook the Open button up to the `OpenFileDialog` object.**
  The `Open` button shows an `OpenFileDialog` and then uses `File.ReadAllText()` to read the file into the textbox:

```
private void open_Click(object sender, EventArgs e) {
  if (openFileDialog1.ShowDialog() == DialogResult.OK) {
    name = openFileDialog1.FileName;
    textBox1.Clear();
    textBox1.Text = File.ReadAllText(name);
  }
}
```

NOTE

Clicking Open shows the OpenFileDialog control.

- **❸ Now, hook up the Save button.**
  The `Save` button uses the `File.WriteAllText()` method to save the file:

```
private void save_Click(object sender, EventArgs e) {
  if (saveFileDialog1.ShowDialog() == DialogResult.OK) {
    string name = saveFileDialog1.FileName;
    File.WriteAllText(name, textBox1.Text);
  }
}
```

NOTE

The ReadAllText() and WriteAllText() methods are part of the File class. That's coming up on the next page. We'll look at them in more detail in just a few pages.

- **❹ Play with the other properties of the dialog boxes.**

  Use the `Title` property of the `SaveFileDialog` to change the text in the title bar.
  Set the `InitialDirectory` property to have the `OpenFileDialog` start in a specified directory.
  Filter the `OpenFileDialog` so it will only show text files using the `Filter` property.

  NOTE

  — If you don't add a filter, then the drop-down lists at the bottom of the open and save dialog boxes will be empty. Try using this filter: "Text Files (*.txt)|*.txt".

SHARPEN YOUR PENCIL SOLUTION

.NET has two built-in classes with a bunch of static methods for working with files and folders. 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

`if (!Directory.Exists(@"C:\SYP")) {`
`  Directory.CreateDirectory(@"C:\SYP");`
`}`
	Check if the C:\SYP folder exists. If it doesn't, create it.

`if (Directory.Exists(@"C:\SYP\Bonk")) {`
`  Directory.Delete(@"C:\SYP\Bonk");`
`}`
	Check if the C:\SYP\Bonk folder exists. If it does, delete it.

`Directory.CreateDirectory(@"C:\SYP\Bonk");`	Create the directory C:\SYP\Bonk.
`Directory.SetCreationTime(@"C:\SYP\Bonk",` `  new DateTime(1976, 09, 25));`	Set the creation time for the C:\SYP\Bonk folder to September 25, 1976.

`string[] files =`
`Directory.GetFiles(@"C:\Windows\",`
`  "*.log",`
`SearchOption.AllDirectories);`
	Get a list of all files in C:\Windows that match the *.log pattern, including all matching files in any subdirectory.

`File.WriteAllText(@"C:\SYP\Bonk\weirdo.txt",`
`  @"This is the first line`
`and this is the second line`
`and this is the last line");`
	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.

`File.Encrypt(@"C:\SYP\Bonk\weirdo.txt");` This is an alternative to using a CryptoStream.	Take advantage of built-in Windows encryption to encrypt the file "weirdo.txt" using the logged-in account's credentials.
`File.Copy(@"C:\SYP\Bonk\weirdo.txt",` `  @"C:\SYP\copy.txt");`	Copy the C:\SYP\Bonk\weirdo.txt file to C:\SYP\Copy.txt.

`DateTime myTime =`
 	
`Directory.GetCreationTime(@"C:\SYP\Bonk");`
	Declare the myTime variable and set it equal to the creation time of the C:\SYP\Bonk folder.

`File.SetLastWriteTime(@"C:\SYP\copy.txt",` `myTime);`	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.
`File.Delete(@"C:\SYP\Bonk\weirdo.txt");`	Delete the C:\SYP\Bonk\weirdo.txt file.

IDisposable makes sure your objects are disposed of 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.

You can use the "Go To Definition" feature in the IDE to show you the official C# 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:

Declare an object in a using block and that object's Dispose() method is called automatically.

```
namespace System
{
    // Summary:
    //     Defines a method to release allocated resources.
    public interface IDisposable
    {
        // Summary:
        //     Performs application-defined tasks
        //     associated with freeing, releasing, or
        //     resetting unmanaged resources.
        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 will 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.

GO TO DEFINITION

There's a handy feature in the IDE that lets you automatically jump to the definition for any variable, object, or method. Just right-click on it and select "Go To Definition," and the IDE will automatically jump right to the code that defines it. You can also press F12 instead of using the menu.

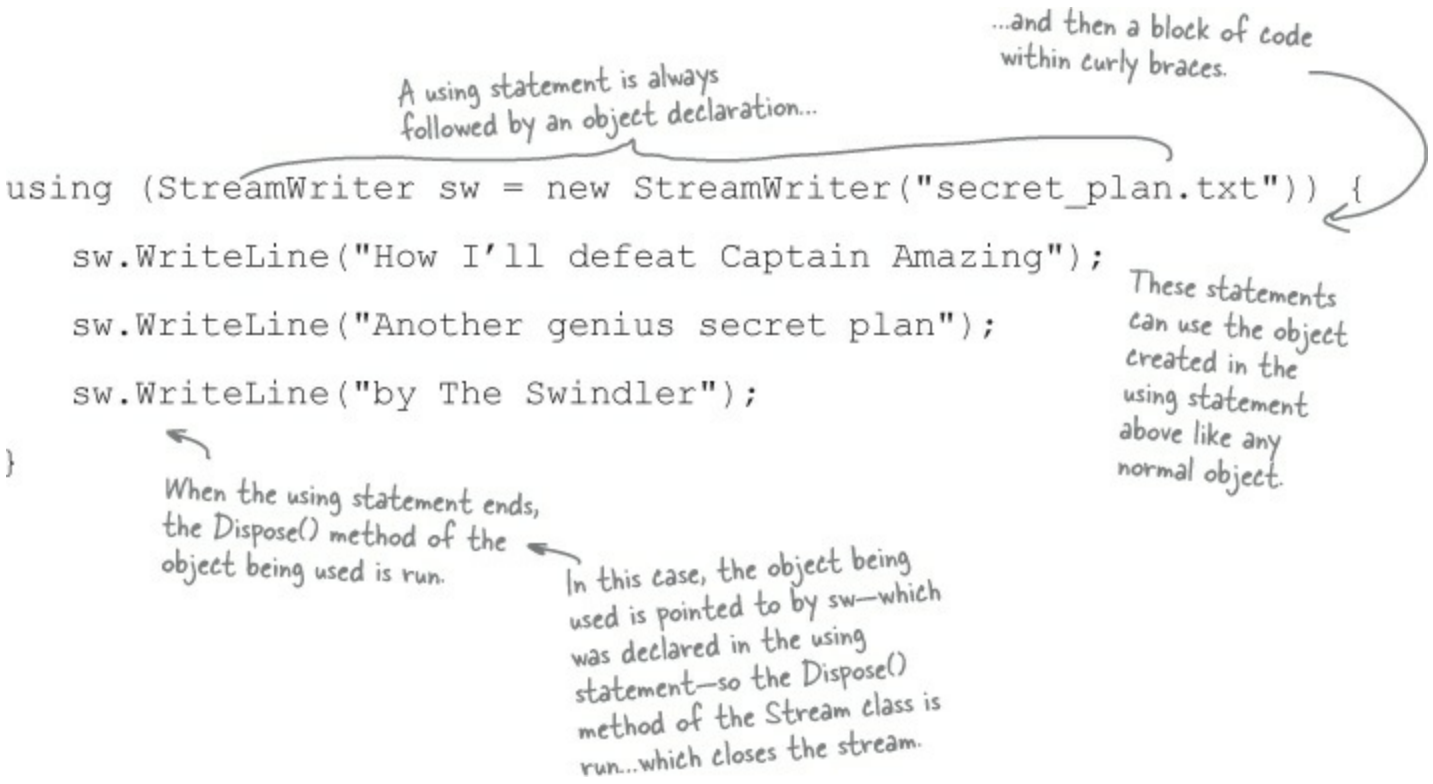
AL-LO-CATE, VERB.

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, the `using` statement **automatically calls the stream's `Dispose()` method** as soon as it finishes running the block of code. Here's how it works:

> **NOTE**
>
> These "using" statements are different from the ones at the top of your code.



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!

Every stream has a `Dispose()` method that closes the stream. So if you declare your stream in a using statement, it will always close itself!

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 (StreamReader reader = new StreamReader("secret_plan.txt"))
using (StreamWriter writer = new StreamWriter("email.txt"))
{
    // statements that use reader and writer
}
```

NOTE

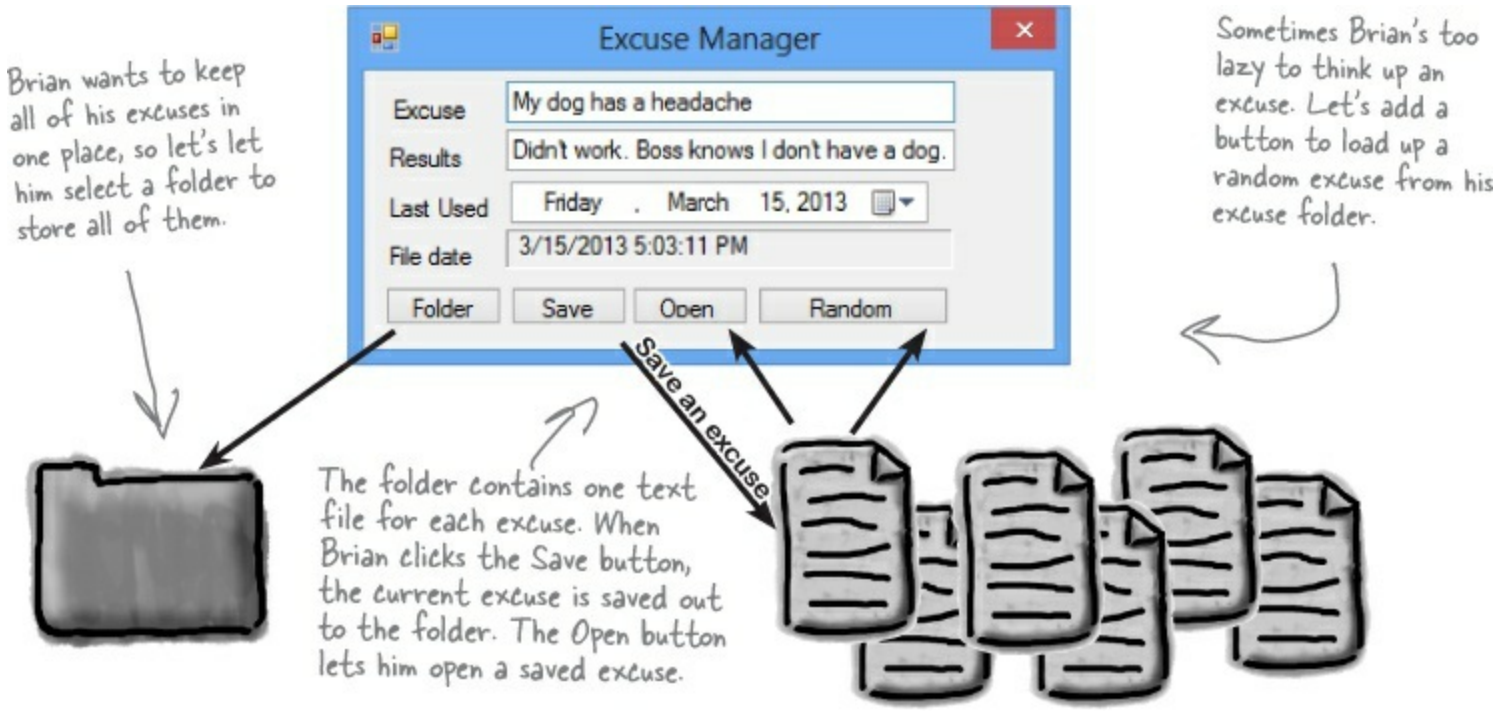
You don't need to call Close() on the streams now, because the using statement will close them automatically.

Trouble at work

Meet Brian. He likes his job as a C# developer, but he *loves* taking the occasional day off. His boss **hates** when people take vacation days, so Brian's got to come up with a good excuse.

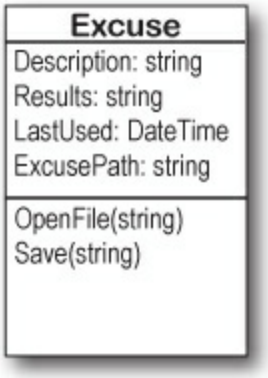
You can help Brian out by building a program to manage his excuses

Use what you know about reading and writing files to build an Excuse Manager that Brian can use to keep track of which excuses he's used recently and how well they went over with the boss.



EXERCISE

Build the Excuse Manager so Brian can manage his excuses at work.



- **❶ BUILD THE FORM.**
  This form has a few special features:

  When the form's first loaded, **only the Folder button should be enabled** — disable the other three buttons until the user selects a folder.
  When the form opens or saves an excuse, it displays the file date for the excuse file using a `Label` control with `AutoSize` set to `False` and `BorderStyle` set to `Fixed3D`.
  After an excuse is saved, the form pops up an "Excuse Written" message box.
  The Folder button brings up a folder browser dialog box. If the user selects a folder, it enables the Save, Open, and Random buttons.
  The form knows when there are unsaved changes. When there are no unsaved changes, the text on the form's title bar is "Excuse Manager". But when the user has changed any of the three fields, the form adds an asterisk (*) to the title bar. The asterisk goes away when the data is saved or a new excuse is opened.
  The form will need to keep track of the current folder and whether or not the current excuse has been saved. You can figure out when the excuse hasn't been saved by **using the `Changed` event handlers** for the three input controls.

NOTE

— When you drag a textbox to a form and double-click on it, you create a Changed event handler for that field.

- **❷ CREATE AN EXCUSE CLASS AND STORE AN INSTANCE OF IT IN THE FORM.**
  Now add a `currentExcuse` field to the form to hold the current excuse. You'll need **three overloaded constructors**: one for when the form's first loaded, one for opening up a file, and one for a random excuse. Add methods `OpenFile()` to open an excuse (for the constructors to use), and `Save()` to save the excuse. Then add this `UpdateForm()` method to update the controls (it'll give you some **hints** about the class):

```
private void UpdateForm(bool changed) {
  if (!changed)  {
    this.description.Text = currentExcuse.Description;
    this.results.Text = currentExcuse.Results;
    this.lastUsed.Value = currentExcuse.LastUsed;
    if (!String.IsNullOrEmpty(currentExcuse.ExcusePath))
      fileDate.Text = File.GetLastWriteTime(currentExcuse.ExcusePath).ToString();
      this.Text = "Excuse Manager";
  }
  else
      this.Text = "Excuse Manager*";
  this.formChanged = changed;
}
```

This parameter indicates whether or not the form has changed. You'll need a field in your form to keep track of this status.

Remember, the ! means NOT—so this checks if the excuse path is NOT null or empty.

Double-click on the input controls so the IDE builds Changed event handlers for you. The event handlers for the three input controls will first change the Excuse instance and then call UpdateForm(true)—then it's up to you to change the fields on your form.

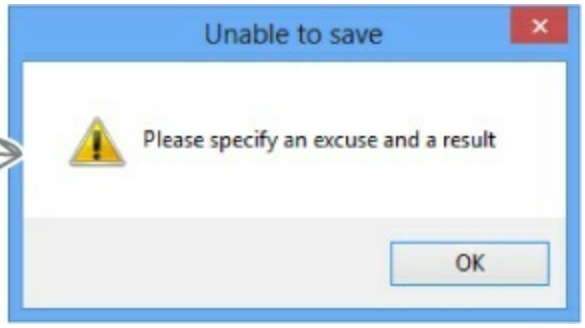
And make sure you initialize the excuse's `LastUsed` value in the form's constructor: `public Form1() { InitializeComponent();`

```
public Form1() {
    InitializeComponent();
    currentExcuse.LastUsed = lastUsed.Value;
}
```

- **❸ MAKE THE FOLDER BUTTON OPEN A FOLDER BROWSER.**
  When the user clicks on the Folder button, the form should pop up a "Browse for Folder" dialog box. The form will need to store the folder in a field so that the other dialog boxes can use it. When the form **first loads**, the Save, Open, and Random Excuse buttons are **disabled**, but if the user selects a folder, then the Folder button enables them.
- **❹ MAKE THE SAVE BUTTON SAVE THE CURRENT EXCUSE TO A FILE.**
  Clicking the Save button should bring up the Save As dialog box.

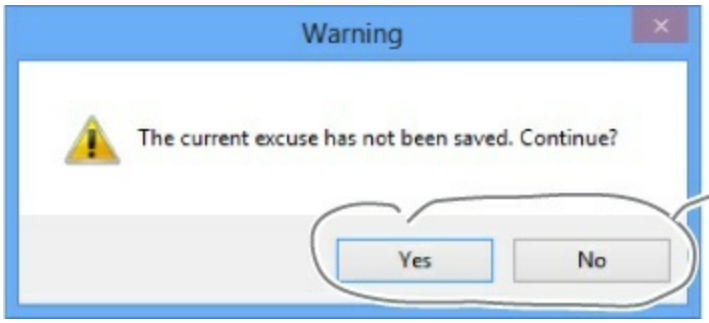
  Each excuse is saved to a separate text file. The first line of the file is the excuse, the second is the result, and the third is the date last used (using the `DateTimePicker`'s `ToString() method`). The `Excuse` class should have a `Save()` method to save an excuse out to a specified file.
  When the Save As dialog box is opened, its folder should be set to the folder that the user selected using the Folder button, and the filename should be set to the excuse plus a *.txt* extension.
  The dialog box should have two filters: Text Files (*.txt) and All Files (*.*). If the user tries to save the current excuse but has left either the excuse or the result blank, the form should pop up a warning dialog box:

You can display this Exclamation icon by using the overloaded MessageBox.Show() method that allows you to specify a MessageBoxIcon parameter.



- **❺ MAKE THE OPEN BUTTON OPEN A SAVED EXCUSE.**
  Clicking the Open button should bring up the Open dialog box.

When the Open dialog box is opened, its folder should be set to the folder that the user selected using the Folder button.
Add an `Open()` method to the `Excuse` class to open an excuse from a given file.
Use `Convert.ToDateTime()` to load the saved date into the `DateTimePicker` control.
If the user tries to open a saved excuse but the current excuse hasn't been saved, it pops up this dialog box:



Show a Yes/No dialog box by using the overloaded MessageBox.Show() method that lets you specify the MessageBoxButtons.YesNo parameter. If the user clicks No, then Show() returns DialogResult.No.

- **❼ FINALLY, MAKE THE RANDOM EXCUSE BUTTON LOAD A RANDOM EXCUSE.**
  When the user clicks the Random Excuse button, it looks in the excuse folder, chooses one of the excuses at random, and opens it.

  The form will need to save a `Random` object in a field and pass it to one of the overloaded constructors of the `Excuse` object.
  If the current excuse hasn't been saved, the button should pop up the same warning dialog box as the Open button.

EXERCISE SOLUTION

Build the Excuse Manager so Brian can manage his excuses at work.

```
private Excuse currentExcuse = new Excuse();
private string selectedFolder = "";
private bool formChanged = false;
Random random = new Random();
```

The form uses fields to store the current Excuse object to the selected folder and remember whether or not the current excuse has changed, and to keep a Random object for the Random Excuse button.

```
private void folder_Click(object sender, EventArgs e) {
    folderBrowserDialog1.SelectedPath = selectedFolder;
    DialogResult result = folderBrowserDialog1.ShowDialog();
    if (result == DialogResult.OK) {
        selectedFolder = folderBrowserDialog1.SelectedPath;
        save.Enabled = true;
        open.Enabled = true;
        randomExcuse.Enabled = true;
    }
}
```

If the user selected a folder, the form saves the folder name and then enables the other three buttons.

The two vertical bars mean OR—this is true if description is empty OR results is empty.

```
private void save_Click(object sender, EventArgs e) {
    if (String.IsNullOrEmpty(description.Text) || String.IsNullOrEmpty(results.Text)) {
        MessageBox.Show("Please specify an excuse and a result",
             "Unable to save", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        return;
    }
    saveFileDialog1.InitialDirectory = selectedFolder;
    saveFileDialog1.Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*";
    saveFileDialog1.FileName = description.Text + ".txt";
    DialogResult result = saveFileDialog1.ShowDialog();
    if (result == DialogResult.OK) {
        currentExcuse.Save(saveFileDialog1.FileName);
        UpdateForm(false);
        MessageBox.Show("Excuse written");
    }
}
```

Here's where the filters are set for the Save As dialog.

This will cause two rows to show up in the "Files of Type" drop-down at the bottom of the Save dialog box: one for Text Files (*.txt), and one for All Files (*.*).

```
private void open_Click(object sender, EventArgs e) {
    if (CheckChanged()) {
        openFileDialog1.InitialDirectory = selectedFolder;
        openFileDialog1.Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*";
        openFileDialog1.FileName = description.Text + ".txt";
        DialogResult result = openFileDialog1.ShowDialog();
        if (result == DialogResult.OK) {
            currentExcuse = new Excuse(openFileDialog1.FileName);
            UpdateForm(false);
        }
    }
}
```

Use the DialogResult enum returned by the Open and Save dialog boxes to make sure you only open or save if the user clicked OK, and not Cancel.

```
private void randomExcuse_Click(object sender, EventArgs e) {
    if (CheckChanged()) {
        currentExcuse = new Excuse(random, selectedFolder);
        UpdateForm(false);
    }
}
```

There's at a bug somewhere in this exercise solution! Can you spot it?

```
private bool CheckChanged() {
    if (formChanged) {
        DialogResult result = MessageBox.Show(
              "The current excuse has not been saved. Continue?",
               "Warning", MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
        if (result == DialogResult.No)
            return false;
    }
    return true;
}

private void description_TextChanged(object sender, EventArgs e) {
    currentExcuse.Description = description.Text;
    UpdateForm(true);
}

private void results_TextChanged(object sender, EventArgs e) {
    currentExcuse.Results = results.Text;
    UpdateForm(true);
}

private void lastUsed_ValueChanged(object sender, EventArgs e) {
    currentExcuse.LastUsed = lastUsed.Value;
    UpdateForm(true);
}

class Excuse {
    public string Description { get; set; }
    public string Results { get; set; }
    public DateTime LastUsed { get; set; }
    public string ExcusePath { get; set; }
    public Excuse() {
        ExcusePath = "";
    }
    public Excuse(string excusePath) {
        OpenFile(excusePath);
    }
    public Excuse(Random random, string folder) {
        string[] fileNames = Directory.GetFiles(folder, "*.txt");
        OpenFile(fileNames[random.Next(fileNames.Length)]);
    }
    private void OpenFile(string excusePath) {
        this.ExcusePath = excusePath;
        using (StreamReader reader = new StreamReader(excusePath)) {
            Description = reader.ReadLine();
            Results = reader.ReadLine();
            LastUsed = Convert.ToDateTime(reader.ReadLine());
        }
    }
    public void Save(string fileName) {
        using (StreamWriter writer = new StreamWriter(fileName))
        {
            writer.WriteLine(Description);
            writer.WriteLine(Results);
            writer.WriteLine(LastUsed);
        }
    }
}
```

MessageBox.Show() also returns a DialogResult enum that we can check.

Here are the three Changed event handlers for the three input fields on the form. If any of them are triggered, that means the excuse has changed, so first we update the Excuse instance and then we call UpdateForm(), add the asterisk to the form's title bar, and set Changed to true.

Passing true to UpdateForm() tells it to just mark the form as changed, but not update the input controls.

The Random Excuse button uses Directory.GetFiles() to read all of the text files in the selected folder into an array, and then chooses a random array index to open.

We made sure to use a using statement every time we opened a stream. That way our files will always be closed.

Here's where the using statement comes in. We declared the StreamWriter inside a using statement, so its Close() method is called for us automatically!

Did you call LastUsed.ToString()? Remember, WriteLine() calls it automatically!

Writing files usually involves making a lot of decisions

You'll write lots of programs that take a single input, maybe from a file, and have to decide what to do based on that input. Here's code that uses one long `if` statement — it's pretty typical. It checks the `part` variable and prints different lines to the file based on which enum it uses. There are lots of choices, so lots of `else if`s:

```
enum BodyPart {
    Head,
    Shoulders,
    Knees,
    Toes
}
```

Here's an enum—we'll want to compare a variable against each of the four members and write a different line to the StreamWriter depending on which one it matches. We'll also write something different if none of them match.

```
private void WritePartInfo(BodyPart part, StreamWriter writer) {
  if (part == BodyPart.Head)
    writer.WriteLine("the head is hairy");
  else if (part == BodyPart.Shoulders)
    writer.WriteLine("the shoulders are broad");
  else if (part == BodyPart.Knees)
    writer.WriteLine("the knees are knobby");
  else if (part == BodyPart.Toes)
    writer.WriteLine("the toes are teeny");
  else
    writer.WriteLine("some unknown part is unknown");
}
```

If we use a series of if/else statements, then we end up writing this "if (part ==[option])" over and over.

We've got a final else in case we didn't find a match.

BRAIN POWER

What sort of things can go wrong when you write code that has this many `if/else` statements? Think about typos and bugs caused by brackets, a single equals sign, etc.

Use a switch statement to choose the right option

Comparing one variable against a bunch of different values is a really common pattern that you'll see over and over again. It's especially common when you're reading and writing files. It's so common, in fact, that C# has a special kind of statement designed specifically for this situation.

> **NOTE**
>
> There's nothing about a switch statement that's specifically related to files. It's just a useful C# tool that we can use here.

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 series of `if/else` statements on the opposite page:

A switch statement compares ONE variable against MULTIPLE possible values.

You'll start with the switch keyword followed by the variable that's going to be compared against a bunch of different possible values.

```
private void WritePartInfo(BodyPart part, StreamWriter writer)
{
    switch (part) {
        case BodyPart.Head:
            writer.WriteLine("the head is hairy");
            break;
        case BodyPart.Shoulders:
            writer.WriteLine("the shoulders are broad");
            break;
        case BodyPart.Knees:
            writer.WriteLine("the knees are knobby");
            break;
        case BodyPart.Toes:
            writer.WriteLine("the toes are teeny");
            break;
        default:
            writer.WriteLine("some unknown part is unknown");
            break;
    }
}
```

The body of the switch statement is a series of cases that compare whatever follows the switch keyword against a particular value.

Every case ends with "break;" so C# knows where one case ends and the next begins.

You can also end a case with "return"—the program will compile as long as there's no way for one case to "fall through" to the next one.

Each of these cases consists of the case keyword followed by the value to compare and a colon. After that is a series of statements followed by "break;". Those statements will be executed if the case matches the comparison value.

Switch statements can end with a "default:" block that gets executed if none of the other cases are matched.

Use a switch statement to let your deck of cards read from a file or write itself out to one

The `switch` statement lets you test one value against a bunch of cases and execute different statements depending on which one it matches.

Writing a card out to a file is straightforward — just make a loop that writes the name of each card out to a file. Here's a method you can add to the `Deck` object that does exactly that:

```
public void WriteCards(string filename) {
    using (StreamWriter writer = new StreamWriter(filename)) {
        for (int i = 0; i < cards.Count; i++) {
            writer.WriteLine(cards[i].Name);
        }
    }
}
```

But what about reading the file in? It's not quite so simple. That's where the `switch` statement can come in handy.

The switch statement starts with a **value** to compare against. This switch statement is called from a method that has a suit stored in a string.

```
Suits suit;
switch (suitString) {
    case "Spades":
        suit = Suits.Spades;
        break;
    case "Clubs":
        suit = Suits.Clubs;
        break;
    case "Hearts":
        suit = Suits.Hearts;
        break;
    case "Diamonds":
        suit = Suits.Diamonds;
        break;
    default:
        MessageBox.Show(suitString + " isn't a valid suit!");
}
```

Each of these **case** lines compares some value against the value in the switch line. If they match, it executes all of the following statements until it hits a **break**.

The **default** line comes at the end. If none of the cases match, the statements after the default get executed instead.

Add an overloaded Deck() constructor that reads a deck of cards in from a file

You can use a `switch` statement to build a new constructor for the `Deck` class that you wrote in the last chapter. This constructor reads in a file and checks each line for a card. Any valid card gets added to the deck.

There's a method that you can find on every string that'll come in handy: `Split()`. It lets you split the string into an array of substrings by passing it a `char[]` array of separator characters that it'll use to split up the string.

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"}.

```
public Deck(string filename) {
   cards = new List<Card>();
   using (StreamReader reader = new StreamReader(filename)) {
      while (!reader.EndOfStream) {
         bool invalidCard = false;
         string nextCard = reader.ReadLine();
         string[] cardParts = nextCard.Split(new char[] { ' ' });
         Values value = Values.Ace;
         switch (cardParts[0]) {
             case "Ace": value = Values.Ace; break;
             case "Two": value = Values.Two; break;
             case "Three": value = Values.Three; break;
             case "Four": value = Values.Four; break;
             case "Five": value = Values.Five; break;
             case "Six": value = Values.Six; break;
             case "Seven": value = Values.Seven; break;
             case "Eight": value = Values.Eight; break;
             case "Nine": value = Values.Nine; break;
             case "Ten": value = Values.Ten; break;
             case "Jack": value = Values.Jack; break;
             case "Queen": value = Values.Queen; break;
             case "King": value = Values.King; break;
             default: invalidCard = true; break;
         }
         Suits suit = Suits.Clubs;
         switch (cardParts[2]) {
             case "Spades": suit = Suits.Spades; break;
             case "Clubs": suit = Suits.Clubs; break;
             case "Hearts": suit = Suits.Hearts; break;
             case "Diamonds": suit = Suits.Diamonds; break;
             default: invalidCard = true; break;
         }
         if (!invalidCard) {
             cards.Add(new Card(suit, value));
         }
      }
   }
}
```

This switch statement checks the first word in the line to see if it matches a value. If it does, the right value is assigned to the value variable.

We do the same thing for the third word in the line, except we convert this one to a suit.

There's an easier way to store your objects in files. It's called serialization.

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.

NOTE

OK, just to come clean here: there's also a method called Enum.Parse() — you'll learn about it in Chapter 14 — that will convert the string "Spades" to the enum value Suits.Spades. 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?

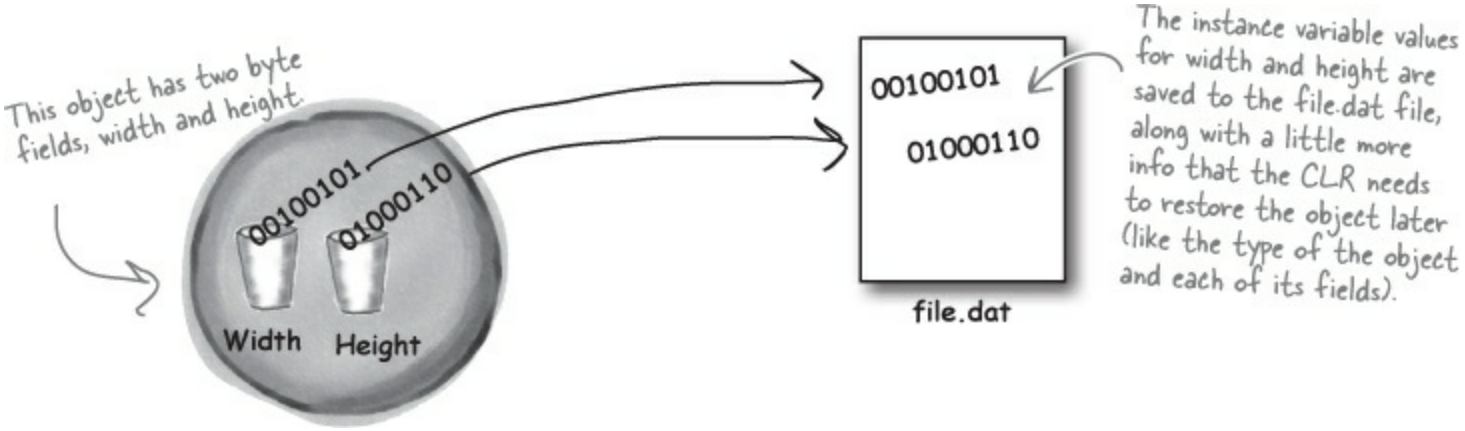
It seems like something mysterious has to happen to an object in order to copy it off of the heap and put it into a file, but it's actually pretty straightforward.

- **❶ 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.

- **❷ 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.



- **❸ And later on...**
  Object on the heap again

Later — maybe days later, and in a different program — you can go back to the file and **deserialize** it. That pulls the original class back out of the file and restores it **exactly as it was**, with all of its fields and values intact.

But what exactly IS an object's state? What needs to be saved?

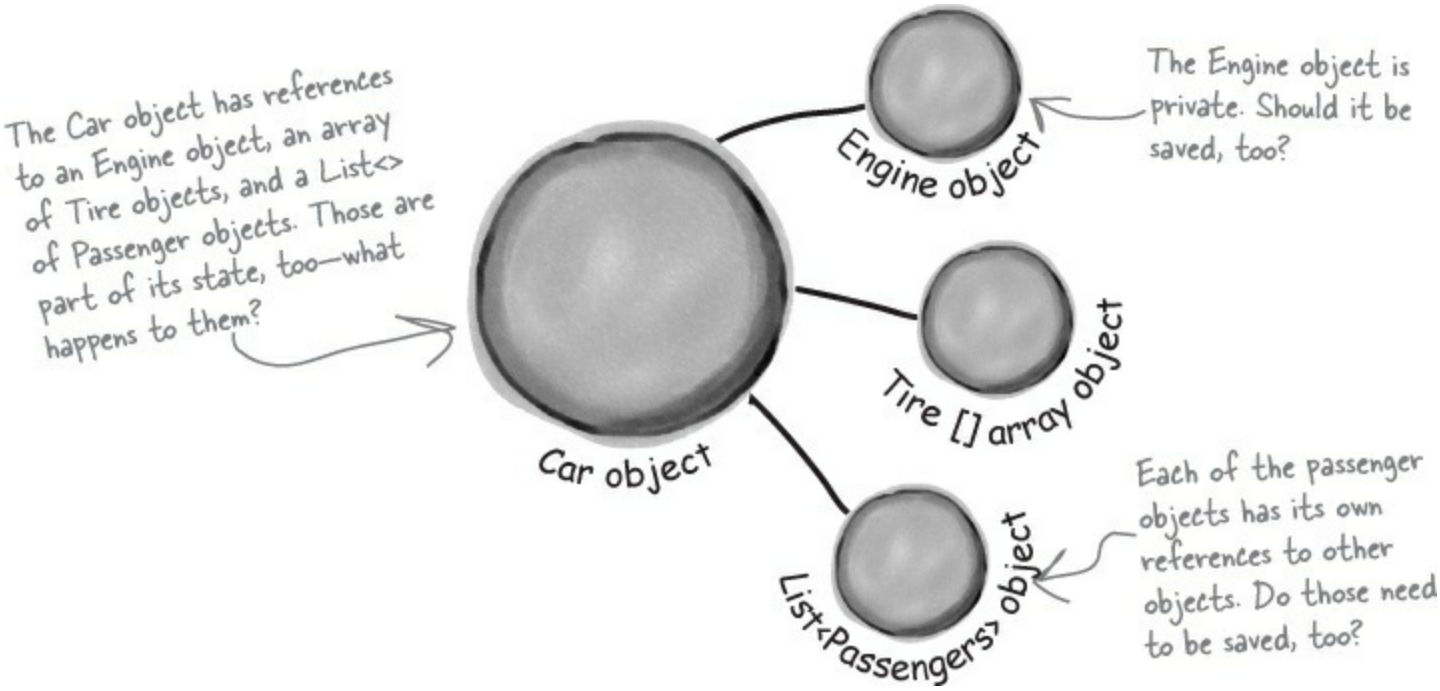
We already know that **an object stores its state in its fields**. So when an object is serialized, every one of those fields needs to be saved to the file.

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.

BRAIN BARBELL

What has to happen for this `Car` object to be saved so that it gets restored back to its original state? Let's say the car has three passengers and a 3-liter engine and all-weather radial tires...aren't those things all part of the `Car` object's state? What should happen to them?

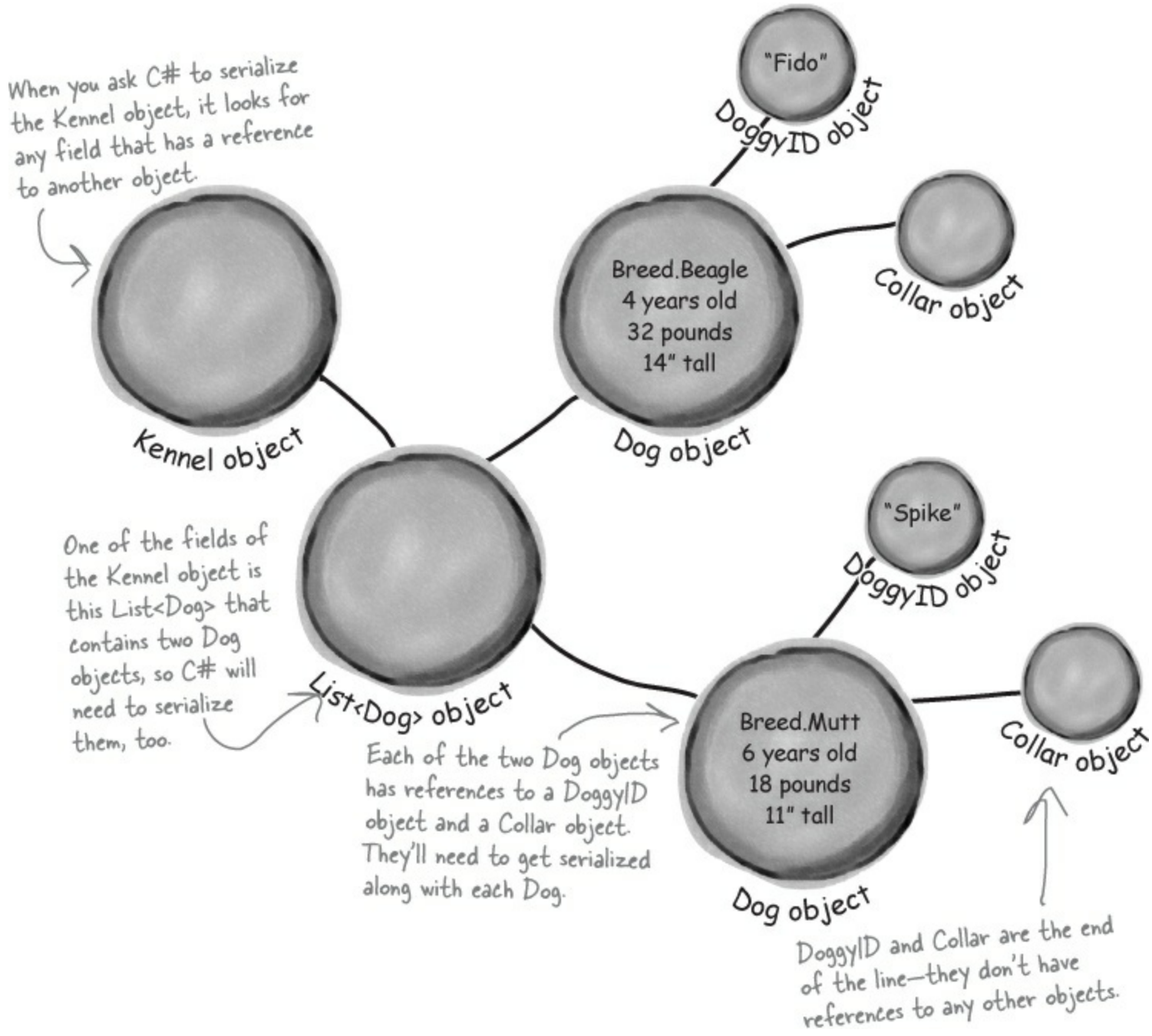


When an object is serialized, all of the objects it refers to get serialized, too...

...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.

NOTE

This whole group of connected objects is sometimes referred to as a graph.



Serialization lets you read or write a whole object graph all at once

You're not just limited to reading and writing lines of text to your files. You can use **serialization** to let your programs copy entire objects to files and read them back in...all in just a few lines of code! There's a tiny amount of prep work you need to do — add one `[Serializable]` line to the top of the class to serialize — but once you do that, everything's ready to write.

It's quick to copy an object out to a file or read it in from one. You can serialize or deserialize it.

You'll need a BinaryFormatter object

If you want to serialize an object graph, the first thing you do is create an instance of `BinaryFormatter`. It's really straightforward to do — and all it takes is one line of code (and an extra `using` line at the top of the class file).

```
using System.Runtime.Serialization.Formatters.Binary;
...
BinaryFormatter formatter = new BinaryFormatter();
```

Now just create a stream and read or write your objects

Use the `Serialize()` method from the `BinaryFormatter` object to write any object out to a stream.

```
using (Stream output = File.Create(filenameString)) {
    formatter.Serialize(output, objectToSerialize);
}
```

The File.Create() method creates a new file. You can open an existing one using File.OpenWrite().

The Serialize() method takes an object and writes it out to a stream. That's a whole lot easier than building a method to write it out yourself!

And once you've got an object serialized out to a file, use the `BinaryFormatter` object's `Deserialize()` method to read it back in. The method returns a reference, so you need to cast the output so that it matches the type of the reference variable you're copying it to.

```
using (Stream input = File.OpenRead(filenameString)) {
    SomeObj obj = (SomeObj)formatter.Deserialize(input);
}
```

When you use Deserialize() to read an object back from a stream, don't forget to cast the return value to match the type of object you're reading.

If you want your class to be serializable, mark it with the [Serializable] attribute

An **attribute** is a special tag that you can add to the top of any C# class. It's how C# stores **metadata** about your code, or information about how the code should be used or treated. When you **add `[Serializable]` to the top of a class just above the class declaration**, you're telling C# that your class is safe for serialization. And you only use it with classes that include fields that are either value types (like an `int`, `decimal`, or `enum`) or other serializable classes. If you don't add the attribute to the class you want to serialize, or if you include a field with a type that isn't serializable, then your program will have an error when you try to run it. ***See for yourself...***

NOTE

Attributes are a way to add information to your class or member declaration. The `[Serializable]` attribute is in the `System` namespace.

DO THIS

- **❶ Create a class and serialize it.**
  Let's serialize Joe so we can keep a file that knows how much money he's got in his pocket even after you close your program. Open the "Fun with Joe and Bob" project from Chapter 3 and update the `Guy` class:

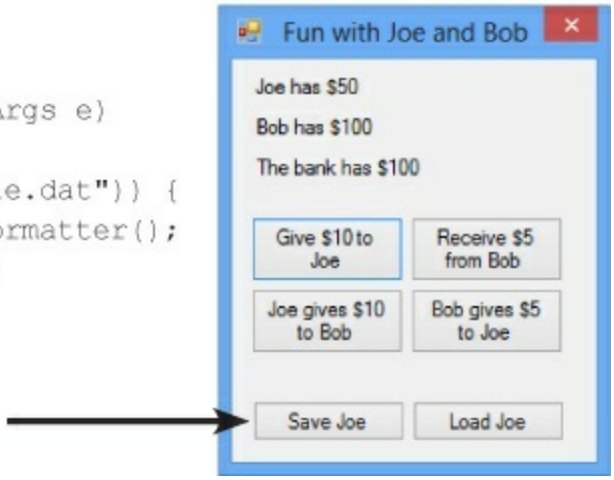
```
[Serializable]
class Guy {
```

You need to add this attribute to the top of any class in order to serialize it.

Next, add a "Save Joe" button and a "Load Joe" button to the form. Here's code for their event handler methods to serialize the `Joe` object to a file called *Guy_file.dat* and read it back:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
...

private void saveJoe_Click(object sender, EventArgs e)
{
    using (Stream output = File.Create("Guy_File.dat")) {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(output, joe);
    }
}
private void loadJoe_Click(object sender, EventArgs e)
{
    using (Stream input = File.OpenRead("Guy_File.dat")) {
        BinaryFormatter formatter = new BinaryFormatter();
        joe = (Guy)formatter.Deserialize(input);
    }
    UpdateForm();
}
```

You'll need these two using lines. The first one is for the file and stream methods, and the second is for serialization.



- **❷ Run the program and play around with it.**
  If Joe had two hundred dollars saved up from his transactions with Bob during your time running the program, it would be a pain to lose all that money just because you needed to exit. Now your program can save Joe out to a file and restore him whenever you want.
  What happens if you delete Guy_File.dat from the bin/Debug folder and then click Load Joe?

Let's serialize and deserialize a deck of cards

Take a deck of cards and write it out to a file. C# makes serializing objects really easy. All you need to do is create a stream and write out your objects.

DO THIS

- **❶ CREATE A NEW PROJECT AND ADD THE DECK AND CARD CLASSES.**
  Right-click on the project in the Solution Explorer and choose Add→Existing Item, and add the `Card` and `Deck` classes (and the `Suits` and `Values` enums and `CardComparer_bySuit` and `CardComparer_byValue` interfaces) you used in Go Fish! in Chapter 8. You'll also need to add the two card comparer classes, since `Deck` uses them. The IDE will copy the files into the new project — make sure you change the `namespace` line at the top of each class file to match your new project's namespace.
- **❷ MARK THE CLASSES SERIALIZABLE.**
  Add the `[Serializable]` attribute to both classes you added to the project.

> **NOTE**
>
> If you don't do this, C# won't let you serialize the classes to a file.

- **❸ ADD A COUPLE OF USEFUL METHODS TO THE FORM.**
  The `RandomDeck` method creates a random deck of cards, and the `DealCards` method deals all of the cards and prints them to the console.

```
Random random = new Random();
private Deck RandomDeck(int number) {
    Deck myDeck = new Deck(new Card[] { });
    for (int i = 0; i < number; i++)
    {
        myDeck.Add(new Card(
            (Suits)random.Next(4),
            (Values)random.Next(1, 14)));
    }
    return myDeck;
}
```

This creates an empty deck and then adds some random cards to it using the Card class from the last chapter.

Don't forget the using statements!

```
using System.IO;

using System.Runtime.
Serialization.Formatters.Binary;
```

```
private void DealCards(Deck deckToDeal, string title) {
    Console.WriteLine(title);
    while (deckToDeal.Count > 0)
    {
        Card nextCard = deckToDeal.Deal(0);
        Console.WriteLine(nextCard.Name);
    }
    Console.WriteLine("------------------");
}
```

The DealCards() method deals each of the cards off of the deck and prints it to the console.

Don't forget to open the IDE's Ouptut window to view the console output from a WinForms program.

- **❹ OK, PREP WORK'S DONE...NOW SERIALIZE THAT DECK.**
  Start by adding buttons to serialize a random deck to a file and read it back. Check the console output to make sure the deck you wrote out is the same as the deck you read.

```
private void button1_Click(object sender, EventArgs e) {
    Deck deckToWrite = RandomDeck(5);
    using (Stream output = File.Create("Deck1.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      bf.Serialize(output, deckToWrite);
    }
    DealCards(deckToWrite, "What I just wrote to the file");
}
private void button2_Click(object sender, EventArgs e) {
    using (Stream input = File.OpenRead("Deck1.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      Deck deckFromFile = (Deck)bf.Deserialize(input);
      DealCards(deckFromFile, "What I read from the file");
    }
}
```

Flip back a page to see the using statements to add to the form.

The BinaryFormatter object takes any object marked with the Serializable attribute—in this case a Deck object—and writes it out to a stream using its Serialize() method.

The BinaryFormatter's Deserialize() method returns an Object, which is just the general type that every C# object inherits from—which is why we need to cast it to a Deck object.

- **❺ NOW SERIALIZE A BUNCH OF DECKS TO THE SAME FILE.**
  Once you open a stream, you can write as much as you want to it. You can serialize as many objects as you need into the same file. So now add two more buttons to write out a random number of decks to the file. Check the output to make sure everything looks good.

```
private void button3_Click(object sender, EventArgs e) {
    using (Stream output = File.Create("Deck2.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      for (int i = 1; i <= 5; i++) {
        Deck deckToWrite = RandomDeck(random.Next(1,10));
        bf.Serialize(output, deckToWrite);
        DealCards(deckToWrite, "Deck #" + i + " written");
      }
    }
}
private void button4_Click(object sender, EventArgs e) {
    using (Stream input = File.OpenRead("Deck2.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      for (int i = 1; i <= 5; i++) {
        Deck deckToRead = (Deck)bf.Deserialize(input);
        DealCards(deckToRead, "Deck #" + i + " read");
      }
    }
}
```

You can serialize one object after another to the same stream.

Notice how the line that reads a single deck from the file uses (Deck) to cast the output of Deserialize() to a Deck. That's because Deserialize() returns an object, but doesn't necessarily know what type of object.

As long as you cast the objects you read off the stream to the right type, you can serialize or deserialize a while bunch of objects, one after another.

- **❻ TAKE A LOOK AT THE FILE YOU WROTE.**
  Open up *Deck1.dat* in Notepad (`File.Create()` created it in the *bin\Debug* folder under your project folder). It may not be something you'd read on the beach, but it's got all the information to restore your whole deck of cards.

When you serialize objects out to a file, they're written in a binary format.

But that doesn't mean it's indecipherable — just compact. That's why you can recognize the strings when you open up a file with serialized objects in it: that's the most compact way C# can write strings to a file — as strings. But writing out a number as a string would be really wasteful. Any `int` can be stored in four bytes. Storing the number 49,369,144 as an 8-character string that you could read takes 8 characters (10 if you include commas), but a binary formatted int only takes 4 bytes.

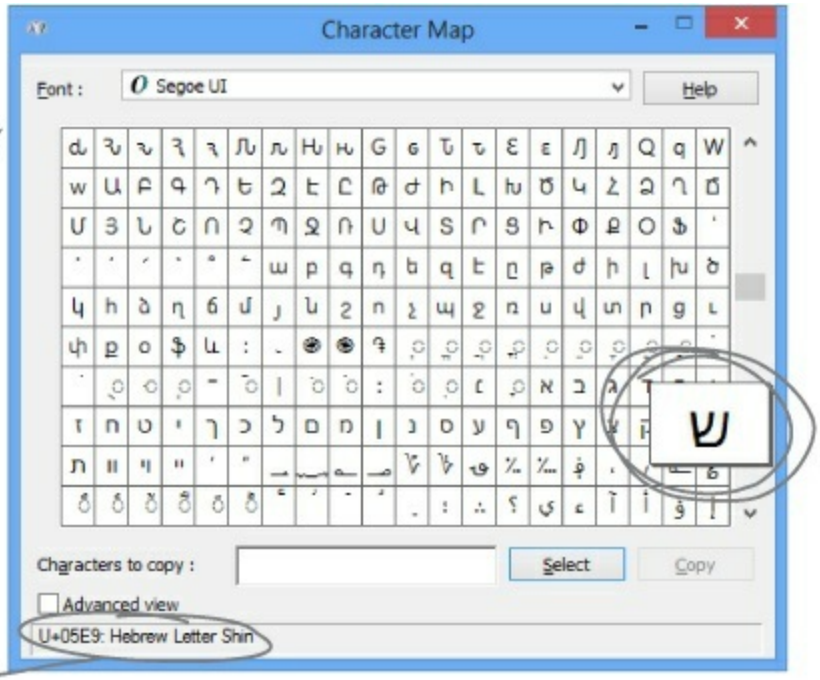
Later in the book you'll learn about a less compact, more human-readable (and editable!) serialization format.

BEHIND THE SCENES

.NET uses **Unicode** to encode a char or string into bytes. Luckily, Windows has a useful little tool to help us figure out how Unicode works. Open up the Character Map (use the Search charm on the Start page to find it, or press Windows-R and type "charmap.exe").

When you look at all the letters and symbols that are used in languages all around the world, you realize just how many different *things* need to be written to a file just to store text. That's why .NET **encodes** all of its strings and characters in a format called Unicode. Encoding just means taking the logical data (like the letter H) and turning it into bytes (the number 72). It needs to do that because letters, numbers, enums, and other data all end up in bytes on disk or in memory. And that's why Character Map is useful — it shows you how letters are encoded into numbers.

Select the Segoe UI font and scroll down until you reach the Hebrew letters. Find the letter Shin and click on it.



As soon as you click on the letter, its Unicode number shows up in the status bar. The Hebrew letter Shin is number 05E9. That's a hexadecimal number—"**hex**" for short.

You can convert it to decimal using the Windows desktop calculator: open it up, put it in Scientific mode, click the "Hex" radio button, enter "05E9", and then click "Dec"—it's 1,513.

Unicode is an industry standard developed by a nonprofit group called the Unicode Consortium, and it works across programs and different computer platforms. Take a minute and look at their website: **http://unicode.org/**

.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. So start a new project and drag three buttons onto a form, and we'll use the `File.WriteAllBytes()` and `ReadAllBytes()` methods to get a sense of exactly how Unicode data is written out to a file.

DO THIS!

- **❶ WRITE A NORMAL STRING OUT TO A FILE AND READ IT BACK.**
  Use the same `WriteAllText()` method that you used in the text editor to have the first button write the string "Eureka!" out to a file called *eureka.txt*. Then create a new byte array called `eurekaBytes`, read the file into it, and then print out all of the bytes read:

```
File.WriteAllText("eureka.txt", "Eureka!");
byte[] eurekaBytes = File.ReadAllBytes("eureka.txt");
foreach (byte b in eurekaBytes)
    Console.Write("{0} ", b);
Console.WriteLine();
```

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`. Now **open up the file in the Simple Text Editor** that you wrote earlier in the chapter. It says "Eureka!"

- **❷ MAKE THE SECOND BUTTON DISPLAY THE BYTES AS HEX NUMBERS.**
  It's not just Character Map that shows numbers in hex. Almost anything you read that has to do with encoding data will show that data in hex, so it's useful to know how to work with it. Make the code for the second button's event handler in your program **identical to the first one**, except change the `Console.Write()` line so it looks like this instead:

```
Console.Write("{0:x2} ", b);
```

> **NOTE**
>
> 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`

- **❸ MAKE THE THIRD BUTTON WRITE OUT HEBREW LETTERS.**
  Go back to Character Map and double-click on the Shin character (or click the Select button). It'll add it to the "Characters to copy" box. Then do the same for the rest of the letters in "Shalom": Lamed (U+05DC), Vav (U+05D5), and Final Mem (U+05DD). Now add the code for the third button's event handler. It'll look exactly like button 2, except for one change. Click the Copy button in Character Map, and then paste the letters over "Eureka!" and add the `Encoding.Unicode` parameter, so it looks like this:

```
File.WriteAllText("eureka.txt", "שלום", Encoding.Unicode);
```

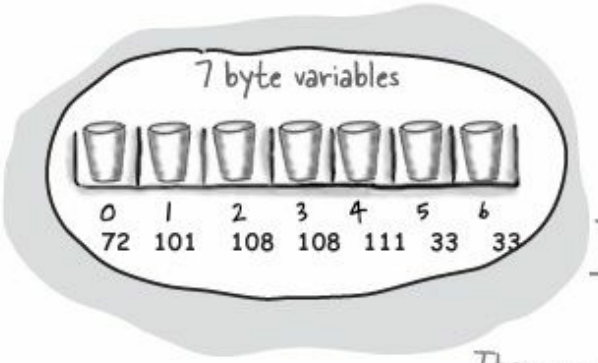
  Did you notice that the IDE pasted the letters in **backward**? That's because it knows that Hebrew is read right-to-left, so any time it encounters Hebrew Unicode letters, it displays them right-to-left. Put your cursor in the middle of the letters — the left and right arrow keys reversed! That makes it a lot easier if you need to type in Hebrew. Now run the code, 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 your simple text editor — it looks right!

C# can use byte arrays to move data around

Since all your data ends up encoded as **bytes**, it makes sense to think of a file as one **big byte array**. And you already know how to read and write byte arrays.

Here's the code to create a byte array, open an input stream, and read the text 'Hello!!' into bytes 0 through 6 of the array.

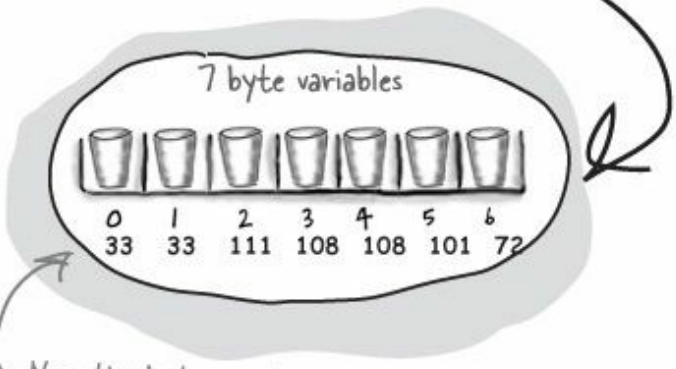
```
byte[] greeting;
greeting = File.ReadAllBytes(filename);
```



These numbers are the Unicode numbers for the characters in "Hello!!"

This is a static method for Arrays that reverses the order of the bytes. We're just using it to show that the changes you make to the byte array get written out to the file exactly.

```
Array.Reverse(greeting);
File.WriteAllBytes(filename, greeting);
```



Now the bytes are in reverse order.

When the program writes the byte array out to a file, the text is in reverse order too.

Reversing the bytes in "Hello!!" only works because each of those characters is one byte long. Can you figure out why this won't work for שלום?

Use a BinaryWriter to write binary data

You ***could*** encode all of your strings, chars, ints, and floats into byte arrays before writing them out to files, but that would get pretty tedious. That's why .NET gives you a very useful class called **`BinaryWriter`** that **automatically encodes your data** and writes it to a file. All you need to do is create a `FileStream` and pass it into the `BinaryWriter`'s constructor (they're in the System.IO namespace, so you'll need `using System.IO;`). Then you can call its methods to write out your data. So let's create a new Console Application that uses `BinaryWriter` to write binary data to a file.

NOTE

StreamWriter also encodes your data. It just specializes in text and text encoding.

DO THIS!

- ❶ 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';
```

- ❷ To use a `BinaryWriter`, first you need to open a new stream with `File.Create()`:

  NOTE

  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.

```
using (FileStream output = File.Create("binarydata.dat"))
using (BinaryWriter writer = new BinaryWriter(output)) {
```

- ❸ 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);
}
```

Each Write() statement encodes one value into bytes, and then sends those bytes to the FileStream object. You can pass it any value type, and it'll encode it automatically.

The FileStream writes the bytes to the end of the file.

SHARPEN YOUR PENCIL

- ❹ 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);

Console.ReadKey();
```

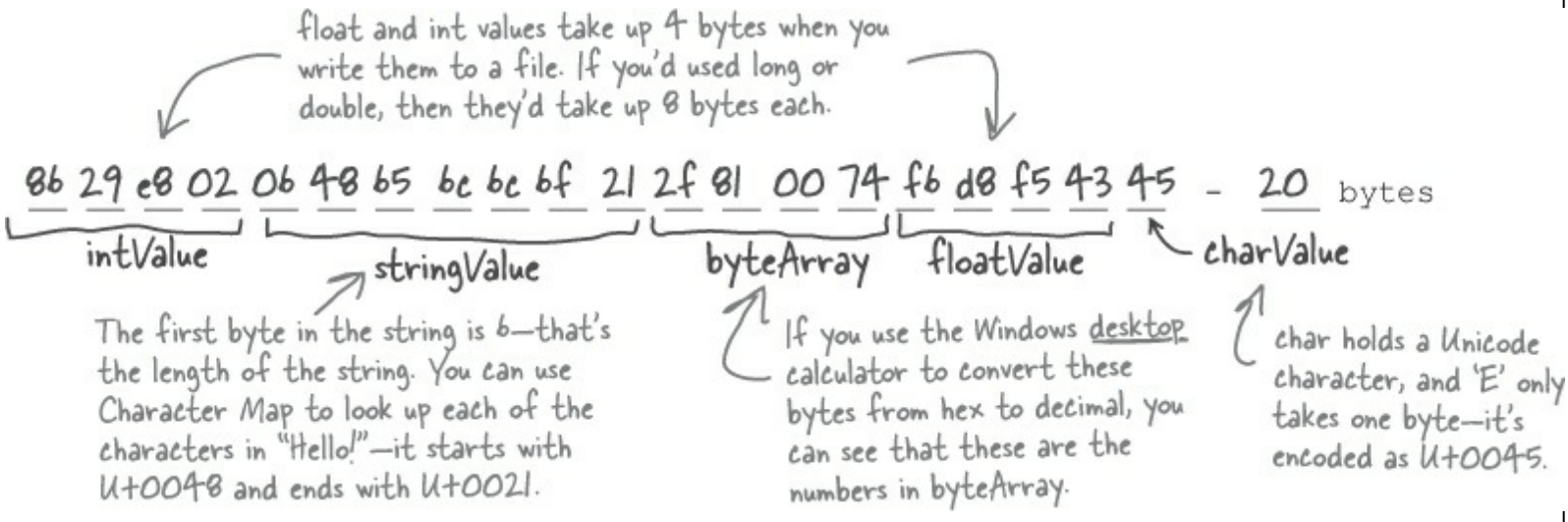
Write down the output in the blanks below. Can you **figure out what bytes correspond** to each of the five `Write()` statements? Mark each group of bytes with the name of the variable.

__ - ___ bytes

NOTE

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. Also, you can look up the string and char Unicode values using Character Map.

SHARPEN YOUR PENCIL SOLUTION



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.

NOTE

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.

- ❶ Start out by setting up the `FileStream` and `BinaryReader` objects:

```
using (FileStream input = File.OpenRead("binarydata.dat"))
using (BinaryReader reader = new BinaryReader(input)) {
```

- ❷ 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.

- ❸ You tell `BinaryReader` what type of data to read by calling its different methods.

If you're adding this code to the end of the program on the previous page, don't forget the other ReadKey() that waits for a keystroke.

```
        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);
    }
    Console.ReadKey();
```

Here's the output that gets printed to the console:

```
int: 48769414 string: Hello! bytes: 47 129 0 116 float: 491.695 char: E
```

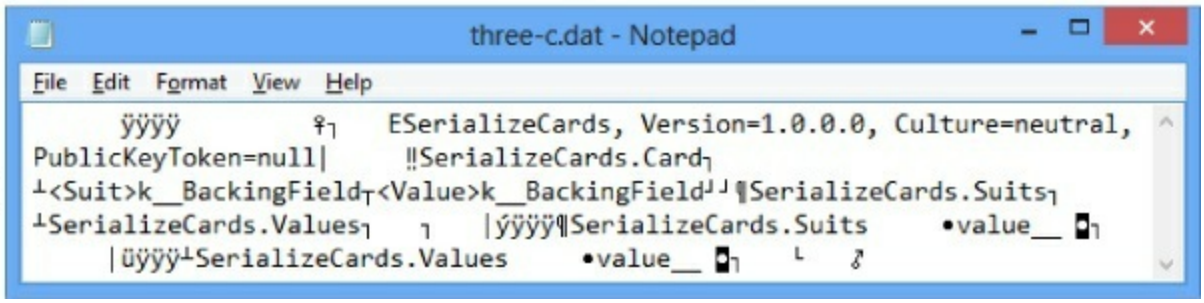
You can read and write serialized files manually, too

Serialized files don't look so pretty when you open them up in Notepad. You'll find all the files you write in your project's *bin\Debug* folder — let's take a minute and get more acquainted with the inner workings of a serialized file.

DO THIS!

- **❶ Serialize two `Card` objects to different files.**
  Use the serialization code you've already written to serialize the **Three of Clubs** to *three-c.dat* and **Six of Hearts** to *six-h.dat*. Check to make sure that both files were written out and are now in a folder, and that they both have the same file size. Then open one of them in Notepad:

There are some words in the file, but it's mostly unreadable.



- **❷ Write a loop to compare the two binary files.**
  We used the `ReadByte()` method to read the next byte from a stream — it returns an `int` that contains the value of that byte. We also used the stream's `Length` field to make sure we read the whole file.

```
byte[] firstFile = File.ReadAllBytes("three-c.dat");
byte[] secondFile = File.ReadAllBytes("six-h.dat");
for (int i = 0; i < firstFile.Length; i++)
    if (firstFile[i] != secondFile[i])
        Console.WriteLine("Byte #{0}: {1} versus {2}",
            i, firstFile[i], secondFile[i]);
```

This loop examines the first byte from each of the files and compares them, then the second byte, then the third, etc. When it finds a difference, it writes a line to the console.

The two files are read into two different byte arrays, so they can be compared byte by byte. Since the same class was serialized to two different files, they'll be almost identical...but let's see just HOW identical they are.

WATCH IT!

When you write to a file, you don't always start from a clean slate!

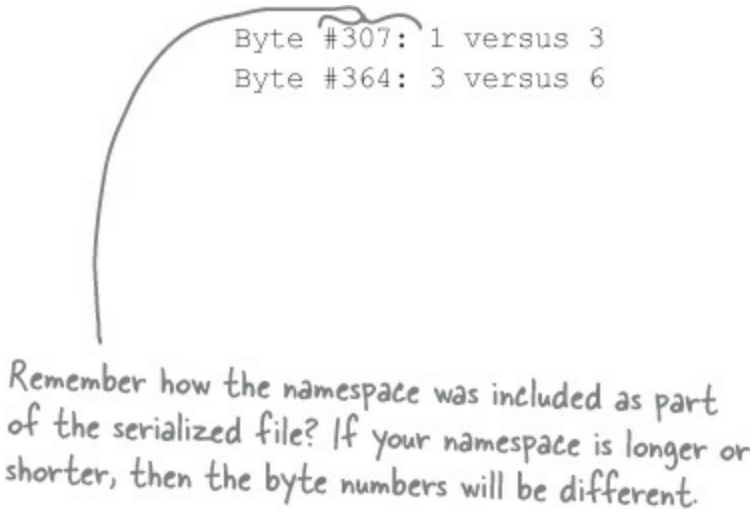
Be careful if you use File.OpenWrite(). It doesn't delete the file — it just starts overwriting the data starting at the beginning. That's why we've been using File.Create() — it creates a new file.

We're not done yet — flip the page!

Find where the files differ, and use that information to alter them

The loop you just wrote pinpoints exactly where the two serialized `Card` files differ. Since the only difference between the two objects were their `Suit` and `Value` fields, then that should be the only difference in their files, too. So if we find the bytes that hold the suit and value, we should be able to **change them to make a new card** with whatever suit and value we want!

- **❸ Take a look at the console output to see how the two files differ.**
  The console should show that two bytes differ:



  That should make a lot of sense! Go back to the `Suits` enum from the last chapter, and you'll find the value for Clubs is 1 and the value for Hearts is 3, so that's the first difference. And the second difference — six versus three — is pretty obviously the card's value. You might see different byte numbers, which isn't surprising: you might be using a different namespace, which would change the length of the file.

> **NOTE**
>
> Hmm, if byte #307 in the serialized file represents the suit, then we should be able to change the suit of the card by reading that file in, changing that one byte, and writing it out again. (Remember, your own serialized file might store the suit at a different location.)

- **❹ Write code to manually create a new file that contains the King of Spades.**
  We'll take one of the arrays that we read, alter it to contain a new card, and write it back out.

```
firstFile[307] = (byte)Suits.Spades;
firstFile[364] = (byte)Values.King;
File.Delete("king-s.dat");
File.WriteAllBytes("king-s.dat", firstFile);
```

If you found different byte numbers in step #3, substitute them in here.

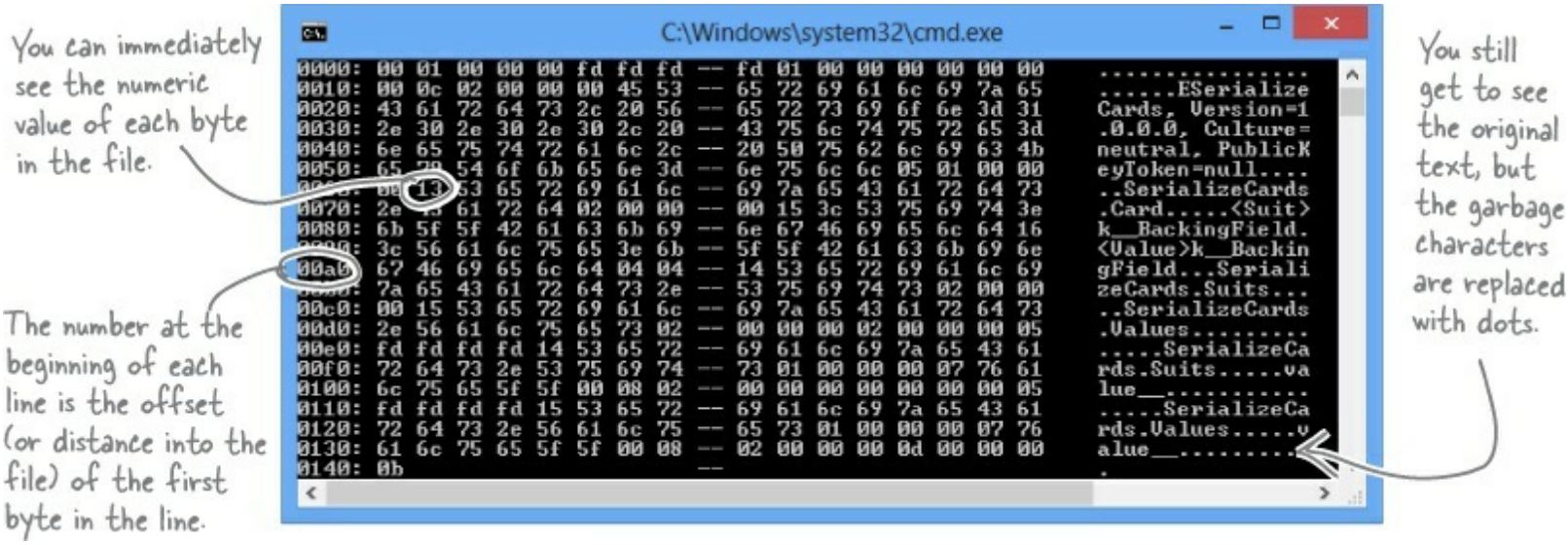
Now that you know which bytes contain the suit and value, you can change just those bytes in the array before it gets written out to king-s.dat.

Now **deserialize the card from *king-s.dat*** and see if it's the King of Spades!

Working with binary files can be tricky

What do you do if you have a file and you aren't quite sure what's inside it? You don't know what application created it, and you need to know something about it — but when you open it in Notepad, it looks like a bunch of garbage. What if you've exhausted all your other options, and really need to just look inside? Looking at that picture, it's pretty clear that Notepad just isn't the right tool.

There's another option — it's a format called a *hex dump*, and it's a pretty standard way to look at binary data. It's definitely more informative than looking at the file in Notepad. Hexadecimal — or "hex" — is a convenient way to display bytes in a file. Every byte takes 2 characters to display in hex, so you can see a lot of data in a really small space, and in a format that makes it easy to spot patterns. Also, 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, and is 4 bytes long when serialized on disk. Here's what that same file looks like as a hex dump, using one of any number of free hex dump programs available for Windows:



Use file streams to build a hex dumper

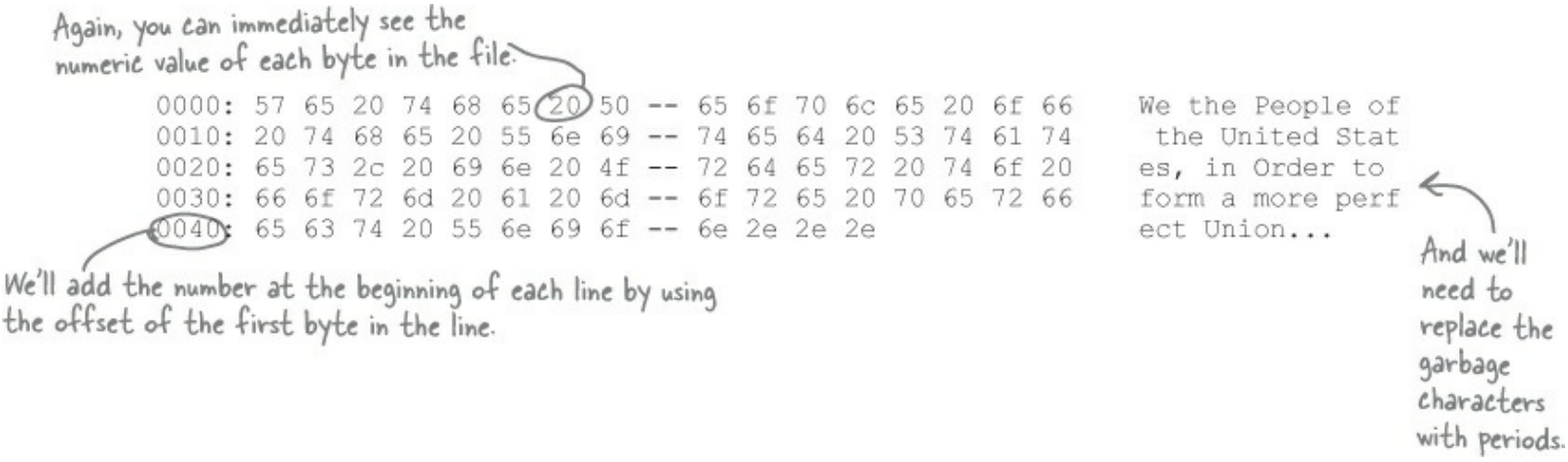
A **hex dump** is a *hexadecimal* view of the contents of a file, and it's a really common way for programmers to take a deep look at a file's internal structure. Most operating systems ship with a built-in hex dump utility. Unfortunately, Windows doesn't. So let's build one!

How to make a hex dump

Start with some familiar text:

```
We the People of the United States, in Order to form a more perfect Union...
```

Here's what a hex dump of that text would look like:



Each of those numbers — 57, 65, 6F — is the value of one byte in the file. The reason some of the "numbers" have letter values is that they're *hexadecimal* (or hex). That's just another way of writing a number. Instead of using 10 digits from 0 to 9, it uses 16 digits from 0 to 9 plus the letters A through F.

Each line in our hex dump represents 16 characters in the input that was used to generate it. In our dump, 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. (Other hex dumps look slightly different, but this one will do for us.)

Working with hex

You can put hex numbers directly into your program — just add the characters `0x` (a zero followed by an x) in front of the number:

```
int j = 0x20;
MessageBox.Show("The value is " + j);
```

When you use the + operator to concatenate a number into a string, it gets converted to decimal. You can use the static `String.Format()` method to convert your number to a hex-formatted string instead:

```
string h = String.Format("{0:x2}", j);
```

NOTE

String.Format() uses parameters just like Console.WriteLine(), so you don't need to learn anything new to use it.

StreamReader and StreamWriter will do just fine (for now)

Our hex dumper will write its dump out to a file, and since it's just writing text a `StreamWriter` will do just fine. But we can also take advantage of the **`ReadBlock()`** method in `StreamReader`. It reads a block of characers 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.

> **NOTE**
>
> The reason the method's called ReadBlock() is that when you call it, it "blocks" (which means it keeps executing and doesn't return to your program) until it's either read all the characters you asked for or run out of data to read.

So add one more button to your program — add this hex dumper to it. Change the first two lines so that they point to real files on your hard drive. Start with a serialized `Card` file — copy it to a folder, then put its path in the StreamReader's constructor.

```
using (StreamReader reader = new StreamReader(@"c:\files\inputFile.txt"))
using (StreamWriter writer = new StreamWriter(@"c:\files\outputFile.txt", false))
{
  int position = 0;

  while (!reader.EndOfStream) {
    char[] buffer = new char[16];
    int charactersRead = reader.ReadBlock(buffer, 0, 16);
    writer.Write("{0}: ", String.Format("{0:x4}", position));
    position += charactersRead;

    for (int i = 0; i < 16; i++) {
      if (i < charactersRead) {
        string hex = String.Format("{0:x2}", (byte)buffer[i]);
        writer.Write(hex + " ");
      }
      else
        writer.Write("   ");

      if (i == 7) { writer.Write("-- "); }
      if (buffer[i] < 32 || buffer[i] > 250) { buffer[i] = '.'; }
    }
    string bufferContents = new string(buffer);
    writer.WriteLine("   " + bufferContents.Substring(0, charactersRead));
  }
}
```

A StreamReader's EndOfStream property returns false if there are characters still left to read in the file.

This ReadBlock() call reads up to 16 characters into a char array.

The static String.Format method converts numbers to strings. "{0:x4}" tells Format() to print the second parameter—in this case, position—as a four-character hex number.

This loop goes through the characters and prints each of them to a line in the output.

Some characters with a value under 32 don't print, so we'll replace all of them with a period.

You can convert a char[] array to a string by passing it to the overloaded constructor for string.

Every string has a Substring method that returns a piece of the string. In this case, it returns the first charactersRead characters starting at the beginning (position 0). (Look back at the top of the loop to see where charactersRead is set—the ReadBlock() method returns the number of characters that it read into the array.)

Use Stream.Read() to read bytes from a stream

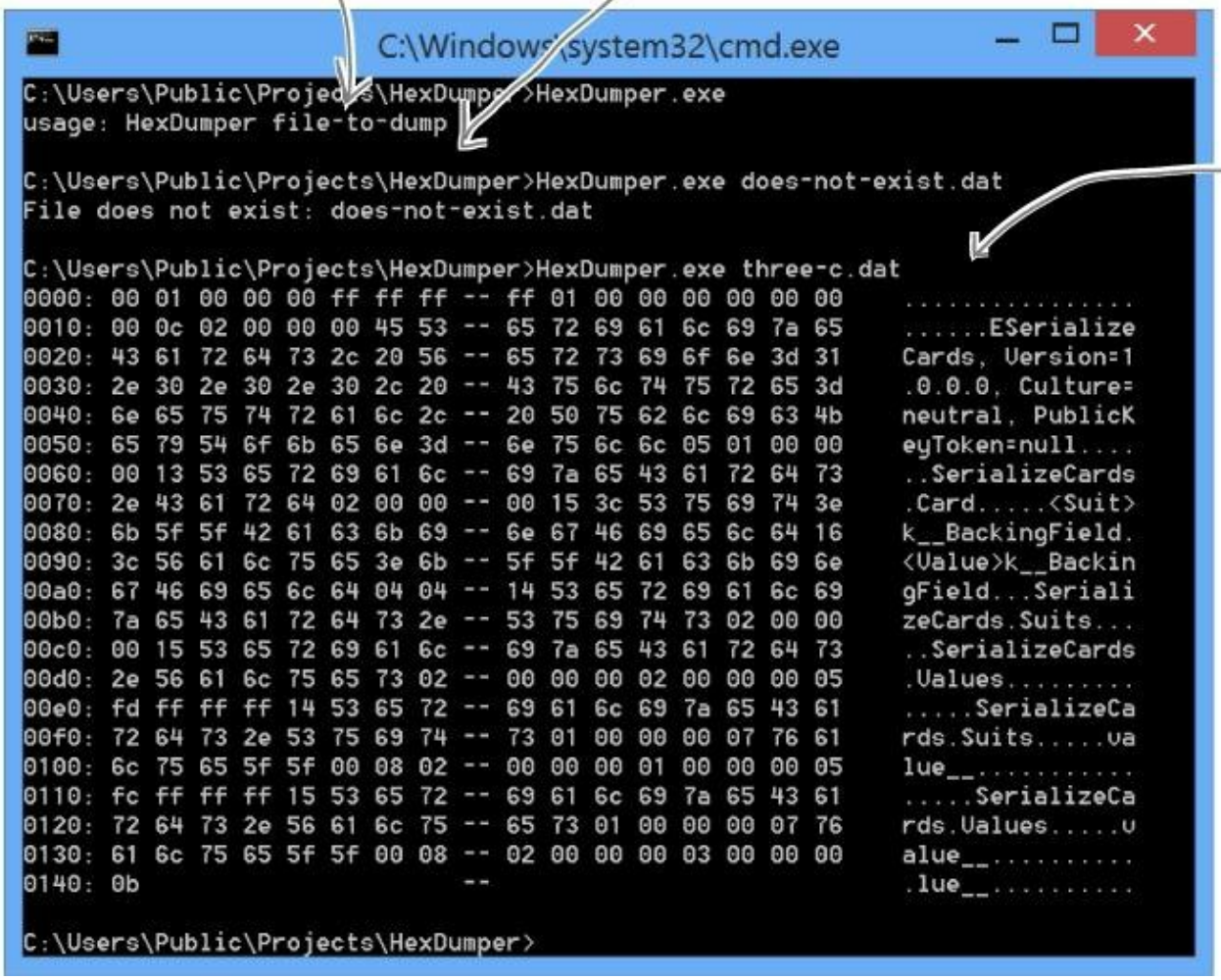
Do this

The hex dumper works just fine for text files. But there's a problem. Try using `File.WriteAllBytes()` to write an array of bytes with values over 127 to a file and then run it through your dumper. Uh oh — they're all read in as "fd"! That's because **`StreamReader` is built to read text files**, which only contain bytes with values under 128. So let's do this right — by reading the bytes directly from the stream using the **`Stream.Read()`** method. And as a bonus, we'll build it just like a real hex dump utility: we'll make it take a filename as a **command-line argument**.

Create a new Console Application and **call it `HexDumper`**. The code for the program is on the facing page. Here's what it will look like when you run the program:

If you run HexDumper without any arguments, it returns an error message and exits with an error code.

It also exits with an error if you pass it the name of a file that doesn't exist.



If you pass it a valid filename, it'll write a hex dump of the contents of the file to the console.

Normally we use Console.WriteLine() to print to the console. But we'll use Console.Error.WriteLine() to print error messages so they don't get redirected if we use > or >> to redirect the output.

USING COMMAND-LINE ARGUMENTS

Every time you create a new Console Application project, Visual Studio creates a Program class with an entry point method that has this declaration: **static void Main(string[] args)**. If you run your program with command-line arguments, the **args** parameter will contain those arguments. And it's not just for Console Applications, either: open up any Windows Forms Application project's Program.cs file, and you'll see the same thing.

You'll want to pass command-line arguments when you're debugging your program. To pass arguments when you run your program in the IDE's debugger, choose "Properties..." from the Project menu and enter them on the Debug tab.

Command-line arguments will be passed using the args parameter.

If args.Length is not equal to 1, then either zero or more than one argument was passed on the command line.

```
static void Main(string[] args)
{
    if (args.Length != 1)
    {
        Console.Error.WriteLine("usage: HexDumper file-to-dump");
        System.Environment.Exit(1);
    }
    if (!File.Exists(args[0]))
    {
        Console.Error.WriteLine("File does not exist: {0}", args[0]);
        System.Environment.Exit(2);
    }
    using (Stream input = File.OpenRead(args[0]))
    {
        int position = 0;
        byte[] buffer = new byte[16];
        while (position < input.Length)
        {
            int charactersRead = input.Read(buffer, 0, buffer.Length);
            if (charactersRead > 0)
            {
                Console.Write("{0}: ", String.Format("{0:x4}", position));
                position += charactersRead;

                for (int i = 0; i < 16; i++)
                {
                    if (i < charactersRead)
                    {
                        string hex = String.Format("{0:x2}", (byte)buffer[i]);
                        Console.Write(hex + " ");
                    }
                    else
                        Console.Write("   ");

                    if (i == 7)
                        Console.Write("-- ");

                    if (buffer[i] < 32 || buffer[i] > 250) { buffer[i] = (byte)'.'; }
                }
                string bufferContents = Encoding.UTF8.GetString(buffer);
                Console.WriteLine("  " + bufferContents);
            }
        }
    }
}
```

This Exit() method quits the program. If you pass it an int, it will return that error code (which is useful when you're writing command scripts and batch files).

Notice how we're using Console.Error.WriteLine() here.

Let's make sure that a valid file was passed. If it doesn't exist, print a different error message and return a different exit code.

We don't need a StreamReader because we're reading bytes directly from the stream.

Use the Stream.Read() method to read bytes directly into a buffer. Notice how this time the buffer is a byte array. That makes sense—we're reading bytes, not characters from a text file.

This part of the program is exactly the same, except the buffer contains bytes and not characters (but String.Format() does the right thing in either case).

This is an easy way to convert a byte array to a string. It's part of Encoding.UTF8 (or another Unicode encoding, or ASCII, or another encoding) because different encodings can map the same byte array to different strings.

NOTE

If you use Start Without Debugging (Ctrl-F5) from the Debug menu to run a Console App, you'll get a convenient pause and a "Press any key to continue..." prompt after the program exits.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Why didn't I have to use the `Close()` method to close the file after I used `File.ReadAllText()` and `File.WriteAllText()`?**

A: **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: **Q: If the `FileStream` has methods for reading and writing, why do I ever need to use `StreamReader` and `StreamWriter`?**

A: **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 — like the first version of the Excuse Generator, which only wrote strings out to files. 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 easy to see how they make your life easier.

Q: **Q: When should I use `File`, and when should I use `FileInfo`?**

A: **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`.

Q: **Q: Back up a minute. Why was "Eureka!" written out with one byte per character, but when I wrote out the Hebrew letters they took up two bytes? And what was that "FF FE" thing at the beginning of the bytes?**

A: **A:** What you're seeing is the difference between two **closely related** Unicode encodings. 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. (If you've used ASCII before, they're the same as the ASCII characters.) If a file only contains those Unicode characters with low numbers, it just prints out their bytes.

Things get a little more complicated when you add higher-numbered Unicode characters 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 leading zeroes.)

Q: **Q: Why is it called a byte order mark?**

A: **A:** Remember how your bytes were reversed? Shin's Unicode value of U+05E9 was written to the file as E9 05. That's called "little endian." Go back to the code that wrote out those bytes and change the third parameter to `WriteAllText(): Encoding.BigEndianUnicode`. That tells it to write the data out in "big endian," which doesn't flip the bytes around. You'll see the bytes come out as "05 E9" this time. You'll also see a different byte order mark: FE FF. And your simple text editor is smart enough to read both of them!

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.

NOTE

The encoding is called UTF-8, which .NET uses by default. You can tell File.WriteAllText() to use a different encoding by passing it a different encoding value. You can learn more about Unicode encodings at http://unicode.org.

EXERCISE

Change Brian's Excuse Manager so it uses binary files with serialized `Excuse` objects instead of text files.

- **❶ Make the `Excuse` class serializable.**
  Mark the `Excuse` class with the `[Serializable]` attribute to make it serializable. Also, you'll need to add the `using` line: `using System.Runtime.Serialization.Formatters.Binary;`
- **❷ Change the `Excuse.Save()` method to serialize the excuse.**
  When the `Save()` method writes a file out to the folder, instead of using `StreamWriter` to write the file out, have it open a file and serialize itself out. You'll need to figure out how the current class can deserialize itself.

> **NOTE**
>
> Hint: What keyword can you use inside of a class that returns a reference to itself?

- **❸ Change the `Excuse.OpenFile()` method to deserialize an excuse.**
  You'll need to create a temporary `Excuse` object to deserialize from the file, and then copy its fields into the current class.
- **❹ Now just change the form so it uses a new file extension.**
  There's just one very small change you need to make to the form. Since we're no longer working with text files, we shouldn't use the *.txt* extension anymore. Change the dialog boxes, default filenames, and directory search code so that they work with `*.excuse` files instead.

That's right! Your code was very easy to change because the class was well encapsulated.

When you've got a class that hides its internal operations from the rest of the program and only exposes the behavior that needs to be exposed, it's called a **well-encapsulated class**. In the Excuse Manager program, the form doesn't have any information about how excuses are saved to files. It just passes a filename into the excuse class, and the class takes care of the rest. That makes it very easy to make big changes to how your class works with files. The better you encapsulate your classes, the easier they are to alter later on.

NOTE

Remember how encapsulation was one of the four core OOP principles? Here's an example of how using those principles makes your programs better.

EXERCISE SOLUTION

Change Brian's Excuse Manager so it uses binary files with serialized `Excuse` objects instead of text files.

> **NOTE**
>
> **You only need to change these three statements in the form: two in the Save button's `Click` event, and one in the Open button's — they just change the dialogs to use the *.excuse* extension, and set the default save filename.**

```
private void save_Click(object sender, EventArgs e) {
  // existing code
  saveFileDialog1.Filter = "Excuse files (*.excuse)|*.excuse|All files (*.*)|*.*";
  saveFileDialog1.FileName = description.Text + ".excuse";
  // existing code
}
private void open_Click(object sender, EventArgs e) {
  // existing code
  openFileDialog1.Filter =
    "Excuse files (*.excuse)|*.excuse|All files (*.*)|*.*";
  openFileDialog1.FileName = description.Text + ".excuse";
  // existing code
}
[Serializable]
class Excuse {
    public string Description { get; set; }
    public string Results { get; set; }
    public DateTime LastUsed { get; set; }
    public string ExcusePath { get; set; }

    public Excuse() {
        ExcusePath = "";
    }
    public Excuse(string excusePath) {
        OpenFile(excusePath);
    }
    public Excuse(Random random, string folder) {
        string[] fileNames = Directory.GetFiles(folder, "*.excuse");
        OpenFile(fileNames[random.Next(fileNames.Length)]);
    }
    private void OpenFile(string excusePath) {
        this.ExcusePath = excusePath;
        BinaryFormatter formatter = new BinaryFormatter();
        Excuse tempExcuse;
        using (Stream input = File.OpenRead(excusePath)) {
            tempExcuse = (Excuse)formatter.Deserialize(input);
        }
        Description = tempExcuse.Description;
        Results = tempExcuse.Results;
        LastUsed = tempExcuse.LastUsed;
    }
    public void Save(string fileName) {
        BinaryFormatter formatter = new BinaryFormatter();
        using (Stream output = File.OpenWrite(fileName)) {
            formatter.Serialize(output, this);
        }
    }
}
```

Standard save and open dialog boxes do the trick here.

The only change to the form is to have it change the file extension it passes to the Excuse class.

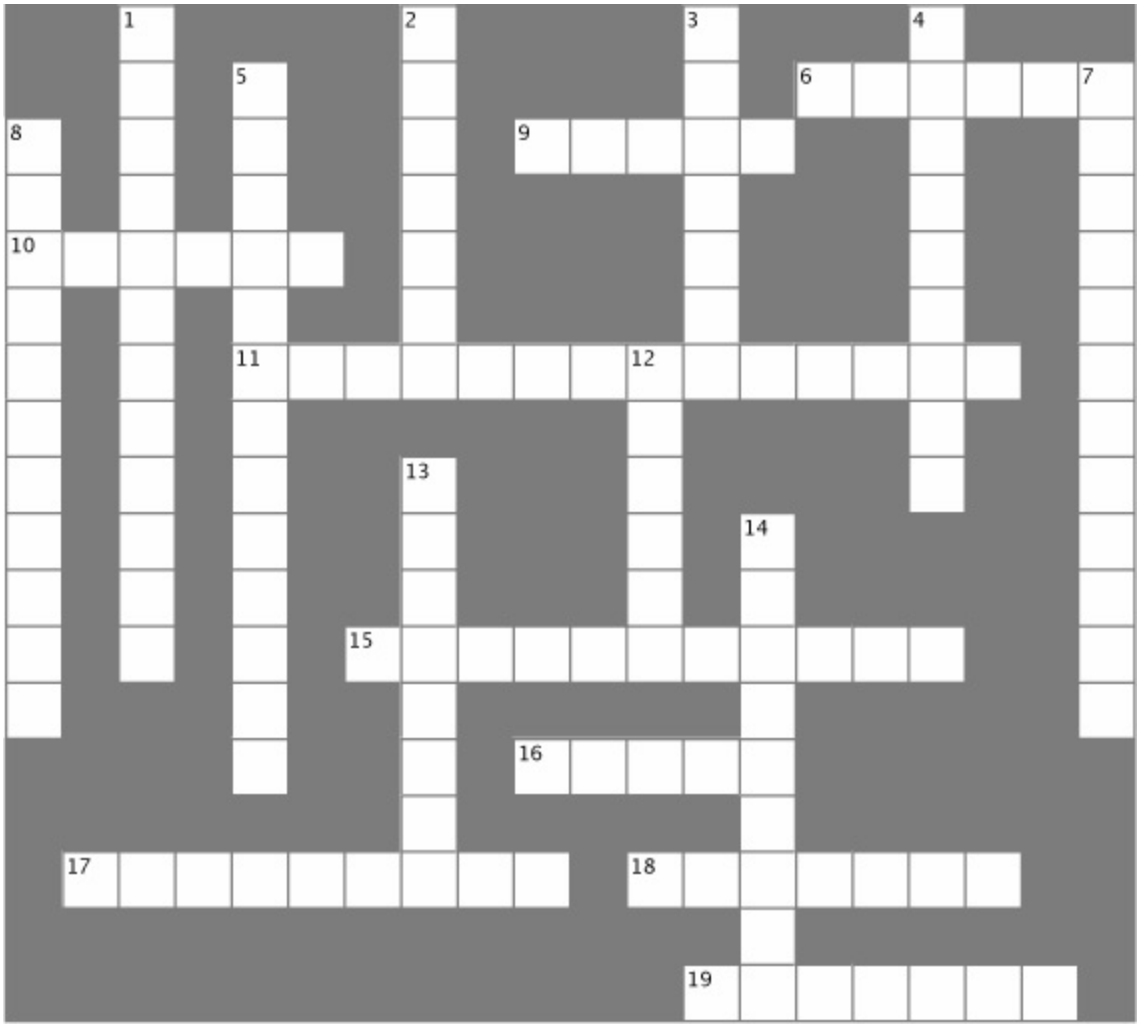
Here's the entire Excuse class.

You'll need `using System.IO;` **and** `using System.Runtime.Serialization.Formatters.Binary;` **in the** `Excuse` **class.**

The constructor for loading random excuses needs to look for the ".excuse" extension instead of "*.txt" files.

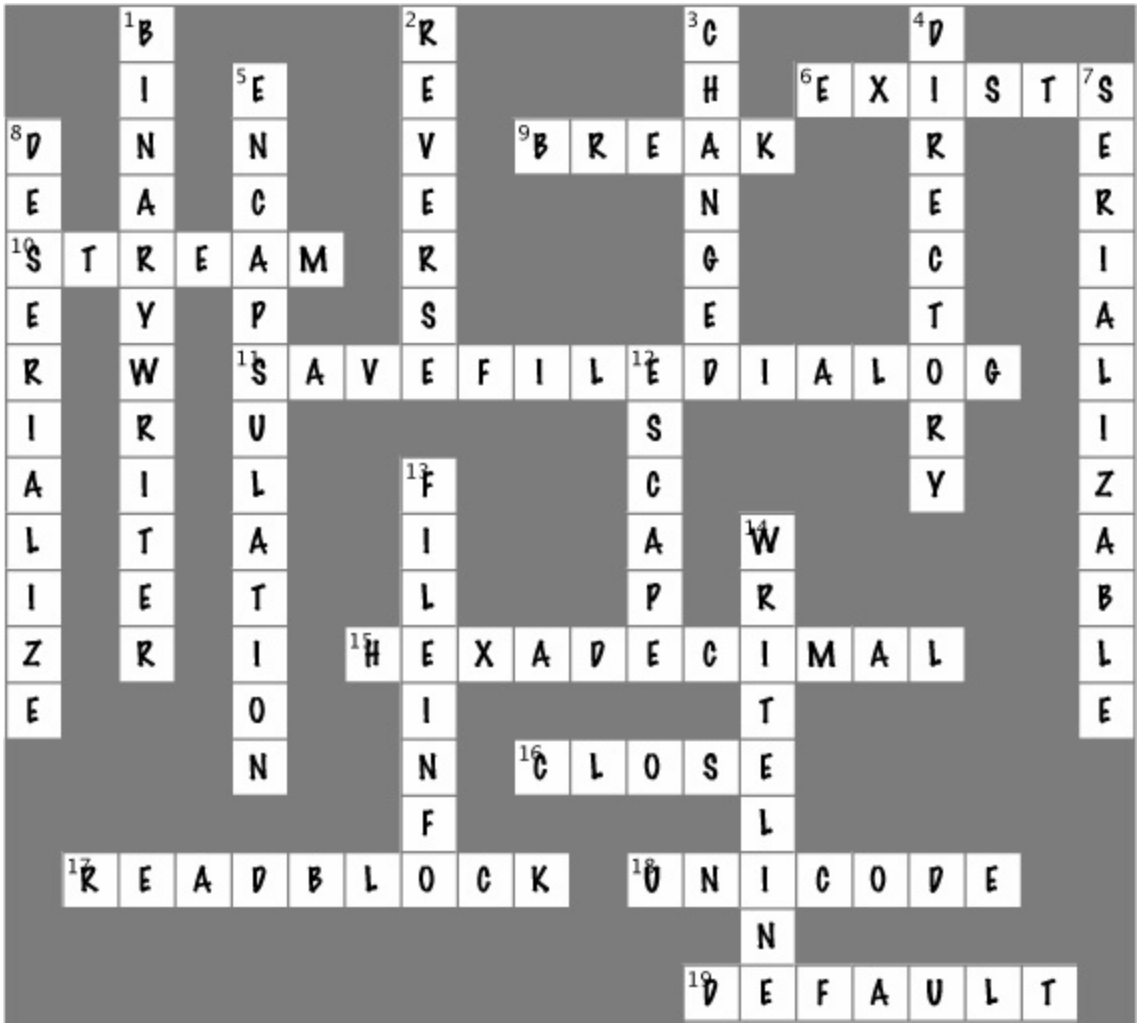
We pass in "this" because we want this class to be serialized.

FILECROSS



Across	Down
6. The method in the `File` class that checks whether or not a specific file is on the drive 9. This statement indicates the end of a case inside a `switch` statement 10. The abstract class that `FileStream` inherits from 11. A nonvisual control that lets you pop up the standard Windows Save As dialog box 15. How you write numbers in base-16 16. If you don't call this method, your stream could be locked open so other methods or programs can't open it 17. The `StreamReader` method that reads data into a `char[]` array 18. An encoding system that assigns a unique number to each character 19. Use this statement to indicate which statements should be executed when the value being tested in a `switch` statement does not match any of the cases	1. This class has a method that writes a type to a file 2. The static method in the `Array` class that turns an array backward 3. The event handler that gets run whenever someone modifies the data in an input control 4. This class has many static methods that let you manipulate folders 5. Using this OOP principle makes it a lot easier to maintain your code 7. If you don't use this attribute to indicate that a class can be written to a stream, `BinaryFormatter` will generate an error 8. This `BinaryFormatter` method reads an object from a stream 12. `\n` and `\r` are examples of this kind of sequence 13. This class lets you perform all the operations in the `File` class for a specific file 14. This method sends text to a stream followed by a line break

FILECROSS SOLUTION



Part II. C# Lab: The Quest

Name:	_____	Date:	_____

This lab gives you a spec that describes a program for you to build, using the knowledge you've gained over the last few chapters.

This project is bigger than the ones you've seen so far. So read the whole thing before you get started, and give yourself a little time. And don't worry if you get stuck — there's nothing new in here, so you can move on in the book and come back to the lab later.

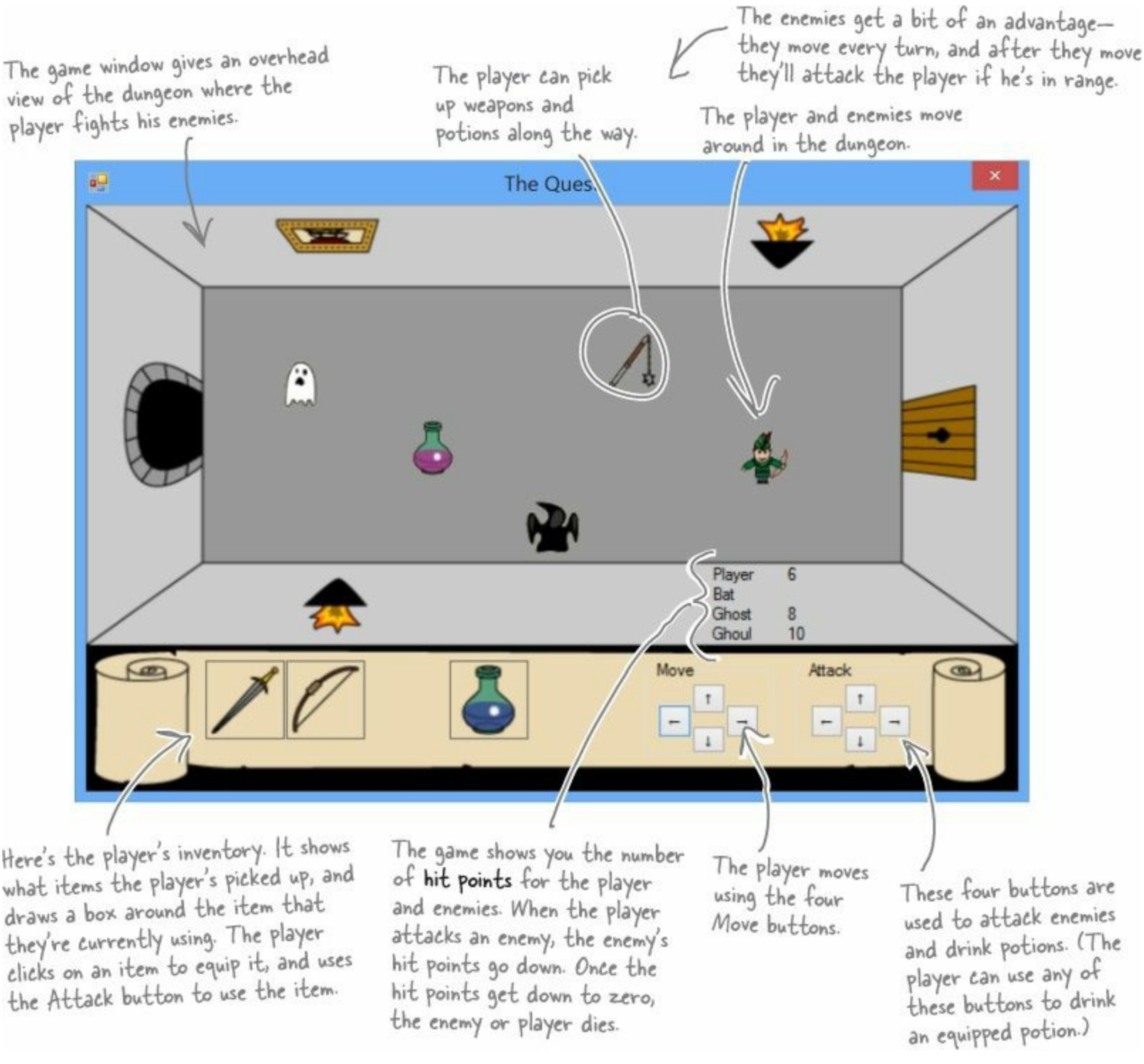
We've filled in a few design details for you, and we've made sure you've got all the pieces you need...and nothing else.

It's up to you to finish the job. There are too many ways to build this lab for us to you a "right" answer. But if you need a hint, other readers have claimed their bragging rights by publishing their solutions on CodePlex, GitHub, and other collaborative source code hosting sites.

The spec: build an adventure game

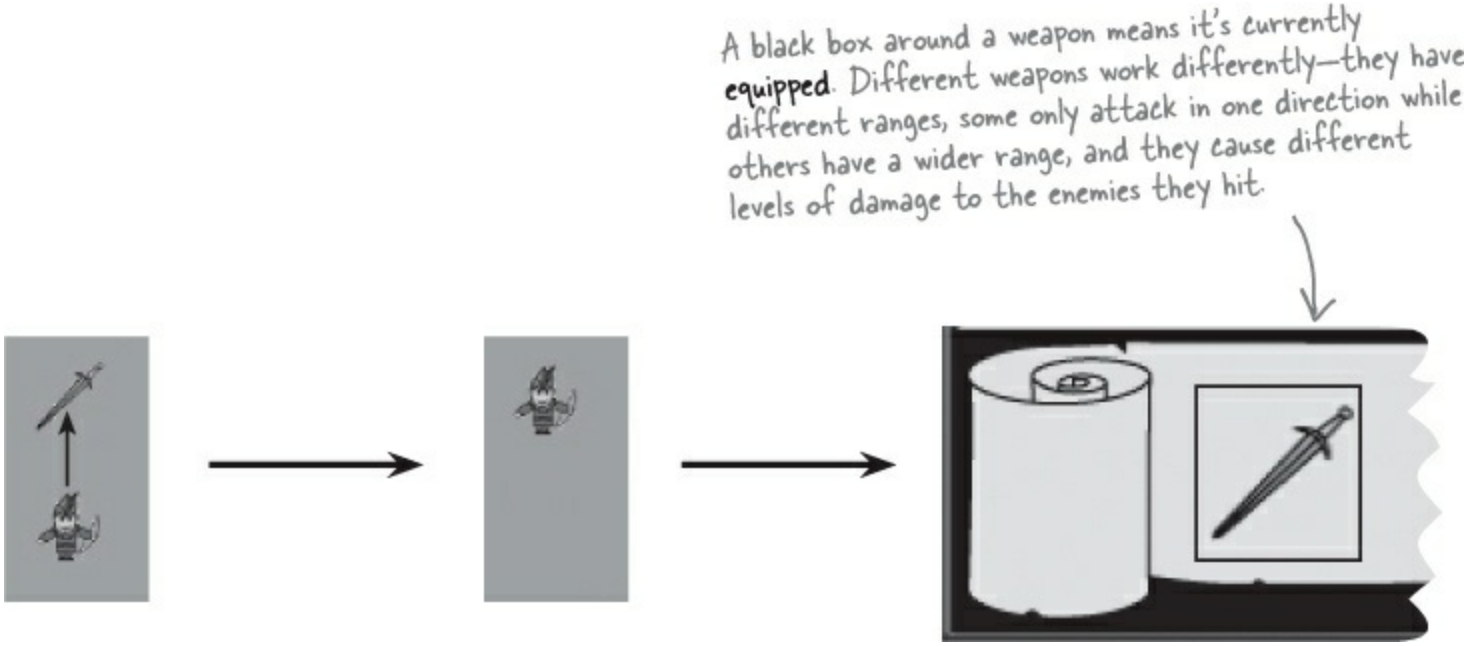
Your job is to build an adventure game where a mighty adventurer is on a quest to defeat level after level of deadly enemies. You'll build a **turn-based system**, which means the player makes one move and then the enemies make one move. The player can move ***or*** attack, and then each enemy gets a chance to move ***and*** attack. The game keeps going until the player either defeats all the enemies on all seven levels or dies.

It's possible to build Windows Desktop programs that automatically scale to any size display, but that's beyond the scope of what we're teaching. (You'll learn all about how to do that with XAML in the next chapter, but that obviously won't help with WinForms.) However, this means that the inventory PictureBoxes, GroupBoxes, and TableLayoutPanel on the form may look right in the designer, but end up in strange places when you run the program. Just drag them so they look right for your screen when you run the program.



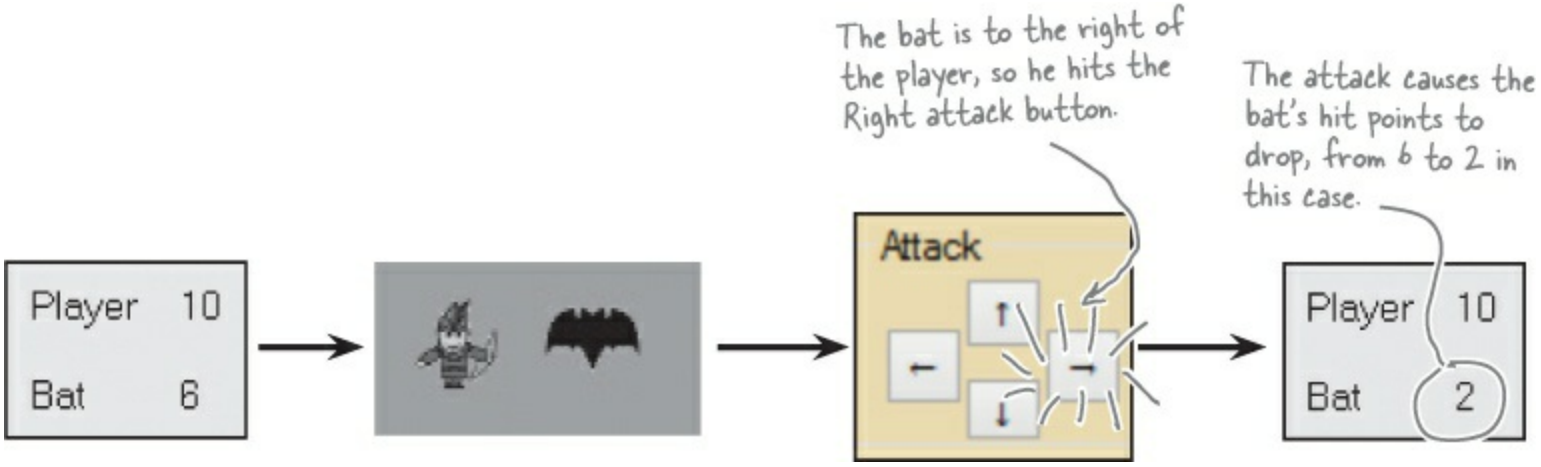
The player picks up weapons...

There are weapons and potions scattered around the dungeon that the player can pick up and use to defeat his enemies. All he has to do is move onto a weapon, and it disappears from the floor and appears in his inventory.



...and attacks enemies with them

Every level in the game has a weapon that the player can pick up and use to defeat his enemies. Once the weapon's picked up, it should disappear from the game floor.



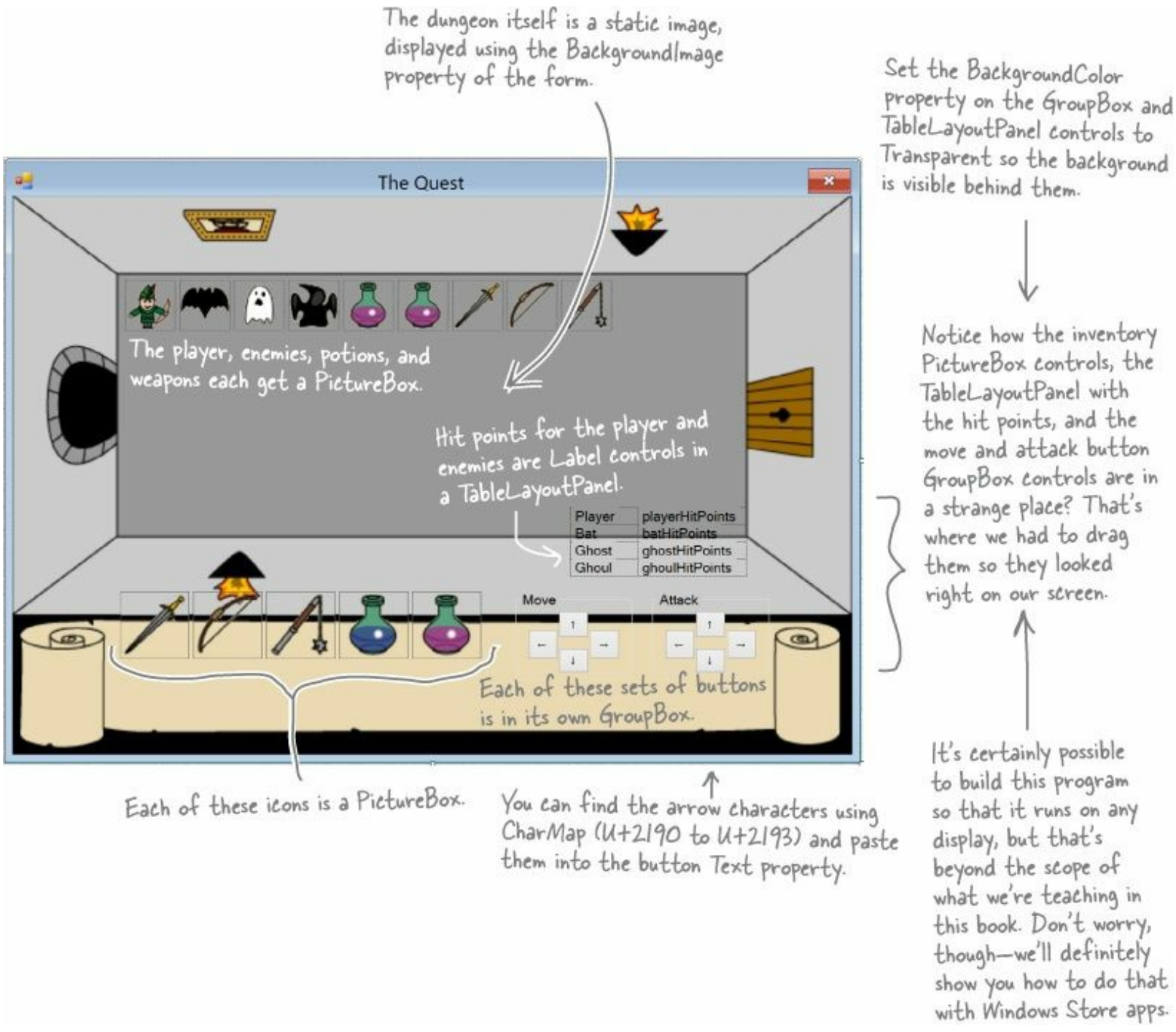
Higher levels bring more enemies

There are three different kinds of enemies: a bat, a ghost, and a ghoul. The first level has only a bat. The seventh level is the last one, and it has all three enemies.

The design: building the form

The form gives the game its unique look. Use the form's `BackgroundImage` property to display the image of the dungeon and the inventory, and a series of PictureBox controls to show the player, weapons, and enemies in the dungeon. You'll use a TableLayoutPanel control to display the hit points for the player, bat, ghost, and ghoul as well as the buttons for moving and attacking.

Use the form's `BackgroundImage` property to set the background image to the dungeon graphic. When you do this, setting controls' background colors to `Transparent` shows the background behind them. Set the `BackgroundImageLayout` property to `Stretch` and the `FormBorderStyle` property to `FixedSingle`, then stretch out the form until there's enough room to add the GroupBoxes and Buttons to the form.

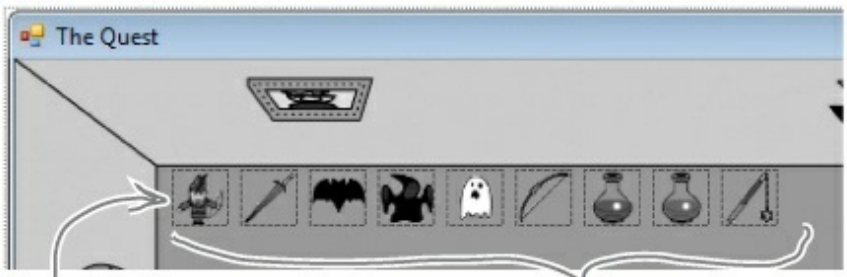


Download the background image and the graphics for the weapons, enemies, and player from the Head First Labs website: www.headfirstlabs.com/books/hfcsharp

Everything in the dungeon is a PictureBox

Players, weapons, and enemies should all be represented by icons. Add nine PictureBox controls, and set their `Visible` properties to `False`. Then, your game can move around the controls, and toggle their `Visible` properties as needed.

You can set a PictureBox's BackColor property to Color.Transparent to let the form's background picture or color show through any transparent pixels in the picture.



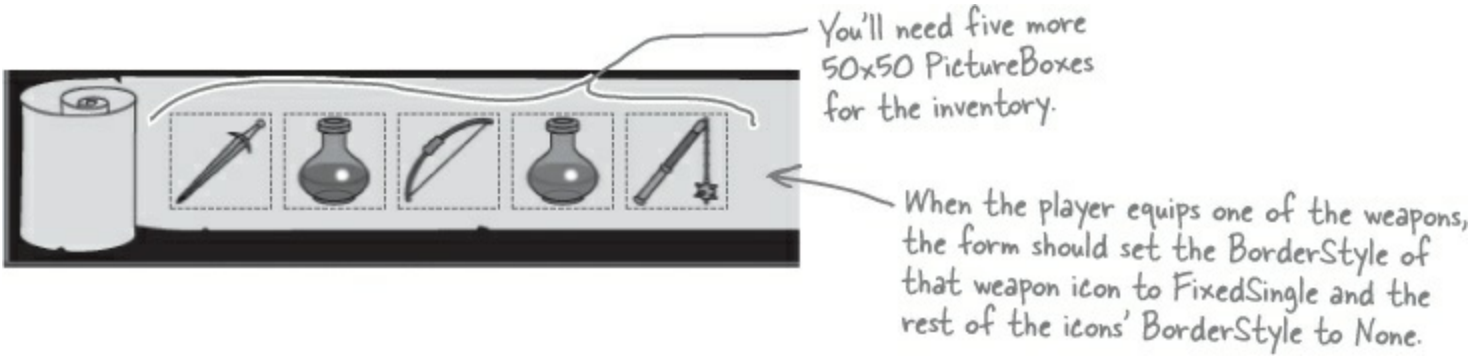
Add nine PictureBox controls to the dungeon. Use the Size property to make each one 30x30. It doesn't matter where you place them—the form will move them around. Use the little black arrow that shows up when you click on the PictureBox to set each to one of the images from the Head First Labs website.

After you've added the nine PictureBox controls, right-click on the player's icon and select "Bring to Front," then send the weapon and potion icons to the back. That ensures player icon stays "above" any items that are picked up.

Controls overlap each other in the IDE, so the form needs to know which ones are in front, and which are in back. That's what the "Bring to Front" and "Send to Back" form designer commands do.

The inventory contains PictureBox controls, too

You can represent the inventory of the player as five 50×50 PictureBox controls. Set the `BackColor` property of each to **`Color.Transparent`** (if you use the Properties window to set the property, just type it into the `BackColor` row). Since the picture files have a transparent background, you'll see the scroll and dungeon behind them:

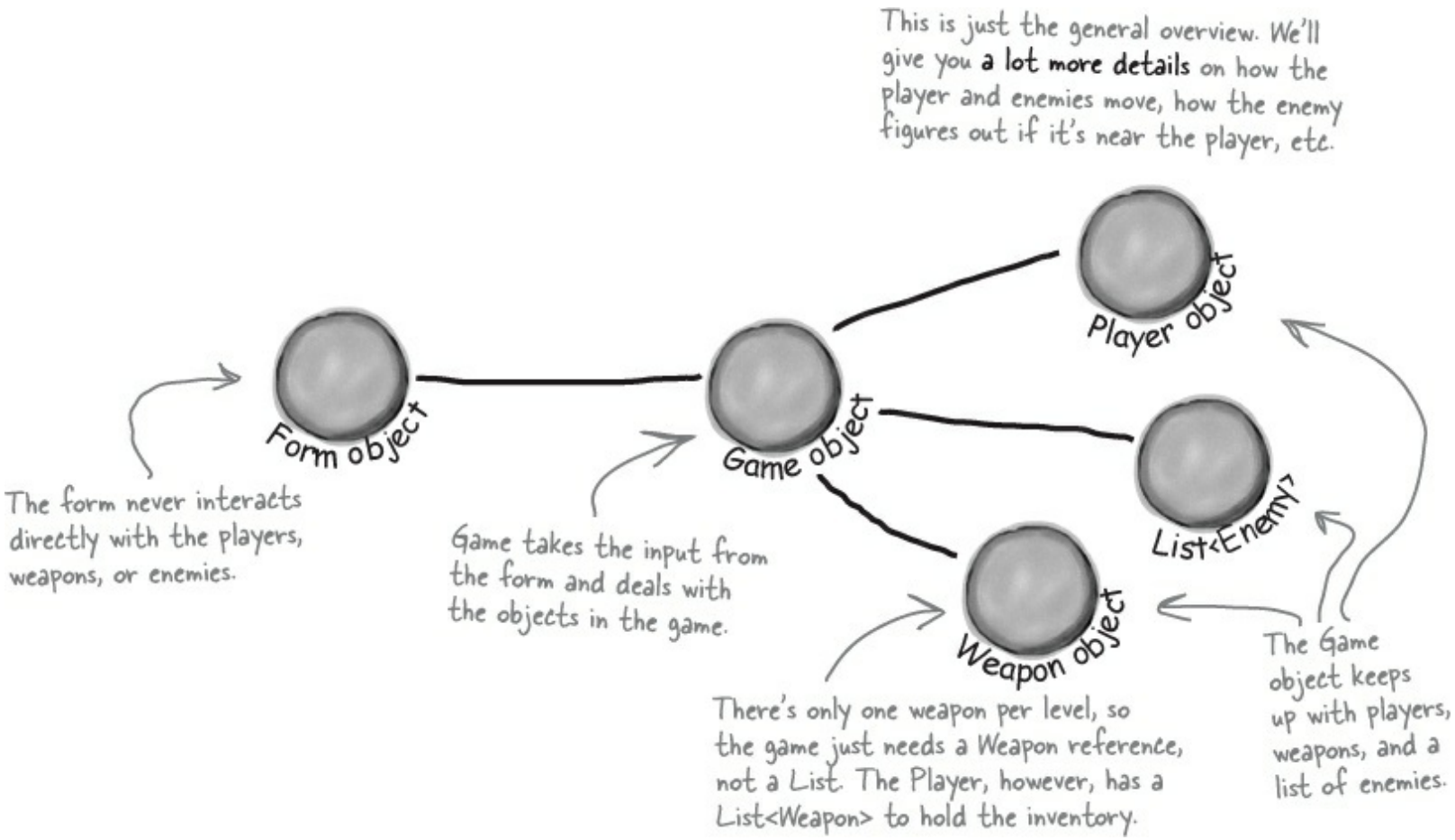


Build your stats window

The hit points are in a `TableLayoutPanel`, just like the attack and movement buttons. For the hit points, create two columns in the panel, and drag the column divider to the left a bit. Add four rows, each 25% height, and add in `Label` controls to each of the eight cells:

The architecture: using the objects

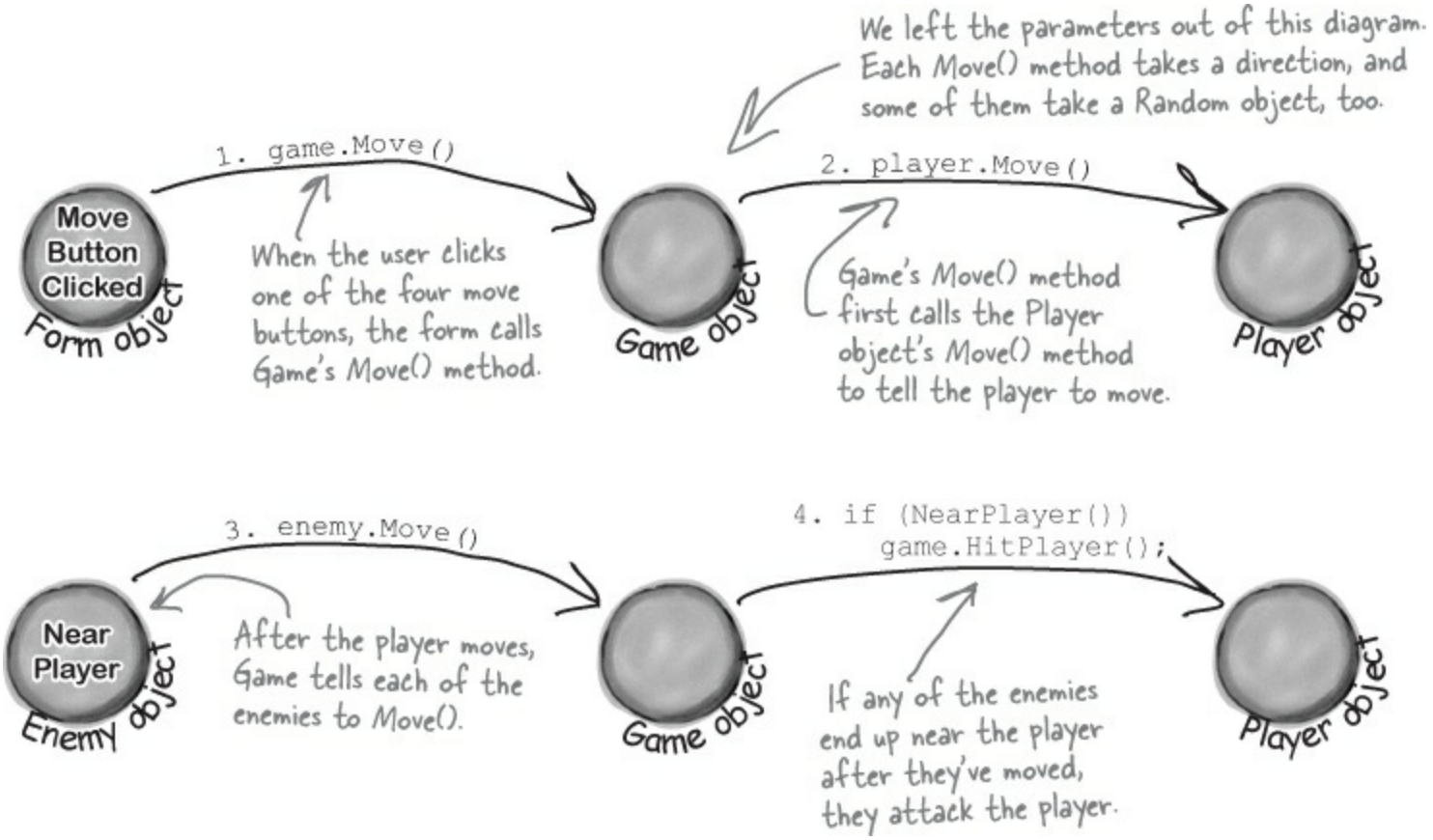
You'll need several types of objects in your game: a `Player` object, several subclasses of an `Enemy` object, and several subclasses of a `Weapon` object. And you'll also need one object to keep up with everything that's going on: the `Game` object.



The Game object handles turns

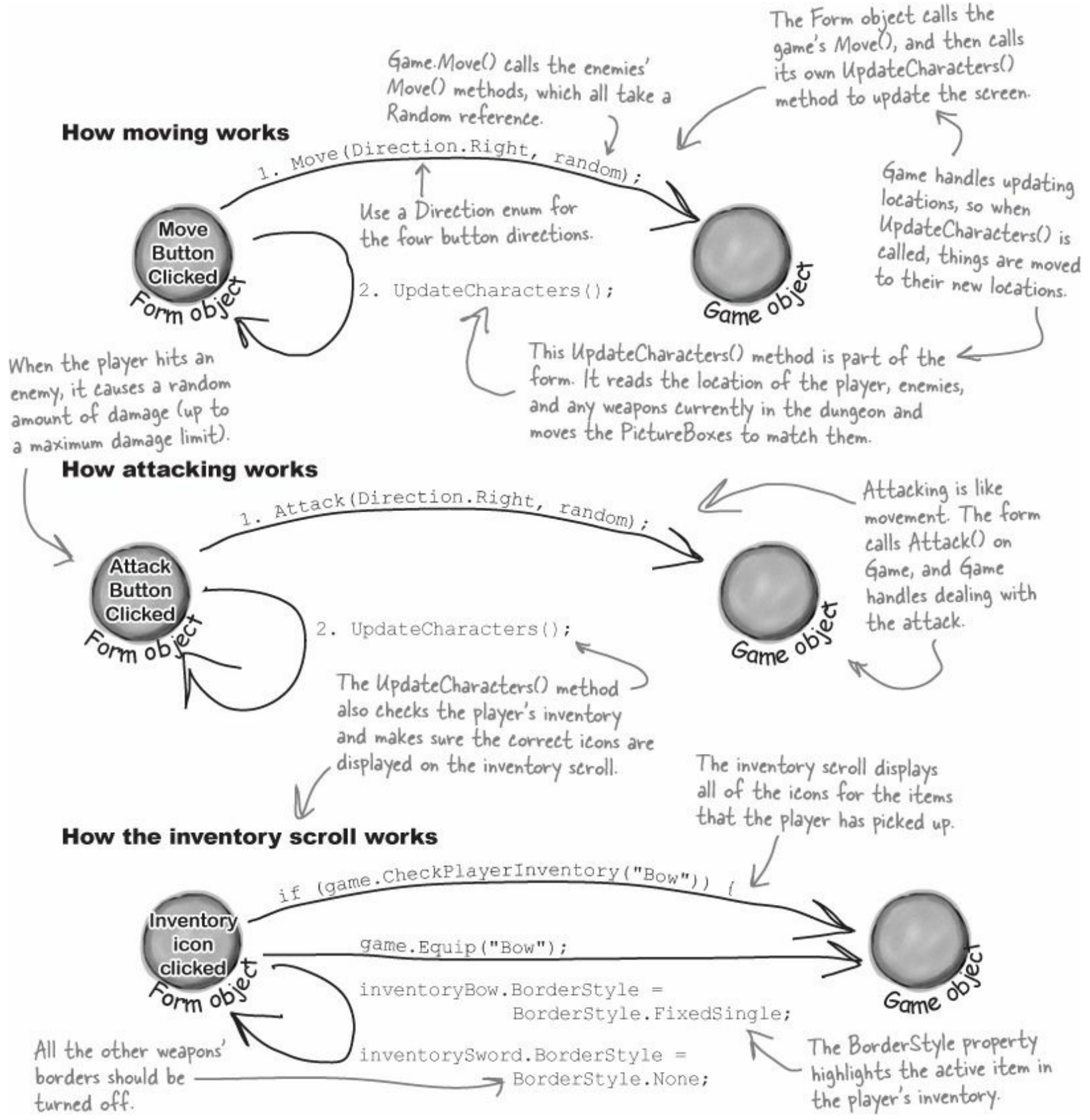
When one of your form's movement buttons is clicked, the form will call the `Game` object's `Move()` method. That method will let the player take a turn, and then let all the enemies move. So it's up to `Game` to handle the turn-based movement portion of the game.

For example, here's how the move buttons work:



Gameplay concerns are separated into the Game object

Movement, attacking, and inventory all begin in the form. So clicking a movement or attack button, or an item in inventory, triggers code in your form. But it's the `Game` object that controls the objects in the game. So the form has to pass on anything that happens to the `Game` object, and then the `Game` object takes it from there:



Building the Game class

We've gotten you started with the `Game` class in the code below. There's a lot for you to do — so read through this code carefully, get it into the IDE, and get ready to go to work:

```
using System.Drawing;

class Game {
  public IEnumerable<Enemy> Enemies { get; private set; }
  public Weapon WeaponInRoom { get; private set; }

  private Player player;
  public Point PlayerLocation { get { return player.Location; } }
  public int PlayerHitPoints { get { return player.HitPoints; } }
  public IEnumerable<string> PlayerWeapons { get { return player.Weapons; } }
  private int level = 0;
  public int Level { get { return level; } }

  private Rectangle boundaries;
  public Rectangle Boundaries { get { return boundaries; } }

  public Game(Rectangle boundaries) {
    this.boundaries = boundaries;
    player = new Player(this,
      new Point(boundaries.Left + 10, boundaries.Top + 70));
  }
  public void Move(Direction direction, Random random) {
    player.Move(direction);
    foreach (Enemy enemy in Enemies)
      enemy.Move(random);
  }
  public void Equip(string weaponName) {
    player.Equip(weaponName);
  }
  public bool CheckPlayerInventory(string weaponName) {
    return player.Weapons.Contains(weaponName);
  }
  public void HitPlayer(int maxDamage, Random random) {
    player.Hit(maxDamage, random);
  }
```

You'll need Rectangle and Point from System.Drawing, so be sure to add this to the top of your class.

These are OK as public properties if Enemy and Weapon are well encapsulated...in other words, just make sure the form can't do anything inappropriate with them.

The game keeps a private Player object. The form will only interact with this through methods on Game, rather than directly.

The Rectangle object has Top, Bottom, Left, and Right fields, and works perfectly for the overall game area.

Game starts out with a bounding box for the dungeon, and creates a new Player object in the dungeon.

Movement is simple: move the player in the direction the form gives us, and move each enemy in a random direction.

These are examples of encapsulation.... Game doesn't know how Player handles these actions; it just passes on the needed information and lets Player do the rest.

```
  public void IncreasePlayerHealth(int health, Random random) {
    player.IncreaseHealth(health, random);
  }

  public void Attack(Direction direction, Random random) {
    player.Attack(direction, random);
    foreach (Enemy enemy in Enemies)
      enemy.Move(random);
  }

  private Point GetRandomLocation(Random random) {
    return new Point(boundaries.Left +
      random.Next(boundaries.Right / 10 - boundaries.Left / 10) * 10,
          boundaries.Top +
      random.Next(boundaries.Bottom / 10 - boundaries.Top / 10) * 10);
  }

  public void NewLevel(Random random) {
    level++;
    switch (level) {
      case 1:
        Enemies = new List<Enemy>() {
           new Bat(this, GetRandomLocation(random)),
        };
        WeaponInRoom = new Sword(this, GetRandomLocation(random));
        break;

    }
  }
}
```

Attack() is almost exactly like Move(). The player attacks, and the enemies all get a turn to move.

GetRandomLocation() will come in handy in the NewLevel() method, which will use it to determine where to place enemies and weapons.

This is just a math trick to get a random location within the rectangle that represents the dungeon area.

We only added the case for Level 1. It's your job to add cases for the other levels.

Finish the rest of the levels

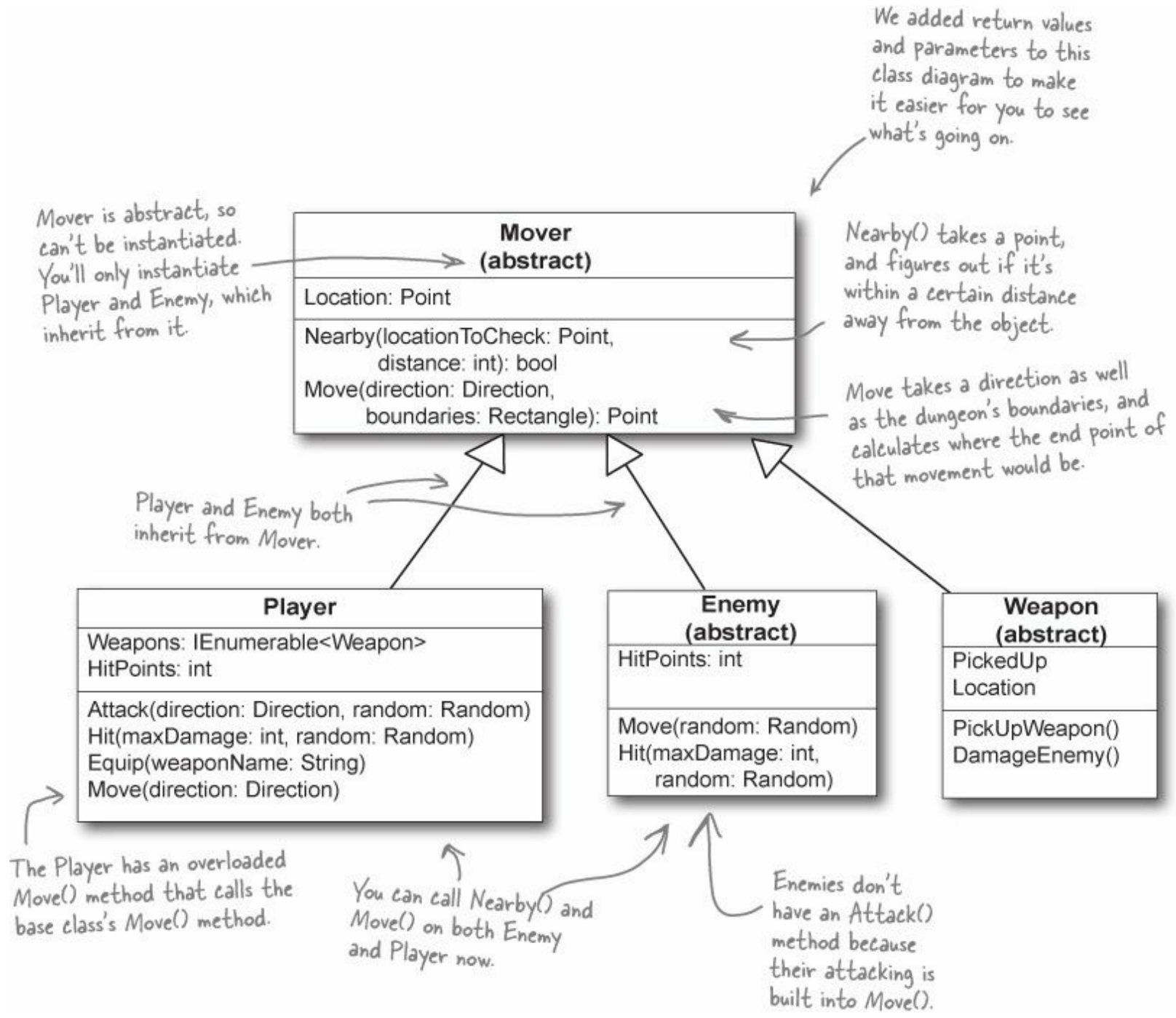
It's your job to finish the `NewLevel()` method. Here's the breakdown for each level:

Level	Enemies	Weapons
2	Ghost	Blue potion **NOTE** We've only got room in the inventory for one blue potion and one red potion. So if the player already has a red potion, then the game shouldn't add a red potion to the level (and the same goes for the blue potion). So if the blue potion is still in the player's inventory from Level 2, nothing appears on this level.
3	Ghoul	Bow
4	Bat, Ghost	Bow, if not picked up on 3; otherwise, blue potion
5	Bat, Ghoul	Red potion
6	Ghost, Ghoul	Mace
7	Bat, Ghost, Ghoul	Mace, if not picked up on 6; otherwise, red potion **NOTE** This only appears if the red potion from Level 5 has already been used up.
8	N/A	N/A - end the game with Application.Exit()

Finding common behavior: movement

You already know that duplicate code is bad, and duplicate code usually shows up when two or more objects share the same behavior. Well, you've got a player that moves and enemies that move, so you'll need a way to avoid having the same movement code duplicated in all of those classes.

Create an abstract `Mover` class to put that common behavior into a single place. `Player` and `Enemy` will inherit from `Mover`. And even though weapons don't move around *much*, they have a location and need to be placed in the dungeon, so they'll inherit from `Mover`, too. `Mover` has a `Move()` method for moving around the dungeon, and a read-only `Location` property that the form can use to position a subclass of `Mover`.



Add a Direction enum

The `Mover` class, as well as several other classes, need a `Direction` enum. Create this enum, and give it four enumerated values: `Up`, `Down`, `Left`, and `Right`.

The Mover class source code

Here's the code for `Mover`:

```
abstract class Mover {
  private const int MoveInterval = 10;
  protected Point location;
  public Point Location { get { return location; } }
  protected Game game;

  public Mover(Game game, Point location) {
    this.game = game;
    this.location = location;
  }

  public bool Nearby(Point locationToCheck, int distance) {
    if (Math.Abs(location.X - locationToCheck.X) < distance &&
       (Math.Abs(location.Y - locationToCheck.Y) < distance)) {
      return true;
    } else {
      return false;
    }
  }
  public Point Move(Direction direction, Rectangle boundaries) {
    Point newLocation = location;
    switch (direction) {
      case Direction.Up:
        if (newLocation.Y - MoveInterval >= boundaries.Top)
          newLocation.Y -= MoveInterval;
        break;
      case Direction.Down:
        if (newLocation.Y + MoveInterval <= boundaries.Bottom)
          newLocation.Y += MoveInterval;
        break;
      case Direction.Left:
        if (newLocation.X - MoveInterval >= boundaries.Left)
          newLocation.X -= MoveInterval;
        break;
      case Direction.Right:
        if (newLocation.X + MoveInterval <= boundaries.Right)
          newLocation.X += MoveInterval;
        break;
      default: break;
    }
    return newLocation;
  }
}
```

You can change MoveInterval if you want your player and enemies to move in bigger or smaller steps.

Since protected properties are only available to subclasses, the form object can't set the location...only read it through the public get method we define.

Instances of Mover take in the Game object and a current location.

The Nearby method checks a Point against this object's current location. If they're within distance of each other, then it returns true; otherwise, it returns false.

The Move() method tries to move one step in a direction. If it can, it returns the new Point. If it hits a boundary, it returns the original Point.

If the end location is outside the boundaries, the new location stays the same as the starting point.

Finally, this new location is returned (which might still be the same as the starting location!).

The Player class keeps track of the player

Here's a start on the `Player` class. Start with this code in the IDE, and then get ready to add to it.

NOTE

The Player and Enemy objects need to stay inside the dungeon, which means they need to know the boundaries of the playing area. Use the Contains() method of the boundaries Rectangle to make sure they don't move out of bounds.

```
class Player : Mover {
  private Weapon equippedWeapon;

  public int HitPoints { get; private set; }

  private List<Weapon> inventory = new List<Weapon>();
  public IEnumerable<string> Weapons {
    get {
      List<string> names = new List<string>();
      foreach (Weapon weapon in inventory)
        names.Add(weapon.Name);
      return names;
    }
  }

  public Player(Game game, Point location)
      : base(game, location) {
    HitPoints = 10;
  }

  public void Hit(int maxDamage, Random random) {
    HitPoints -= random.Next(1, maxDamage);
  }

  public void IncreaseHealth(int health, Random random) {
    HitPoints += random.Next(1, health);
  }

  public void Equip(string weaponName) {
    foreach (Weapon weapon in inventory) {
      if (weapon.Name == weaponName)
        equippedWeapon = weapon;
    }
  }
}
```

The Weapons property returns a collection of strings with the weapon names.

Player inherits from Mover, so this passes in the Game and location to that base class.

The player's constructor sets its HitPoints to 10 after the base constructor is called.

When an enemy hits the player, it causes a random amount of damage. And when a potion increases the player's health, it increases it by a random amount.

The Equip() method tells the player to equip one of his weapons. The Game object calls this method when one of the inventory icons is clicked.

A Player object can only have one Weapon object equipped at a time.

Even though potions help the player rather than hurt the enemy, they're still **considered weapons** by the game. That way the inventory can be a List<Weapon>, and the game can point to one with its WeaponInRoom reference.

Write the Move() method for the Player

`Game` calls the `Player`'s `Move()` method to tell a player to move in a certain direction. `Move()` takes the direction to move as an argument (using the `Direction` enum you should have already added). Here's the start of that method:

> **NOTE**
>
> This happens when one of the movement buttons on the form is clicked.

```
public void Move(Direction direction) {
  base.location = Move(direction, game.Boundaries);
  if (!game.WeaponInRoom.PickedUp) {
    // see if the weapon is nearby, and possibly pick it up
  }
}
```

Move is in the Mover base class.

You'll fill in the rest of this method. Check and see if the weapon is near the player (within a single unit of distance). If so, pick up the weapon and add it to the player's inventory.

> **NOTE**
>
> When the player picks up a weapon, it needs to disappear from the dungeon and appear in the inventory.
>
> The Weapon and form will handle making the weapon's PictureBox invisible when the player picks it up...that's not the job of the Player class.

If the weapon is the only one that the player has, go ahead and equip it immediately. That way, the player can use it right away, on the next turn.

Add an Attack() method, too

Next up is the `Attack()` method. This is called when one of the form's attack buttons is clicked, and carries with it a direction (again, from the `Direction` enum). Here's the method signature:

> **NOTE**
>
> The weapons all have an Attack() method that takes a Direction enum and a Random object. The player's Attack() will figure out which weapon is equipped and call its Attack().
>
> If the weapon is a potion, then Attack() removes it from the inventory after the player drinks it.

```
public void Attack(Direction direction, Random random) {
  // Your code goes here
}
```

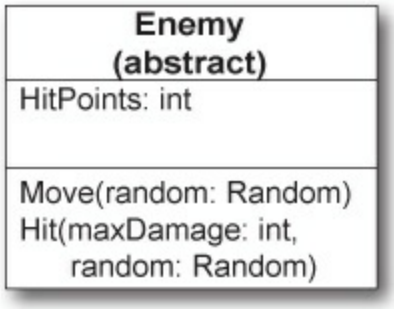
If the player doesn't have an equipped weapon, this method won't do anything. If the player does have an equipped weapon, this should call the weapon's `Attack()` method.

But potions are a special case. If a potion is used, remove it from the player's inventory, since it's not available anymore.

> **NOTE**
>
> Potions will implement an IPotion interface (more on that in a minute), so you can use the "is" keyword to see if a Weapon is an implementation of IPotion.

Bats, ghosts, and ghouls inherit from the Enemy class

We'll give you another useful abstract class: `Enemy`. Each different sort of enemy has its own class that inherits from the `Enemy` class. The different kinds of enemies move in different ways, so the `Enemy` abstract class leaves the `Move` method as an abstract method — the three enemy classes will need to implement it differently, depending on how they move.




```
abstract class Enemy : Mover {
  private const int NearPlayerDistance = 25;

  public int HitPoints { get; private set; }
  public bool Dead { get {
       if (HitPoints <= 0) return true;
       else return false;
    }
  }
  public Enemy(Game game, Point location, int hitPoints)
       : base(game, location) { HitPoints = hitPoints; }

  public abstract void Move(Random random);

  public void Hit(int maxDamage, Random random) {
     HitPoints -= random.Next(1, maxDamage);
  }

  protected bool NearPlayer() {
    return (Nearby(game.PlayerLocation, NearPlayerDistance));
  }

  protected Direction FindPlayerDirection(Point playerLocation) {
    Direction directionToMove;
       if (playerLocation.X > location.X + 10)
          directionToMove = Direction.Right;
    else if (playerLocation.X < location.X - 10)
          directionToMove = Direction.Left;
    else if (playerLocation.Y < location.Y - 10)
          directionToMove = Direction.Up;
    else
          directionToMove = Direction.Down;
     return directionToMove;
  }
}
```

The form can use this read-only property to see if the enemy should be visible in the game dungeon.

Each subclass of Enemy implements this.

When the player attacks an enemy, it calls the enemy's Hit() method, which subtracts a random number from the hit points.

The Enemy class inherited the Nearby() method from Mover, which it can use to figure out whether it's near the player.

If you feed FindPlayerDirection() the player's location, it'll use the base class's location field to figure out where the player is in relation to the enemy and return a Direction enum that tells you in which direction the enemy needs to move in order to move toward the player.

Write the different Enemy subclasses

The three subclasses of `Enemy` are pretty straightforward. Each enemy has a different number of starting hit points, moves differently, and does a different amount of damage when it attacks. You'll need to have each one pass a different `startingHitPoints` parameter to the `Enemy` base constructor, and you'll have to write different `Move()` methods for each subclass.

Here's an example of how one of those classes might look:

```
class Bat : Enemy {
    public Bat(Game game, Point location)
        : base(game, location, 6)
    { }
    public override void Move(Random random) {
        // Your code will go here
    }
}
```

The bat starts with six hit points, so it passes 6 to the base class constructor.

You probably won't need a subclass constructor for these; the base class handles everything.

The bat starts with six hit points. It'll keep moving toward the player and attacking **as long as it has one or more hit points.** When it moves, there's a 50% chance that it'll move toward the player, and a 50% chance that it'll move in a random direction. After the bat moves, it checks if it's near the player — if it is, then it attacks the player with up to two hit points of damage.

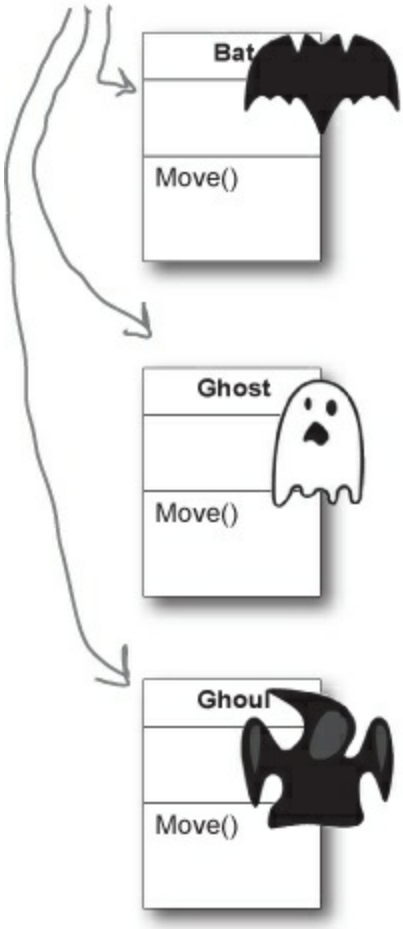
> **NOTE**
>
> The bat flies around somewhat randomly, so it uses Random to fly in a random direction half the time.
>
> Once an enemy has no more hit points, the form will no longer display it. But it'll still be in the game's Enemies list until the player finishes the level.
>
> We'll have to make sure the form sees if an enemy should be visible at every turn.

The ghost is harder to defeat than the bat. But like the bat, it will only move and attack if its hit points are greater than zero. It starts with eight hit points. When it moves, there's a one in three chance that it'll move toward the player, and a two in three chance that it'll stand still. If it's near the player, it attacks the player with up to three hit points of damage.

The ghoul is the toughest enemy. It starts with 10 hit points, and only moves and attacks if its hit points are greater than zero. When it moves, there's a two in three chance that it'll move toward the player, and a one in three chance that it'll stand still. If it's near the player, it attacks the player with up to four hit points of damage.

> **NOTE**
>
> The ghost and ghoul use Random to make them move more slowly than the player.

Each of these subclasses the Enemy base class, which in turn subclasses Mover.



Weapon inherits from Mover; each weapon inherits from Weapon

We need a base `Weapon` class, just like we had a base `Enemy` class. And each weapon has a location, as well as a property indicating whether or not it's been picked up. Here's the base `Weapon` class:

Weapon inherits from Mover because it uses its Nearby() and Move() methods in DamageEnemy().

```
abstract class Weapon : Mover {

  public bool PickedUp { get ; private set; }

  public Weapon(Game game, Point location) {
    : base(game, location)
  {
    PickedUp = false;
  }

  public void PickUpWeapon() { PickedUp = true; }

  public abstract string Name { get; }

  public abstract void Attack(Direction direction, Random random);

  protected bool DamageEnemy(Direction direction, int radius,
                             int damage, Random random) {
    Point target = game.PlayerLocation;
    for (int distance = 0; distance < radius / 2; distance++)
      foreach (Enemy enemy in game.Enemies) {
        if (Nearby(enemy.Location, target, distance)) {
          enemy.Hit(damage, random);
          return true;
        }
      }
      target = Move(direction, target, game.Boundaries);
    }
    return false;
  }
}
```

A picked up weapon shouldn't be displayed anymore...the form can use this get accessor to figure that out.

The constructor calls the Mover base constructor (which sets the game and location fields), and then sets pickedUp to false (because it hasn't been picked up yet).

Each weapon class needs to implement a Name property and an Attack() method that determines how that weapon attacks.

Each weapon's Name property returns its name ("Sword", "Mace", "Bow").

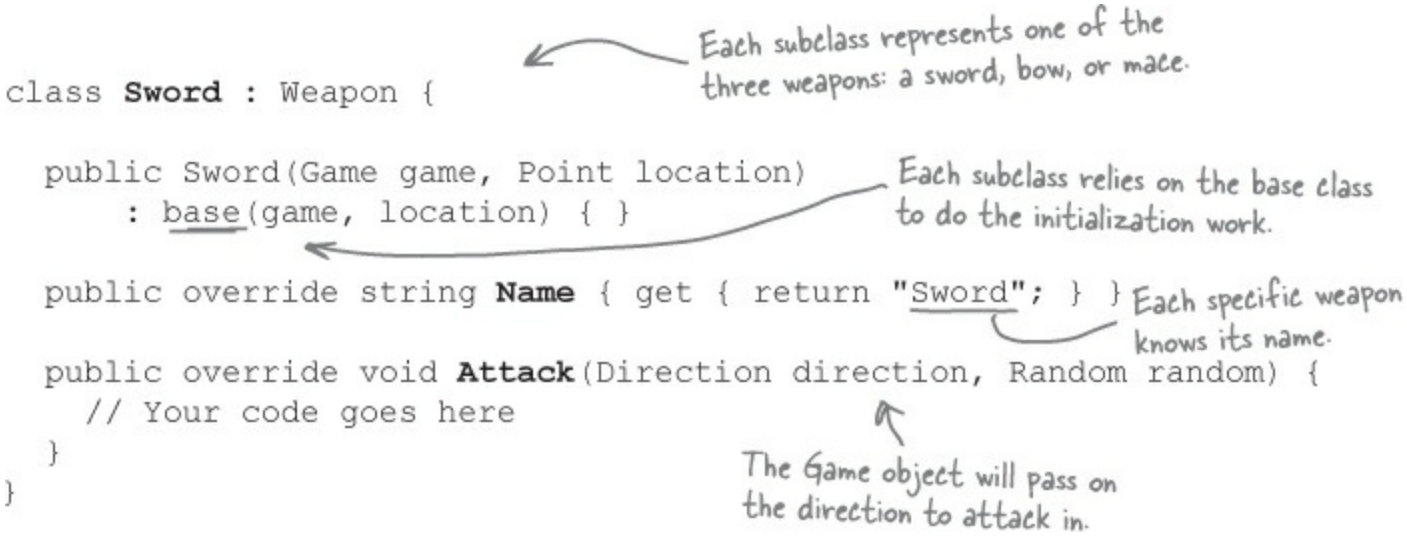
Each weapon has a different range and pattern of attack, so the weapons implement the Attack() method differently.

The DamageEnemy() method is called by Attack(). It attempts to find an enemy in a certain direction and radius. If it does, it calls the enemy's Hit() method and returns true. If no enemy's found, it returns false.

We gave you a simple DamageEnemy() method that checks a series of points in the direction the weapon is attacking, and returns `true` if an enemy is near that point. Take a close look at the iterator in the `for` loop (`distance < radius / 2`). Can you figure out why we're dividing the weapon's radius by 2? The Nearby() method in the `Mover` class takes only two parameters, a `Point` and an `int`, compares the `Point` to the current location, and returns `true` if the `Point` is near the location. For the `DamageEnemy` calculation, you'll need to add an overloaded Nearby() method that compares two points and returns `true` if they're within the specified distance of each other. You'll also need an overloaded `Move` method to move a `Point` in a direction and return the new `Point`. Can you figure out how to modify the Nearby() and Move() methods so that the overloaded methods don't duplicate code?

Different weapons attack in different ways

Each subclass of `Weapon` has its own name and attack logistic. Your job is to implement these classes. Here's the basic skeleton for a `Weapon` subclass:




```
class Sword : Weapon {

  public Sword(Game game, Point location)
      : base(game, location) { }

  public override string Name { get { return "Sword"; } }

  public override void Attack(Direction direction, Random random) {
    // Your code goes here
  }
}
```

NOTE

The player can use the weapons over and over — they never get dropped or used up.

- The sword is the first weapon the player picks up. It's got a wide angle of attack: if he attacks up, then it first tries to attack an enemy that's in that direction. If there's no enemy there, it looks in the direction that's clockwise from the original attack and attacks any enemy there. If it still fails to hit, then it attempts to attack an enemy counterclockwise from the original direction of attack. It's got a radius of 10, and causes 3 points of damage.

  NOTE

  Think carefully about this...what is to the right of the direction left? What is to the left of up?

- The bow has a very narrow angle of attack, but it's got a very long range — it's got an attack radius of 30, but only causes 1 point of damage. Unlike the sword, which attacks in three directions (because the player swings it in a wide arc), when the player shoots the bow in a direction, it only shoots in that one direction.

- The mace is the most powerful weapon in the dungeon. It doesn't matter in which direction the player attacks with it — since he swings it in a full circle, it'll attack any enemy within a radius of 20 and cause up to 6 points of damage.

NOTE

The different weapons will call DamageEnemy() in various ways. The Mace attacks in all directions, so if the player's attacking to the right, it'll call DamageEnemy(Direction.Right, 20, 6, random). If that didn't hit an enemy, it'll attack Up. If there's no enemy there, it'll try Left, then Down — that makes it swing in a full circle.

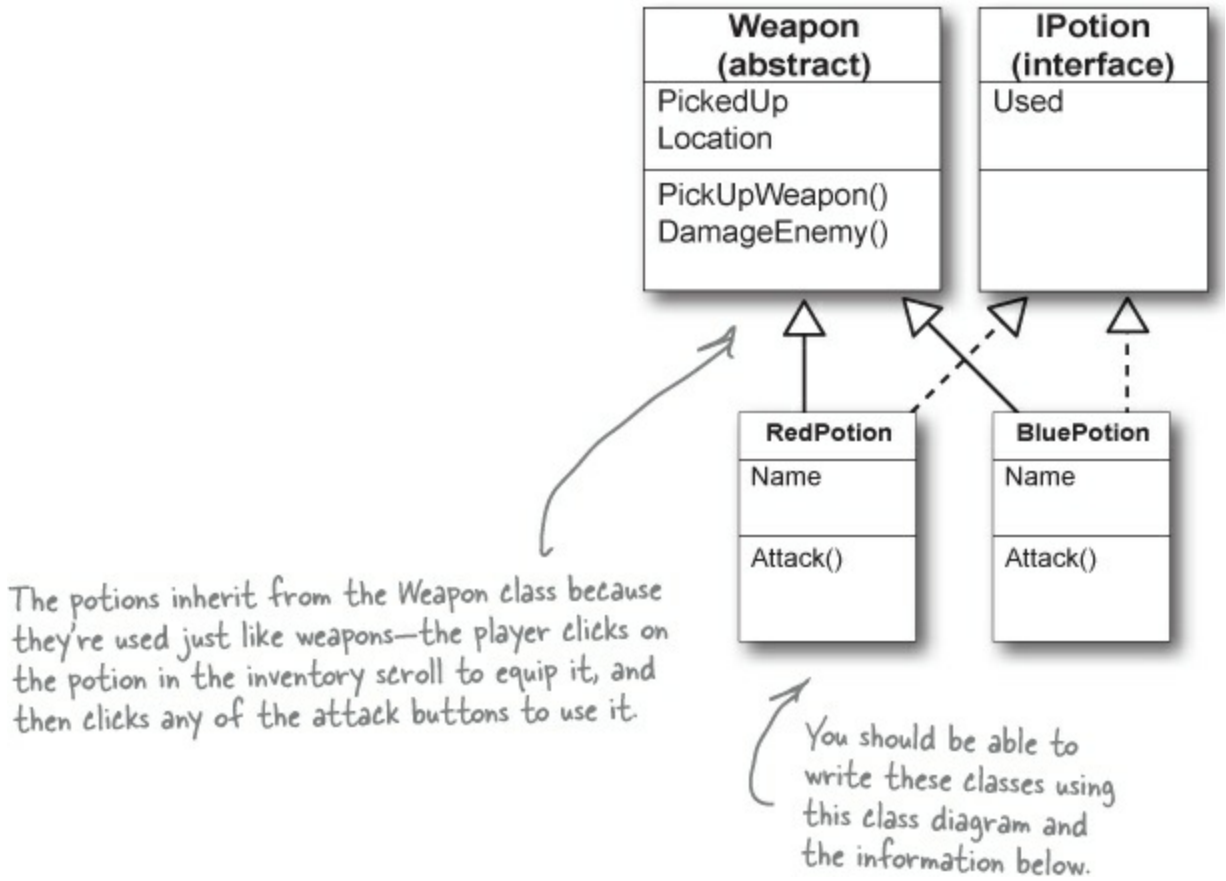
Potions implement the IPotion interface

There are two potions, a blue potion and a red potion, which increase the player's health. They act just like weapons — the player picks them up in the dungeon, equips them by clicking on the inventory, and **uses them by clicking one of the attack buttons**. So it makes sense for them to inherit from the abstract `Weapon` class.

But potions act a little differently, too, so you'll need to add an `IPotion` interface so they can have extra behavior: increasing the player's health. The `IPotion` interface is really simple. Potions only need to add one read-only property called `Used` that returns `false` if the player hasn't used the potion, and `true` if he has. The form will use it to determine whether or not to display the potion in the inventory.

```
interface IPotion {
  bool Used { get; }
}
```

IPotion makes potions usable only once. It's also possible to find out if a Weapon is a potion with "if (weapon is IPotion)" because of this interface.



The potions inherit from the Weapon class because they're used just like weapons—the player clicks on the potion in the inventory scroll to equip it, and then clicks any of the attack buttons to use it.

You should be able to write these classes using this class diagram and the information below.

- The `BluePotion` class's `Name` property should return the string `Blue Potion`. Its `Attack()` method will be called when the player uses the blue potion — it should increase the player's health by up to five hit points by calling the `IncreasePlayerHealth()` method. After the player uses the potion, the potion's `Used` property should return `true`.

  NOTE

  If the player picks up a blue potion on level 2, uses it, and then picks up another one on level 4, the game will end up creating two different BluePotion instances.

- The `RedPotion` class is very similar to `BluePotion`, except that its `Name` property returns the string `Red Potion`, and its `Attack()` method increases the player's health by up to 10 hit points.

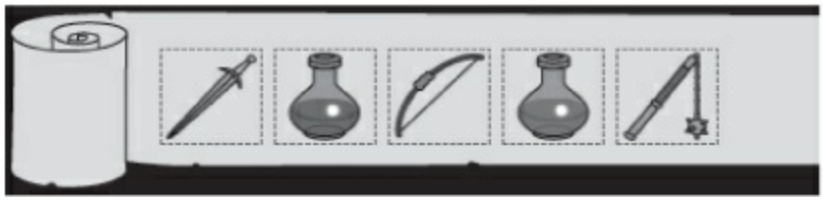
The form brings it all together

There's one instance of the `Game` object, and it lives as a private field in your form object. It's created in the form's `Load` event, and the various event handlers in the form use the fields and methods on the `Game` object to keep the game play going.

Everything begins with the form's `Load` event handler, which passes the `Game` a `Rectangle` that defines the boundaries of the dungeon play area. Here's some form code to get you going:

```
private Game game;
private Random random = new Random();
private void Form1_Load(object sender,
                        EventArgs e) {
    game = new Game(new Rectangle(78, 57, 420, 155));
    game.NewLevel(random);
    UpdateCharacters();
}
```

These are the boundaries of the dungeon in the background image you'll download and add to the form. You may need to **experiment** to find the values that fit your screen size.



Remember to double-click on **each** PictureBox so the IDE adds a separate event handler method for each of them.

USING A RECTANGLE

You'll find a lot of Rectangles any time you work with WinForms. You can create one by passing it X, Y, Width, and Height values, or two Points (for opposite corners). Once you've got a Rectangle value, you can also access its Left, Right, Top, and Bottom, as well as its X, Y, Width, and Height values.

The form has a separate event handler for each of these PictureBox's `Click` events. When the player clicks on the sword, it first checks to make sure the sword is in the player's inventory using the `Game` object's `CheckPlayerInventory()` method. If the player's holding the sword, the form calls `game.Equip()` to equip it. It then sets each PictureBox's `BorderStyle` property to draw a box around the sword, and make sure none of the other icons has a box around it.

	There's an event handler for each of the four movement buttons. They're pretty simple. First, the button calls `game.Move()` with the appropriate `Direction` value, and then it calls the form's `UpdateCharacters()` method.
	The four attack button event handlers are also really simple. Each button calls `game.Attack()`, and then calls the form's `UpdateCharacters()` method. If the player equips a potion, it's still used the same way — by calling `game.Attack()` — but potions have no direction. So make the Left, Right, and Down buttons invisible when the player equips a potion, and change the text on the Up button to say Drink.

NOTE

Make sure you change the buttons back when the player equips the sword, bow, or mace.

The form's UpdateCharacters() method moves the PictureBoxes into position

The last piece of the puzzle is the form's `UpdateCharacters()` method. Once all the objects have moved and acted on each other, the form updates everything...so weapons that been dropped have their PictureBoxes' `Visible` properties set to `false`, enemies and players are drawn in their new locations (and dead ones are made invisible), and inventory is updated.

Here's what you need to do:

- **❶ UPDATE THE PLAYER'S POSITION AND STATS.**
  The first thing you'll do is update the player's PictureBox location and the label that shows his hit points. Then you'll need a few variables to determine whether you've shown each of the various enemies.

```
public void UpdateCharacters() {
  Player.Location = game.PlayerLocation;
  playerHitPoints.Text =
    game.PlayerHitPoints.ToString();

  bool showBat = false;
  bool showGhost = false;
  bool showGhoul = false;
  int enemiesShown = 0;
  // more code to go here...
```

The showBat variable will be set to true if we made the bat's PictureBox visible. Same goes for showGhost and showGhoul.

- **❷ UPDATE EACH ENEMY'S LOCATION AND HIT POINTS.**
  Each enemy could be in a new location and have a different set of hit points. You need to update each enemy after you've updated the player's location:

```
foreach (Enemy enemy in game.Enemies) {
  if (enemy is Bat) {
    bat.Location = enemy.Location;
    batHitPoints.Text = enemy.HitPoints.ToString();
    if (enemy.HitPoints > 0) {
      showBat = true;
      enemiesShown++;
    }
  }
  // etc...
```

This goes right after the code from above.

This will affect the visibility of the enemy PictureBox controls in just a bit.

You'll need two more if statements like this in your foreach loop—one for the ghost and one for the ghoul.

  Once you've looped through all the enemies on the level, check the `showBat` variable. If the bat was killed, then `showBat` will still be `false`, so make its PictureBox invisible and clear its hit points label. Then do the same for `showGhost` and `showGhoul`.

- **❸ UPDATE THE WEAPON PICTUREBOXES.**
  Declare a `weaponControl` variable and use a big `switch` statement to set it equal to the PictureBox that corresponds to the weapon in the room.

```
sword.Visible = false;
bow.Visible = false;
redPotion.Visible = false;
bluePotion.Visible = false;
mace.Visible = false;
Control weaponControl = null;
switch (game.WeaponInRoom.Name) {
   case "Sword":
      weaponControl = sword; break;
```

Make sure your controls' names match these names. It's easy to end up with bugs that are difficult to track down if they don't match.

You'll have more cases for each weapon type.

The rest of the cases should set the variable `weaponControl` to the correct control on the form. After the `switch`, set `weaponControl.Visible` to `true` to display it.

- **❹ SET THE VISIBLE PROPERTY ON EACH INVENTORY ICON PICTUREBOX.**
  Check the `Game` object's `CheckPlayerInventory()` method to figure out whether or not to display the various inventory icons.
- **❺ HERE'S THE REST OF THE METHOD.**
  The rest of the method does three things. First, it checks to see if the player's already picked up the weapon in the room, so it knows whether or not to display it. Then it checks to see if the player died. And finally, it checks to see if the player's defeated all of the enemies. If he has, then the player advances to the next level.

```
weaponControl.Location = game.WeaponInRoom.Location;
if (game.WeaponInRoom.PickedUp)
  weaponControl.Visible = false;
else
  weaponControl.Visible = true;

if (game.PlayerHitPoints <= 0) {
  MessageBox.Show("You died");
  Application.Exit();
}

if (enemiesShown < 1) {
  MessageBox.Show("You have defeated the enemies on this level");
  game.NewLevel(random);
  UpdateCharacters();
}
```

Every level has one weapon. If it's been picked up, we need to make its icon invisible.

Application.Exit() immediately quits the program. It's part of System.Windows.Forms, so you'll need the appropriate using statement if you want to use it outside of a form.

If there are no more enemies on the level, then the player's defeated them all and it's time to go to the next level.

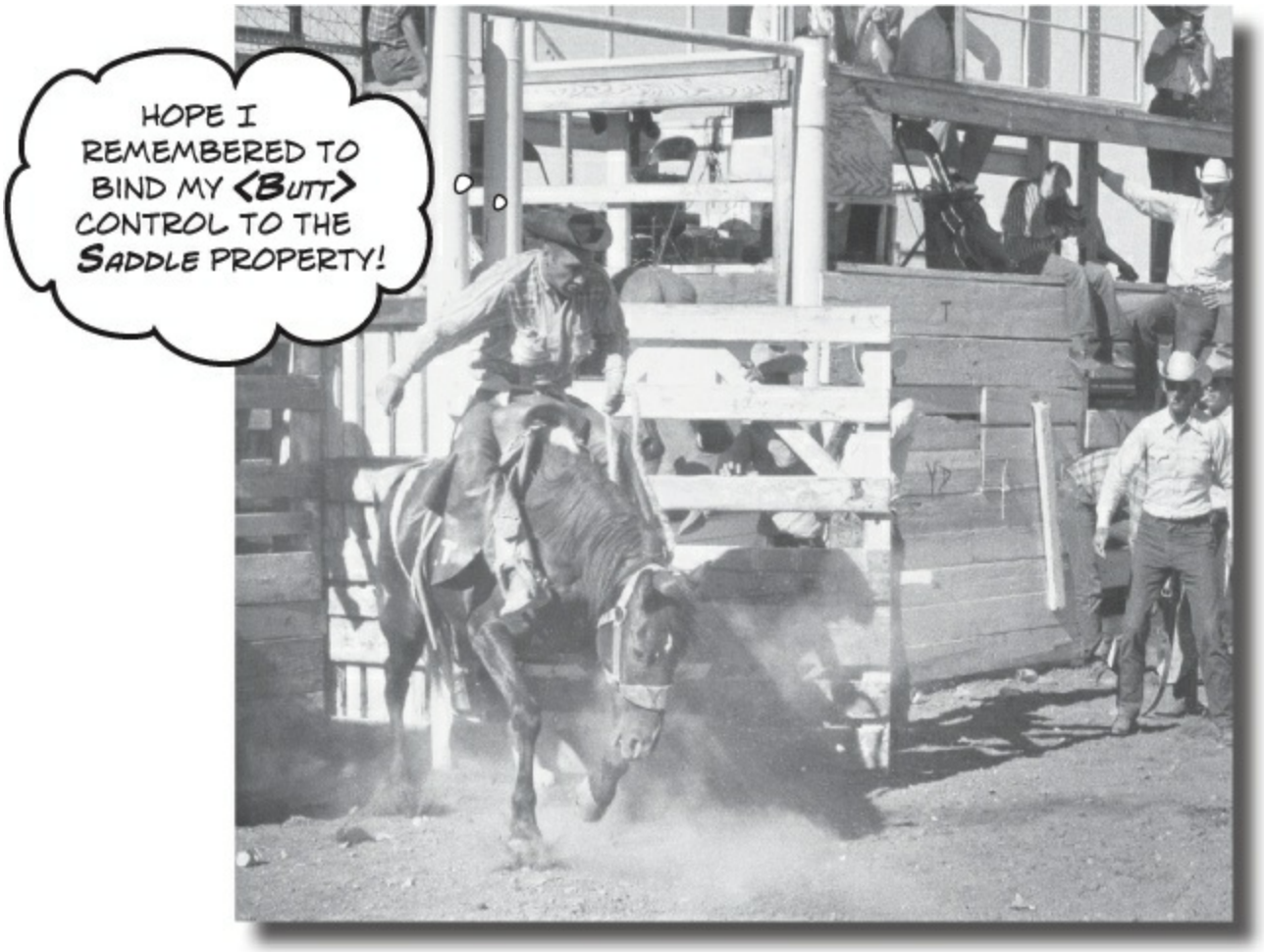
The fun's just beginning!

Seven levels, three enemies...that's a pretty decent game. But you can make it even better. Here are a few ideas to get you started....

- **Make the enemies smarter.**
  Can you figure out how to change the enemies' **Move()** methods so that they're harder to defeat? Then see if you can change their constants to properties, and add a way to change them in the game.
- **Add more levels.**
  The game doesn't have to end after seven levels. See if you can add more...can you figure out how to make the game go on indefinitely? If the player does win, make a cool ending animation with dancing ghosts and bats! And the game ends pretty abruptly if the player dies. Can you think of a more user-friendly ending? Maybe you can let the user restart the game or retry his last level.
- **Add different kinds of enemies.**
  You don't need to limit the dangers to ghouls, ghosts, and bats. See if you can add more enemies.
- **Add more weapons.**
  The player will definitely need more help defeating any new enemies you've added. Think of new ways that the weapons can attack, or different things that potions can do. Take advantage of the fact that **Weapon** is a subclass of **Mover** — make magic weapons the player has to chase around!
- **Add more graphics.**
  You can go to **www.headfirstlabs.com/books/hfcsharp/** to find more graphics files for additional enemies, weapons, and other images to help spark your imagination.
- **Make it an action game.**
  Here's an interesting challenge. Can you figure out how to use the **KeyDown** event and **Timer** you used in the Key Game in Chapter 4 to change this from a turn-based game into an action game?

This is your chance to show off! Did you come up with a cool new version of the game? Post it to CodePlex or another project hosting site. Then join the Head First C# forum and post about it to claim your bragging rights: http://www.headfirstlabs.com/books/hfcsharp/

Chapter 10. Designing Windows Store Apps with XAML: Taking your apps to the next level



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

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

Brian's running Windows 8

Too bad Brian's old-fashioned, old-style desktop app looks so outdated! He's sick of poking at tiny checkboxes in the desktop. Brian wants his excuse generator to be a true Windows app. Can we give him one?

BEHIND THE SCENES

Do this!

Want to get these ideas into your brain fast? Then do these things before starting this chapter!

Windows Store apps are more complex than WinForms programs. That's why WinForms are a really effective teaching tool, but not nearly as effective for building killer apps. It's useful to take a step back and think about how you're learning, because that will help you learn more effectively. Let's do that now.

You've laid down a good foundation of core knowledge about C#, objects, collections, and other .NET tools. Now we'll get back to building Windows Store apps with XAML. Over the next few pages, we're going to use the IDE to explore the objects that WinForms programs create and manage. Try to stay aware of what's different — and, just as importantly, what's the same — when exploring those objects, and then when you use the IDE to explore the objects that make up a Windows Store app built with XAML.

Before you continue on with this chapter, do these things:

We spent Chapter 1 and much of Chapter 2 introducing you to building Windows Store apps with XAML and the .NET Framework for Windows Store. We're going to build on knowledge from those chapters. If you want the best learning experience, we recommend that you do a few things:

- Switching gears from WinForms back to XAML is totally fine for some people, but for others it's really jarring. **This will help you get these ideas into your brain faster!**
- **Go back to Chapter 1** and build the *Save the Humans* game again from scratch. This time, make sure you type in all of the code.
- And actually ***read the code***! There are still some things in it that we haven't covered yet, but you should recognize enough of it to start to lay down a good mental foundation.
- Try to piece together how the game works. Don't give yourself a hard time, though. Like we said, there's still a lot we haven't covered yet.
- Pay special attention to what you do with the Basic Page template and how you swap it in for the default *MainPage.xaml*, because you'll be doing that several times in this chapter.
- **Redo** the XAML projects in Chapter 2, too. Even the exercise. Now you're ready!

NOTE

One more thing...

You haven't covered 100% of what WinForms apps can do. In fact, WinForms has a graphics engine called GDI+ that's capable of surprisingly good graphics, printing, and user interaction (but it's lot more work than doing the same thing in XAML). One of the most important ways to learn programming principles is to see the same thing done in more than one way.

SCAVENGER HUNT!

You've learned a whole bunch of important C# concepts since you built *Save the Humans* in Chapter 1, and you've gotten plenty of practice using them. Now it's time to put on your detective hat and test your sleuthing skills. See if you can find all of these C# items in the *Save the Humans* code. We got you started with one of the answers. Can you find the rest?

(Some of these items have more than one correct answer.)

☐ **Use a string as a key to look up an object in a `Dictionary`.**

_____

_____

_____

☐ **Use an object initializer.**

_____

_____

_____

☐ **Add two different types of object to the same collection.**

_____

_____

_____

☐ **Call a static method.**

_____

_____

_____

☐ **Use an event handler method.**

_____

_____

_____

☐ **Use the `as` keyword to downcast an object.**

_____

_____

_____

☐ **Pass a reference to an object into a method.**

_____

_____

_____

☑ **Instantiate an object and one of its methods.**

_A new StoryBoard object is instantiated in the_____

_AnimateEnemy() method, and its Begin() method is___

_called._____

☐ **Use an enum to assign a value.**

_____

_____

_____

SCAVENGER HUNT! SOLUTION

You've learned a whole bunch of important C# concepts since you built *Save the Humans* in Chapter 1, and you've gotten plenty of practice using them. Now it's time to put on your detective hat and test your sleuthing skills. See if you can find all of these C# items in the *Save the Humans* code. We got you started with one of the answers. Can you find the rest?

(Some of these items have more than one correct answer.)

> **NOTE**
>
> These are the ones we found; you may have found others!

☑ **Use a string as a key to look up an object in a `Dictionary`.**

In the second line of the AddEnemy() method,____
the string "EnemyTemplate" is used to look up a___
ControlTemplate object in the Resources dictionary.

☑ **Use an object initializer.**

The From, To, and Duration properties of the_____
DoubleAnimation object are initialized with an object
initializer in the AnimateEnemy() method._____

☑ **Add two different types of object to the same collection.**

A StackPanel object (human) and a Rectangle object
(target) are added to the playArea.Children collection
in the StartGame() method._____

☑ **Call a static method.**

The static SetLeft() and SetTop() methods on the___
Canvas class are called in the target_PointerEntered()
event handler method._____

☑ **Use an event handler method.**

The Properties window is used to create an event
handler method for the PointerPressed event of the
"human" StackPanel._____

☑ **Use the `as` keyword to downcast an object.**

The Resources dictionary has the type <object, object>, so the return value of Resources["EnemyTemplate"] is downcast to the specific type ControlTemplate._____

☑ **Pass a reference to an object into a method.**

A reference to a ContentControl object is passed as the first parameter to the AnimateEnemy() method.__

_____

☑ **Instantiate an object and one of its methods.**

A new StoryBoard object is instantiated in the AnimateEnemy() method, and its Begin() method is called._____

☑ **Use an enum to assign a value.**

The Visibility enum is used in the EndTheGame() method to set startButton.Visibility to the value Visibility.Collapsed._____

Windows Forms use an object graph set up by the IDE

When you create a Windows Desktop program, the IDE sets up your form and generates a bunch of code in the *Form1.Designer.cs* file. But what's actually in that file? There's no mystery there. You already know that each control that's on your form is an object, and you know that you can store references to objects in fields. So somewhere in that generated code is a field declaration for each object, code to instantiate it, and code to display it on the form. Let's go find those lines so we can see exactly what's going on with your forms.

- ❶ Open up the Simple Text Editor project you built in Chapter 9 and open *Form1.Designer.cs*. Scroll to the bottom and find the field declarations. You should see one declaration for each control:

```
Windows Form Designer generated code

private System.Windows.Forms.OpenFileDialog openFileDialog1;
private System.Windows.Forms.SaveFileDialog saveFileDialog1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.Button open;
private System.Windows.Forms.Button save;
private System.Windows.Forms.TableLayoutPanel tableLayoutPanel1;
private System.Windows.Forms.FlowLayoutPanel flowLayoutPanel1;
```

- ❷ Expand the Windows Form Designer–generated code section and find the code that instantiates the controls:

```
private void InitializeComponent()
{
    this.openFileDialog1 = new System.Windows.Forms.OpenFileDialog();
    this.saveFileDialog1 = new System.Windows.Forms.SaveFileDialog();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.open = new System.Windows.Forms.Button();
    this.save = new System.Windows.Forms.Button();
    this.tableLayoutPanel1 = new System.Windows.Forms.TableLayoutPanel();
    this.flowLayoutPanel1 = new System.Windows.Forms.FlowLayoutPanel();
```

The IDE may have generated these lines in a different order, but there should be one line for each control on the form.

- ❸ Controls like the TableLayoutPanel and FlowLayoutPanel that contain other controls have a public property called **`Controls`**. It's a `ControlCollection` object, which in a lot of ways is very similar to a `List<Control>` object. Every control on your form is a subclass of `Control`, and adding it to a panel's `Controls` collection causes it to draw itself inside that panel. Scroll down and find where the Open and Save buttons are added to the FlowLayoutPanel:

```
this.flowLayoutPanel1.Controls.Add(this.save);
this.flowLayoutPanel1.Controls.Add(this.open);
```

The FlowLayoutPanel is in a cell in the TableLayoutPanel. Find where it and the TextBox are added:

```
this.tableLayoutPanel1.Controls.Add(this.textBox1, 0, 0);
this.tableLayoutPanel1.Controls.Add(this.flowLayoutPanel1, 0, 1)
```

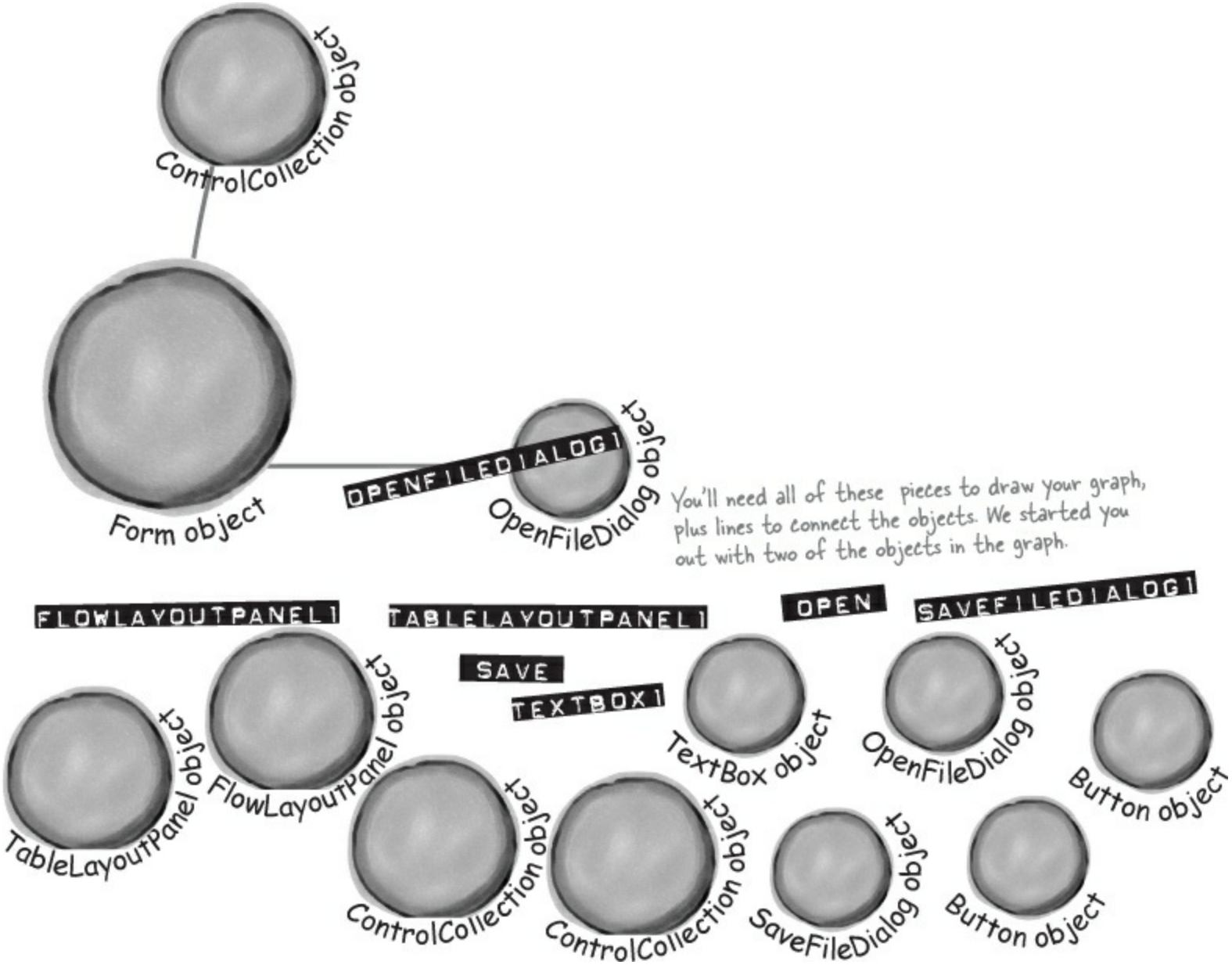
Your `Form` object is also a container, and it contains the TableLayoutPanel:

```
this.Controls.Add(this.tableLayoutPanel1);
```

SHARPEN YOUR PENCIL

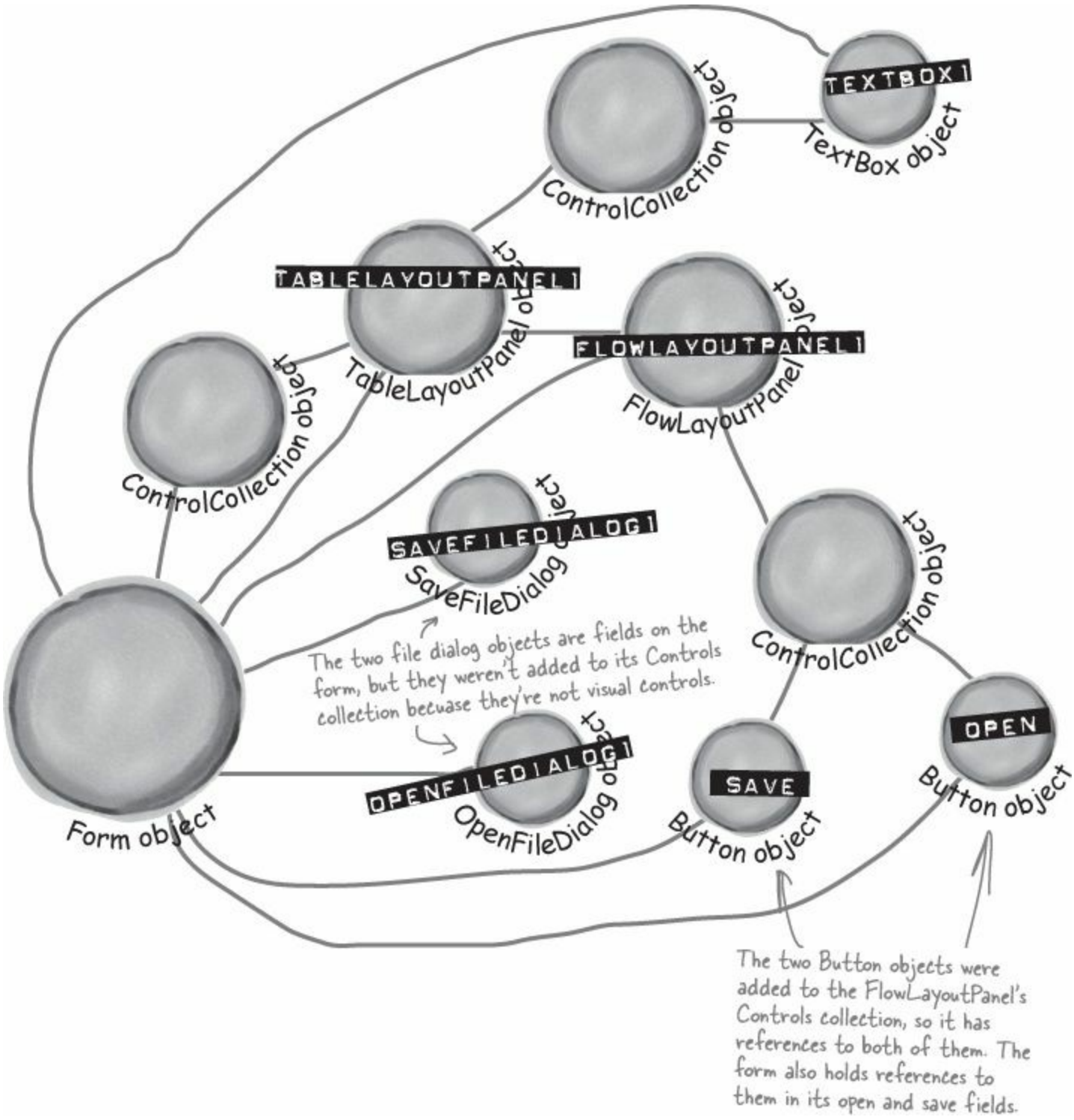
Look at the `new` and `Controls.Add()` statements that the IDE generated for the Simple Text Editor form and draw the object graph that they create when they're executed.

> **NOTE**
>
> You'll need to open up the Form1.Designer.cs file in the code that you built for the Simple Text Editor in Chapter 9.



SHARPEN YOUR PENCIL SOLUTION

Look at the `new` and `Controls.Add()` statements that the IDE generated for the Simple Text Editor form and draw the object graph that they create when they're executed.



NOTE

This is going to come in really handy when you're doing exercises. Have a look at the other methods in that Debug class, too.

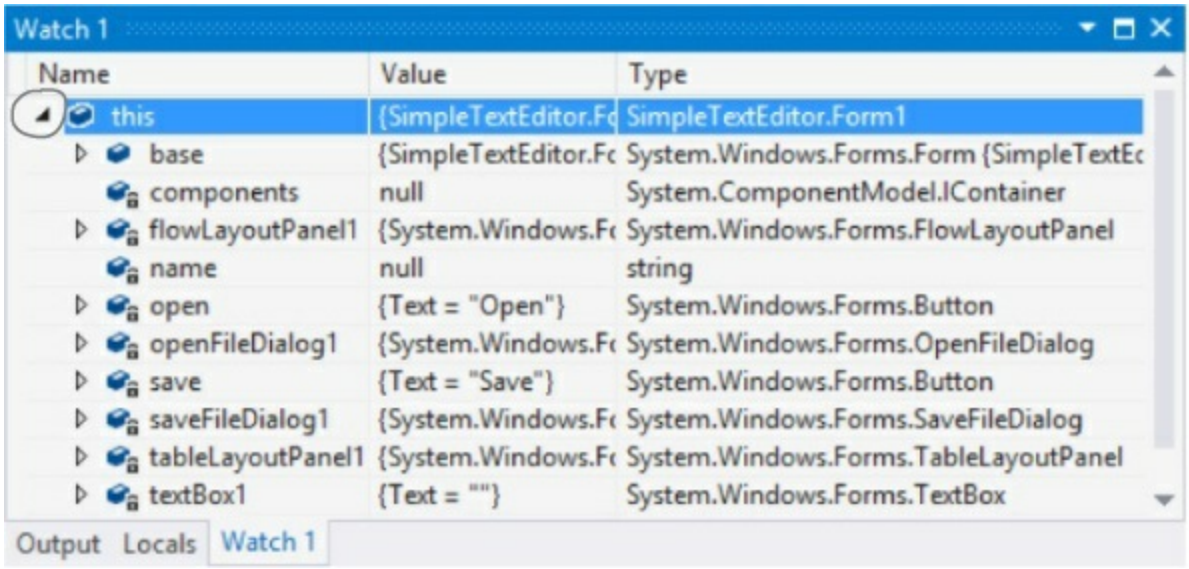
Debugging tip

`System.Diagnostics.Debug.WriteLine()` will write text to the output window while you're debugging. You can use it just like you used `Console.WriteLine()` in Windows Forms apps.

Use the IDE to explore the object graph

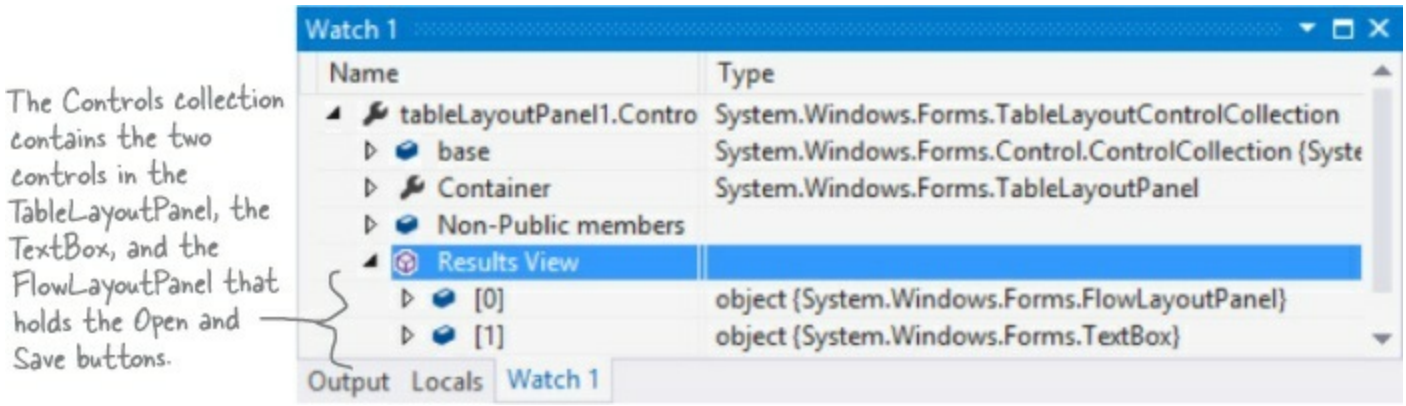
DO THIS!

Go back to your Simple Text Editor project and put a breakpoint on the call to `InitializeComponent()` in the form's constructor, then start debugging the program. When it hits the breakpoint, press F10 to step over the method. Then **go to the Watch window and enter** `this` to explore the object graph.



Click the triangle next to "this" to expand it to see all of the fields that the IDE generated to hold references to the form's controls.

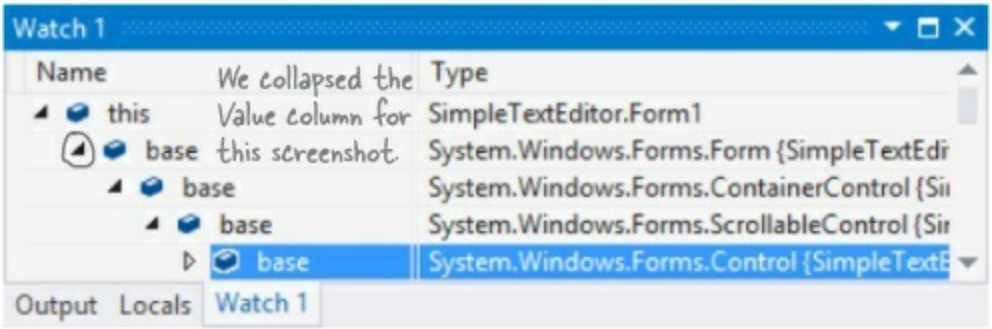
Add a watch for `tableLayoutPanel1.Controls` and expand Results View to see the objects it contains:



The Controls collection contains the two controls in the TableLayoutPanel, the TextBox, and the FlowLayoutPanel that holds the Open and Save buttons.

The `System.Windows.Form` class doesn't actually have a `Controls` property. It inherits that property from its superclass, `ContainerControl`, which inherits it from `ScrollableControl`, which inherits it from `Control`. Keep expanding base in the Watch window to move up the inheritance hierarchy to `System.Windows.Forms.Control`. (That's where `Form` inherits its Controls collection.) Expand at the `Controls` collection's **Results** View. You'll see an object for each control on the form!

Expand "base" to see the properties an object inherits from its base class:



NOTE

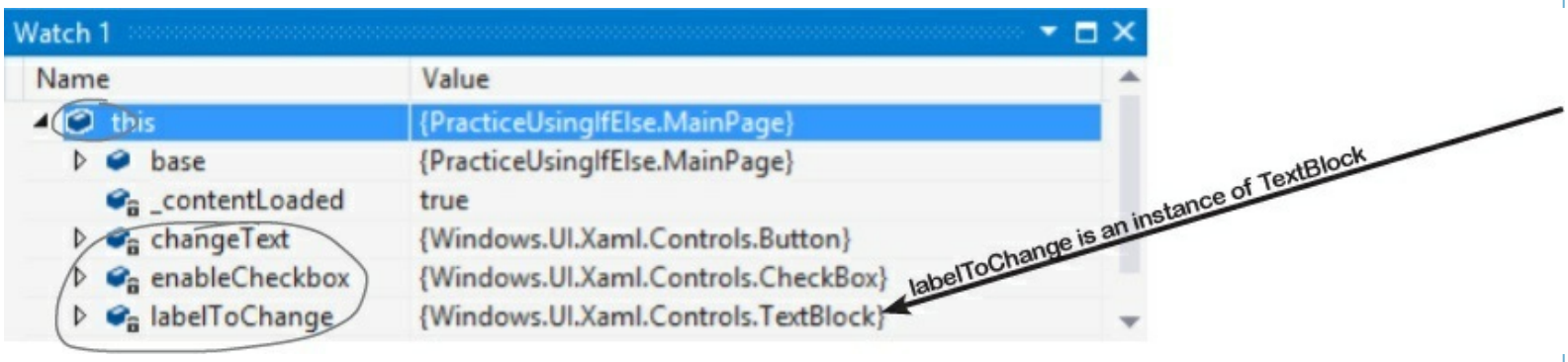
That's your last look at WinForms apps for a few chapters! We'll come back to them a couple more times, though, because they do make really good tools for learning and exploring C#.

Windows Store apps use XAML to create UI objects

DO THIS!

When you use XAML to build the user interface for a Windows Store app, you're building out an object graph. And just like with WinForms, you can explore it with IDE's Watch window. **Open the program from the Chapter 2 "practice using if-else statements" exercise**. Then open *MainPage.xaml.cs*, place a breakpoint on the first lne of `changeText_Click()`, and **use the IDE to explore the app's UI objects**.

- ❶ Start debugging, then press the button so the program hits the breakpoint. Visual Studio 2013 for Windows has a slightly different window layout than VS2013 for Desktop because it has more features, including multiple watch windows (which comes in handy when you need to watch many things). Choose **Debug→Windows→Watch→Watch 1** to display one of the watch windows and watch `this`:



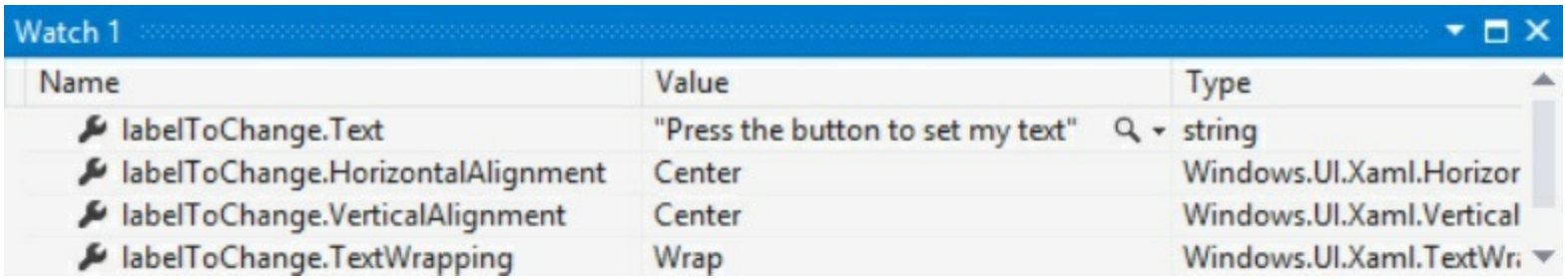
- ❷ Now have another look at the XAML that defines the page:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Button x:Name="changeText" Content="Change the label if checked"
            HorizontalAlignment="Center" Click="changeText_Click"/>

    <CheckBox x:Name="enableCheckbox" Content="Enable label changing"
              HorizontalAlignment="Center" IsChecked="true"
              Grid.Column="1"/>

    <TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
               Text="Press the button to set my text"
               HorizontalAlignment="Center" VerticalAlignment="Center"
               Grid.ColumnSpan="2"/>
</Grid>
```

If you get deployment errors when you open your program from Chapter 2, try choosing "Clean" from the Build menu.

The XAML that defines the controls on a page is turned into a Page object with fields and properties that contain references to UI controls.

- ❸ Add some of the `labelToChange` properties to the Watch window:

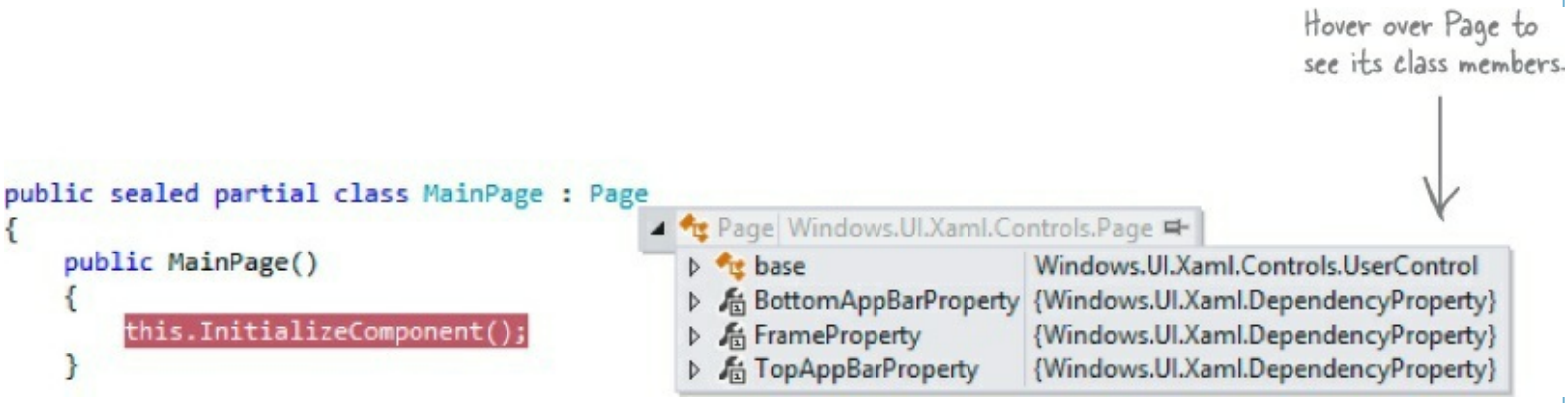


The app automatically sets the properties based on your XAML:

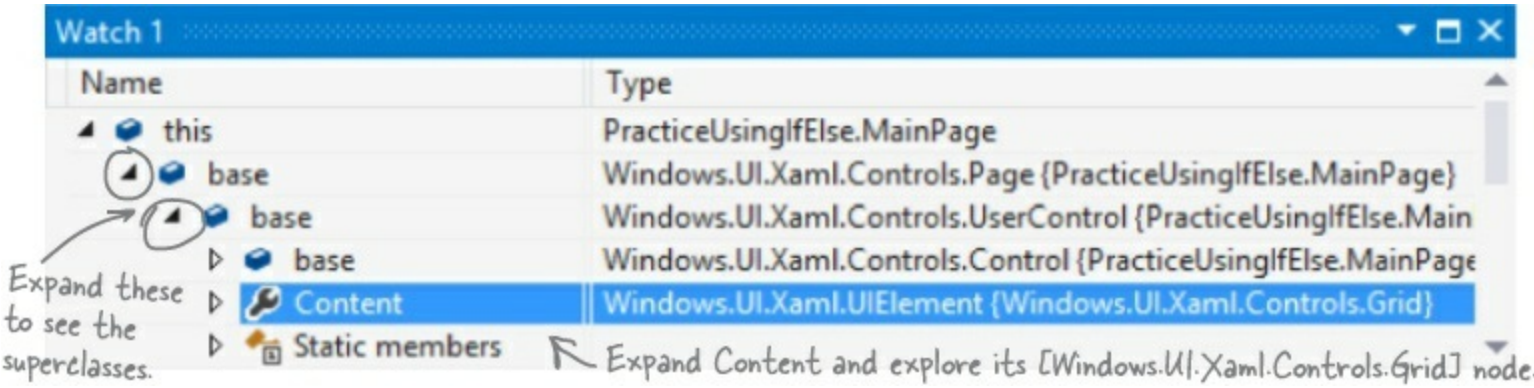
```
<TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
           Text="Press the button to set my text" ←
           HorizontalAlignment="Center" VerticalAlignment="Center"
           Grid.ColumnSpan="2"/>
```

But try putting `labelToChange.Grid` or `labelToChange.ColumnSpan` into the Watch window. The control is a `Windows.UI.Controls.TextBlock` object, and that object doesn't have those properties. Can you guess what's going on with those XAML properties?

- ❹ Place a breakpoint in the constructor. Then open *MainPage.xaml.cs*, and find the class declaration for `MainPage`. Take a look at the declaration — it's a subclass of `Page`. Hover over `Page` so the IDE:



Now start your program again and press F10 to step over the call to `InitializeComponent()`. Go back to the Watch window and expand `this` >> `base` >> `base` to traverse back up the inheritance hierarchy.



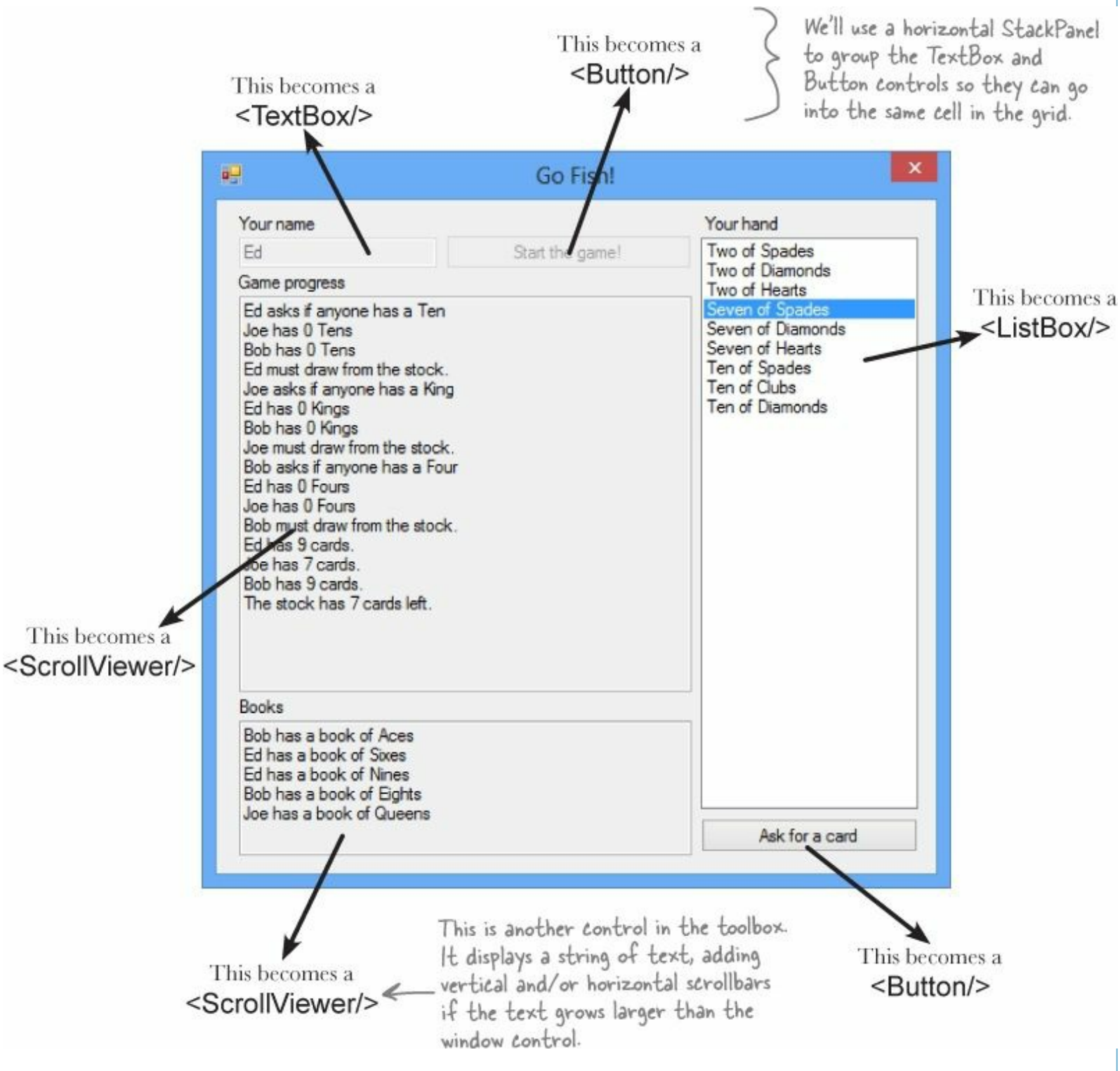
Take a little time and explore the objects that your XAML generated. We'll dig into some of these objects later on in the book. For now, just poke around and get a sense of how many objects are behind your app.

If you used the WPF Learner's Guide in Appendix B to build desktop versions of the projects in Chapter 1 and Chapter 2, flip to the appendix now to find the replacement pages for Chapter 10 and beyond.

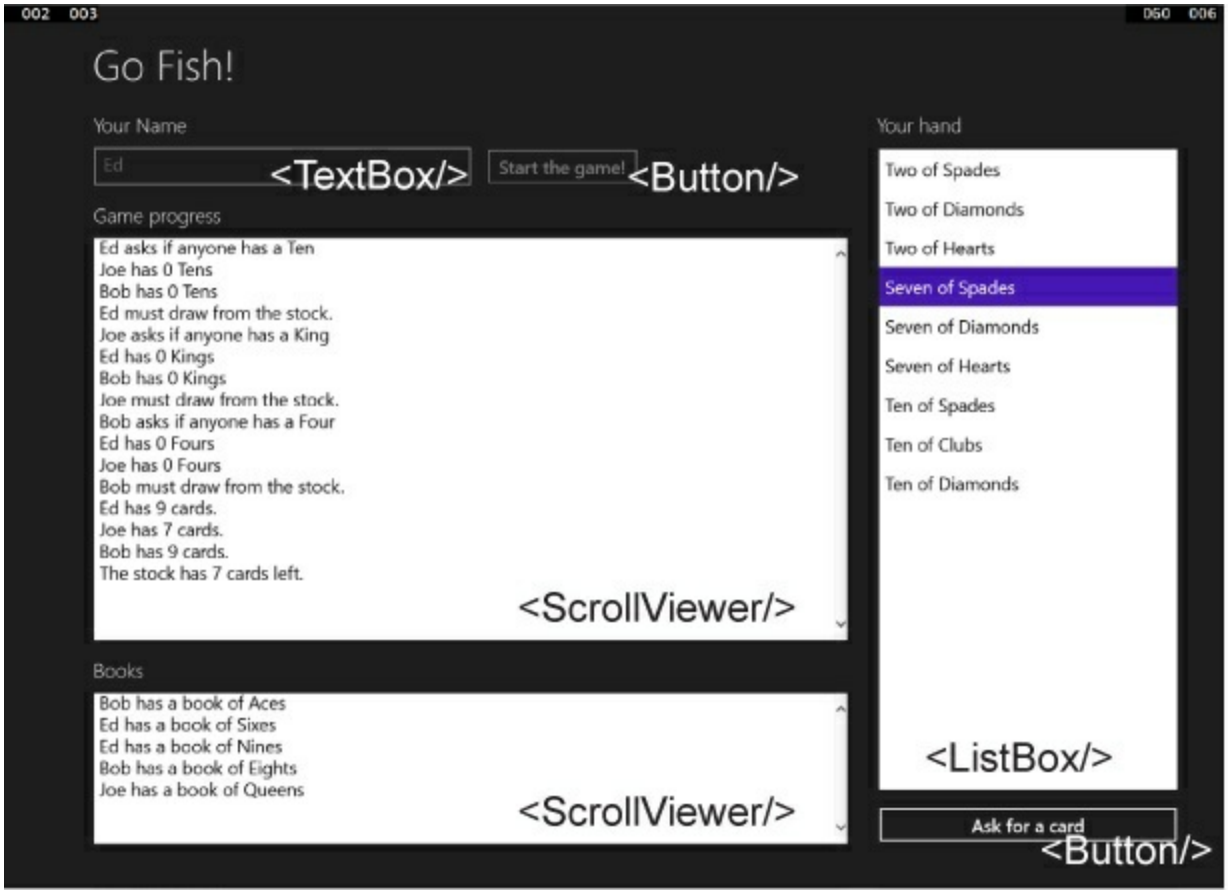
Redesign the Go Fish! form as a Windows Store app page

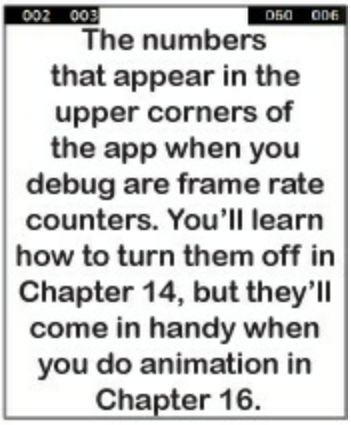
The Go Fish! game that you built in Chapter 8 would make a great Windows Store app. Open Visual Studio 2013 for Windows and **create a new Windows Store project**, then delete *MainPage.xaml* and replace it with a Basic Page (just like you did when you built *Save the Humans*). Over the next few pages, you'll redesign your Go Fish! game in XAML as a page that adjusts to different sized devices. And instead of using Windows Desktop controls on a form, you'll use Windows Store app controls on a page.

DO THIS!



Here's how those controls will look on the app's main page:

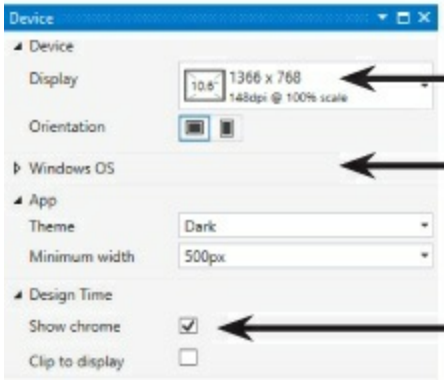




Most of the code to manage the gameplay will remain the same, but the UI code will change.

The controls will be contained in a grid, with rows and columns that expand or contract based on the size of the display. This will allow the game to shrink or grow to fit the screen. You can use the **Device window** in the IDE to test different screen configurations.

The game will be playable no matter what the page dimensions are.



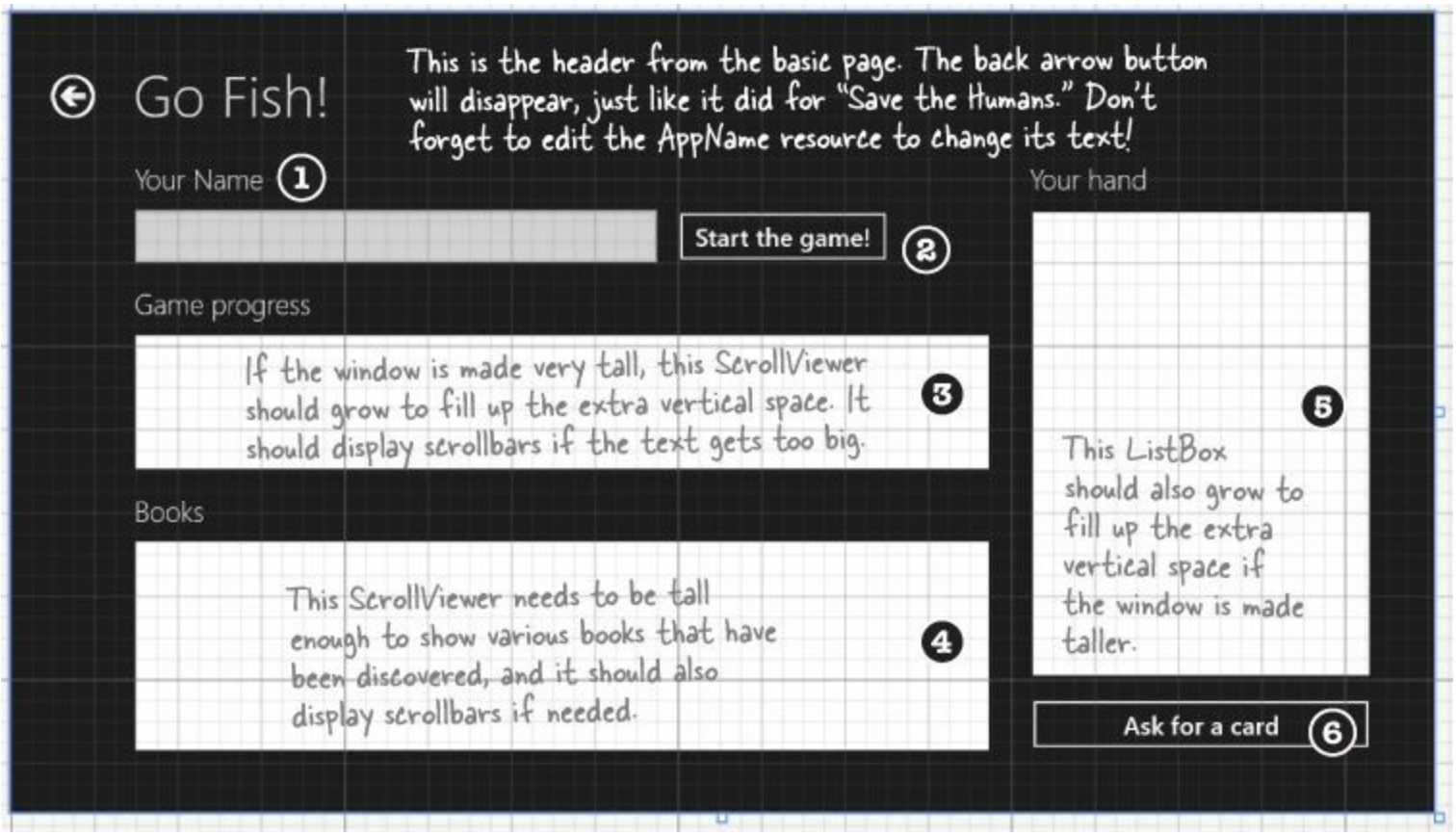
The Device controls let you see your page in portrait, landscape, and different sizes.

The Windows OS options let you set the edge and contrast options. You can use the App options to set the theme and minimum width.

Uncheck the "Show chrome" box to turn off the picture of the Surface bezel.

Page layout starts with controls

XAML and WinForms have one thing in common: they both rely on controls to lay out your page. The Go Fish! page has two buttons, a ListBox to show the hand, a TextBox for the user to enter the name, and four TextBlock labels. It also has two ScrollViewer controls with a white background to display the game progress and books.



The Basic Page template includes a grid that has two rows. The top row contains the header with the app name. The second row holds the contents, which are defined by this grid. This whole grid will be contained in row 1 of the Basic Page (it only has one column, column #0). Drag a Grid out of the Toolbox and **drop it in the bottom part of the page** to put it in the second row. Here's the XAML—you can find `<Grid>` that you just added to the page in the XAML window and modify it by hand, or you can use the IDE to lay out the page:

```
<Grid Grid.Row="1" Margin="120,0,60,60">
    <TextBlock Text="Your Name" Margin="0,0,0,20"
        Style="{StaticResource SubheaderTextBlockStyle}"/>
```

The margins indent the grid so it's aligned with the page. The left margin is always 120 pixels.

Here's the opening tag for the grid.

We'll use a StackPanel to put the TextBox for the player's name and the Start button in one cell:

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
    <TextBox x:Name="playerName" FontSize="24"
        Width="500" MinWidth="300" />

    <Button x:Name="startButton" Margin="20,0"
    Content="Start the game!"/>
</StackPanel>
```

This adds a 20-pixel space between the TextBox and the Button. When the Margin property only has two numbers, they specify horizontal (left/right) and vertical (top/bottom) margins.

-

Each label on the page ("Your name," "Game progress," etc.) is a TextBlock with a small margin above it and `SubHeaderTextBlockStyle` applied:

```
<TextBlock Text="Game progress"
    Style="{StaticResource SubheaderTextBlockStyle}"
    Margin="0,20,0,20" Grid.Row="2"/>
```

A ScrollViewer control displays the game progress, with scrollbars that appear if the text is too big for the window:

-
```
<ScrollViewer Grid.Row="3" FontSize="24"
        Background="White" Foreground="Black" />
```

Here's another TextBlock and ScrollViewer to display the books. The default vertical and horizontal alignment for the ScrollViewer is `Stretch`, and that's going to be really useful. We'll set up the rows and columns so the ScrollViewer controls expand to fit any screen size.

-
```
<TextBlock Text="Books" Style="{StaticResource SubheaderTextBlockStyle}"
          Margin="0,20,0,20" Grid.Row="4"/>

<ScrollViewer FontSize="24" Background="White" Foreground="Black"
             Grid.Row="5" Grid.RowSpan="2" />
```

We used a small 40-pixel column to add space, so the ListBox and Button controls need to go in the third column. The ListBox spans rows 2 through 6, so we gave it `Grid.Row="1"` and `Grid.RowSpan="5"` — this will also let the ListBox grow to fill the page.

Remember, rows and columns start at zero, so a control in the third column has Grid.Column="2".

```
<TextBlock Text="Your hand" Style="{StaticResource SubheaderTextBlockStyle}"
Grid.Row="0" Grid.Column="2" Margin="0,0,0,20"/>

<ListBox x:Name="cards" Background="White" FontSize="24" Height="Auto"
Margin="0,0,0,20" Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"/>
```

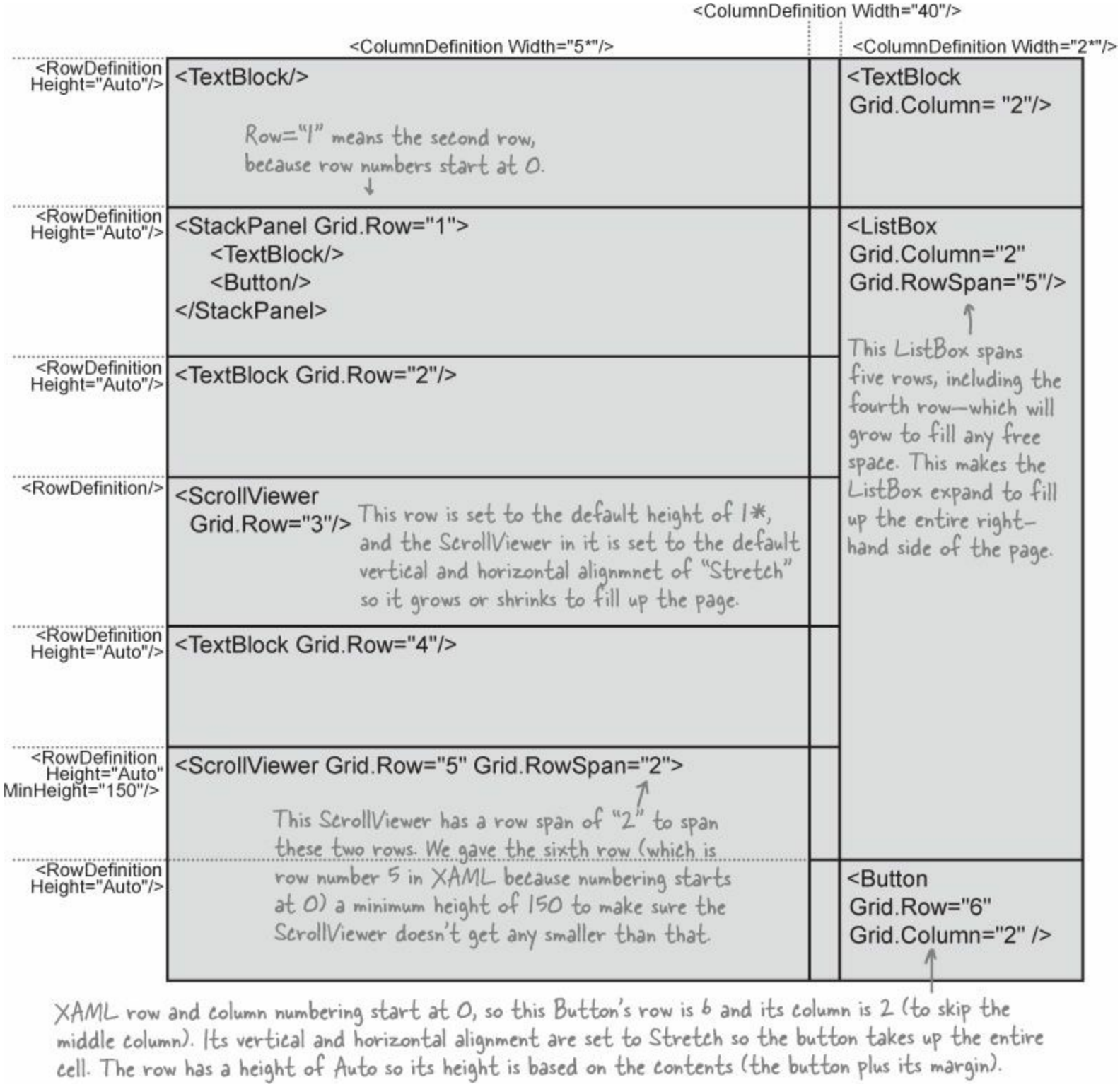
The "Ask for a card" button has its horizontal and vertical alignment set to `Stretch` so that it fills up the cell. The 20-pixel margin at the bottom of the ListBox adds a small gap.

-
```
<Button x:Name="askForACard" Content="Ask for a card"
HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
Grid.Row="6" Grid.Column="2"/>
```

We'll finish this grid on the next page →

Rows and columns can resize to match the page size

Grids are very effective tools for laying out pages because they help you design pages that can be displayed on many different devices. Heights or widths that end in * **adjust automatically** to different screen geometries. The Go Fish! page has three columns. The first and third have widths of 5* and 2*, so they will **grow or shrink proportionally** and always keep a 5:2 ratio. The second column has a fixed width of 40 pixels to keep them separated. Here's how the rows and columns for the page are laid out (including the controls that live inside them):



Here's how the row and column definitions make the page layout work:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="5*"/>
    <ColumnDefinition Width="40"/>
    <ColumnDefinition Width="2*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto" MinHeight="150" />
    <RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
```

The first column will always be 2.5 times as wide as the third (a 5:2 ratio), with a 40-pixel column to add space between them. The ScrollViewer and ListBox controls that display data have HorizontalAlignment set to "Stretch" to fill up the columns.

The fourth row has the default height of 1* to make it grow or shrink to fill up any space that isn't taken up by the other rows. The ListBox and first ScrollViewer span this row, **so they will grow and shrink too.**

Almost all of the row heights are set to `Auto`. There's only one row that will grow or shrink, and any control that spans this row will also grow or shrink.

You can add the row and column definitions above or below the controls in the grid. We added them below this time.

```
</Grid>
```

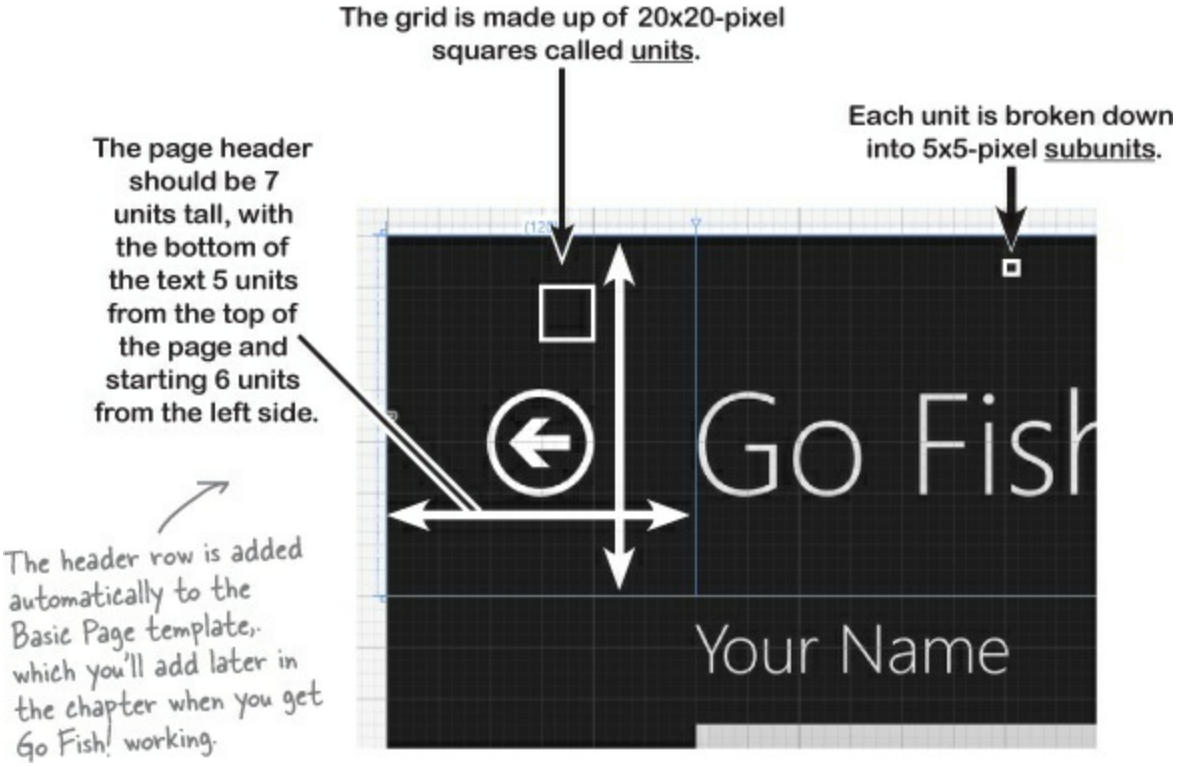
Here's the closing tag for the grid. You'll bring this all together at the end of the chapter when you finish porting the Go Fish! game to a Windows Store app.

It doesn't matter if the column and row definitions are at the top or the bottom of the grid, as long as they come before the closing `</Grid>` tag. And everything between the opening `<Grid Grid.Row=...>` tag to the closing `</Grid>` tag can appear anywhere between the opening and closing tags of the outer Grid. Since there's another Grid in the Basic Page template (it contains the back button and app name), a good place for this new Grid is right after the other inner Grid.

Use the grid system to lay out app pages

Ever notice how different Windows Store apps have a similar look? That's because they use a **grid system** to give every app what Microsoft designers call a "consistent silhouette." The grid consists of squares called *units* and *subunits* — and you've already seen them, because they're built into the IDE.

If you didn't use these buttons at the bottom of the designer to turn on the grid lines, snapping, and snapping to grid lines back in Chapter 1, use them now.



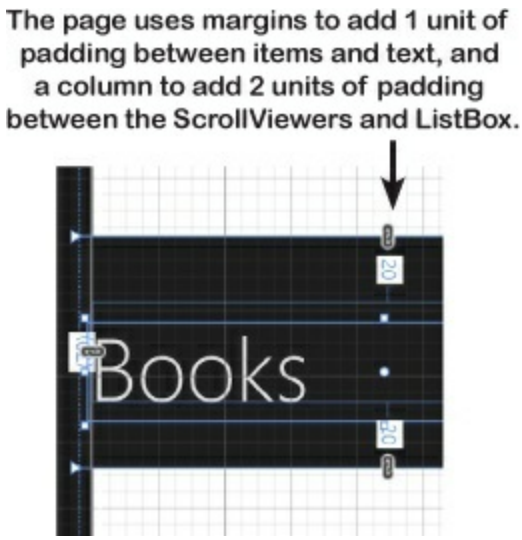
In the Go Fish! app, you use the `Margin` property in the `<Grid>` that contained all of your controls to create the spacing. The `Margin` property consists of either one number (a thickness value for left, top, right, and bottom), two numbers (where the first is left and right, and the second is top and bottom), or four comma-separated numbers specifying left, top, right, and bottom thickness.

Your Go Fish! app's main page has a left margin of 120 pixels (6 units), top margin of 0 pixels, and right and bottom margins of 60 pixels (3 units):

```
<Grid Grid.Row="1" Margin="120,0,60,60">
```

It also has a 1-unit margin above and below each label:

```
<TextBlock Text="Books"
           Style="{StaticResource SubheaderTextBlockStyle}"
           Margin="0,20,0,20" Grid.Row="4"/>
```



THERE ARE NO DUMB QUESTIONS

Q: **Q: What does setting the row height or column width to "Auto" do?**

A: **A:** When you set a row's `Height` property or a column's `Width` property to `Auto`, that causes the row or column to expand or contract so that it exactly fits its contents. You can try this out yourself to see how it works. Create a new Blank App, edit the grid in *MainPage.xaml*, and add a bunch of rows and columns with heights and widths set to `Auto`. You won't see anything in the designer because the rows and columns are empty and collapsed down to zero height. Add controls with different heights to the cells, and you'll see the rows and columns expand to fit the controls.

Q: **Q: So how is that different from setting the row height or column width to 1*, 2*, or 5*?**

A: **A:** Using the `*` for the row height or column width causes the rows or columns to expand **proportionally** to fill the entire grid. If you have three columns with widths of `3*`, `3*`, and `4*`, the `3*` columns will have a width of 30% of the total grid width minus the fixed and auto columns, and the `4*` column will have 40% of that total.
This is one reason why the default width or height of 1* makes sense. If all of the rows or columns have that default setting, then they will evenly distribute themselves as the grid grows or shrinks.

Q: **Q: "Pixels." You keep using that word. I do not think it means what you think it means.**

A: **A:** Many XAML developers use the term *pixel*, but you're right — technically you aren't using the same kind of pixels you see on your screen. The technical term for the numbers in the `Margin`, `Height`, `Width`, and other properties is **device-independent unit**. Windows Store apps need to work on many different screen sizes and shapes, so no matter how big or small a device your app is displayed on, each device-independent unit will always be a 1/96th of an inch. A five-by-five square of these device-independent units combine to make a page layout subunit, and a four-by-four square of subunits make a page layout unit. It's a little confusing talking about units for page layout versus device-independent unit, so we'll keep using the word *pixel* to mean device-independent unit.
You can specify any XAML height or width in different units by adding `in` (inches), `cm` (centimeters), or `pt` (points, where a typographer's point is 1/72nd of an inch). Try laying out a page using inches or centimeters. Then take a physical ruler and hold it up to your screen to prove to yourself that Windows resizes your app properly.

Q: **Q: Is there an easy way to make sure that my app looks good on many different monitors?**

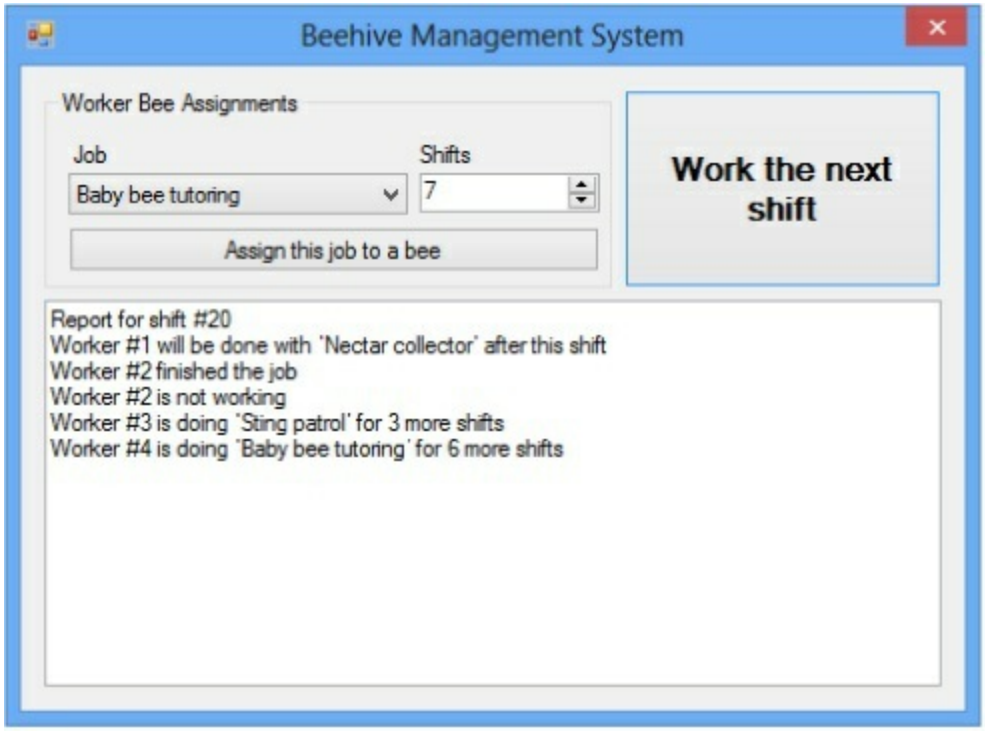
A: **A:** Yes, the IDE gives you useful tools for doing that. The IDE's XAML page designer gives you a few different ways to see how your page will look on different devices. You can use the Device window to show your page in different resolutions or split modes. And later on, we'll show you how to run your app in a simulator that lets you interact with your app in simulated devices that have various sizes and shapes.

When a row or column has a height or width of Auto, that tells it to grow or shrink to exactly fit its contents.

You can read more about laying out app pages in the Dev Center: http://msdn.microsoft.com/en-us/library/windows/apps/hh872191.aspx

EXERCISE

Use XAML to redesign each of these Windows Desktop forms as Windows Store apps. Create a new Windows Store Blank App project for each of them, add a new Basic Page item for each of these apps (just like you did for *Save the Humans*), and modify each page by updating the grid and adding controls. You don't need to get them working. Just create the XAML so they match the screenshots.

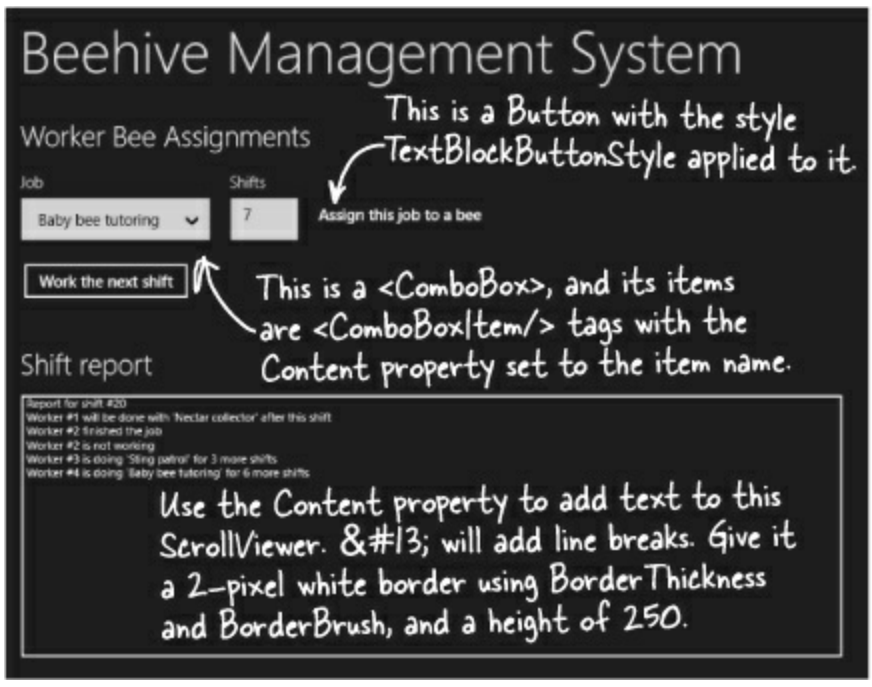


Find the right spot in each Basic Page XAML to add your new Grid or StackPanel to contain the rest of the controls for the page. You'll add them to the second row (`Grid.Row="1"`) of a newly added Blank Page.

XAML IS FLEXIBLE ABOUT TAG ORDER

We asked you to add the XAML code for your page layout below the row definitions because it's an easy location to find in your XAML file. Some developers like to keep the XAML code in the same order that it's displayed on the page. They might put it underneath the closing </Grid> tag for the grid that contains the back button and page title instead. We encourage you to experiment with this, because it's good to get a feel for what seems most intuitive for you.

Use StackPanels to design this form. It's broken into two groups. Subheaders have SubheaderTextBlockStyle applied, a 40-pixel margin between groups, and a 20-pixel margin after the group header. The labels above the items have BodyTextBlockStyle applied, and a 10-pixel margin above the item. There's a 20-pixel margin between items.




```
<StackPanel Grid.Row="1" Margin="120,0">
<TextBlock/>
<StackPanel Orientation="Horizontal">
  <StackPanel>
    <TextBlock/>
    <ComboBox>
      <ComboBoxItem/>
      <ComboBoxItem/>
      ... 4 more ...
    </ComboBox>
  </StackPanel>
  <StackPanel>
    <TextBlock/>
    <TextBox/>
  </StackPanel>
  <Button/>
</StackPanel>
<Button/>
<TextBlock/>
<ScrollViewer/>
</StackPanel>
```

Set the ComboBox control's SelectedIndex property to 0 so it displays the first item.

Use a Grid to design this form. It has eight rows with `height` set to `Auto` so they expand to fit their contents. Use StackPanels to put multiple controls in the same row.

```
<Grid Grid.Row="1" Margin="120,0">
  <TextBlock/>
  <TextBox/>
  <TextBlock/>
  <ListBox>
    <ListBoxitem/>
    <ListBoxitem/>
    ... 4 more ...
  </ListBox>
  <TextBlock>
  <StackPanel Orientation="Horizontal">
    <TextBox/>
    <ComboBox> ... 4 items ... </ComboBox>
    <Button/>
  </StackPanel>
  <ScrollViewer/>
  <StackPanel Orientation="Horizontal">
    <Button/>
    <Button/>
  </StackPanel>
```

Get your pages to look just like these screenshots by adding **dummy data** to the controls that would normally be filled in using the methods and properties in your classes.

When you use the IDE's "New Item..." option to add a Basic Page to your project, you might see an error message in the designer because it's expecting the project to be built. Rebuild the solution to make the error go away.

EXERCISE SOLUTION

Use XAML to redesign each of these Windows Desktop forms as Windows Store apps. Create a new Windows Store Blank App project for each of them, add a new Basic Page item for each of these apps (just like you did for *Save the Humans*), and modify each page by updating the grid and adding controls. You don't need to get them working. Just create the XAML so they match the screenshots.

```
<StackPanel Grid.Row="1" Margin="120,0">
    <TextBlock Text="Worker Bee Assignments"
               Style="{StaticResource SubheaderTextBlockStyle}"/>
    <StackPanel Orientation="Horizontal" Margin="0,20,0,0">
        <StackPanel Margin="0,0,20,0">
            <TextBlock Text="Job" Margin="0,0,0,10"
                       Style="{StaticResource BodyTextBlockStyle}"/>
            <ComboBox SelectedIndex="0">
                <ComboBoxItem Content="Baby bee tutoring"/>
                <ComboBoxItem Content="Egg care"/>
                <ComboBoxItem Content="Hive maintenance"/>
                <ComboBoxItem Content="Honey manufacturing"/>
                <ComboBoxItem Content="Nectar collector"/>
                <ComboBoxItem Content="Sting patrol"/>
            </ComboBox>
        </StackPanel>
        <StackPanel>
            <TextBlock Text="Shifts" Margin="0,0,0,10"
                       Style="{StaticResource BodyTextBlockStyle}"/>
            <TextBox/>
        </StackPanel>
        <Button Content="Assign this job to a bee" Margin="20,20,0,0"
                Style="{StaticResource TextBlockButtonStyle}" />
    </StackPanel>
    <Button Content="Work the next shift" Margin="0,20,0,0" />

    <TextBlock Text="Shift report" Margin="0,40,0,20"
               Style="{StaticResource SubheaderTextBlockStyle}"/>
    <ScrollViewer BorderThickness="2" BorderBrush="White" Height="250"
                  Content="
Report for shift #20&#13;
Worker #1 will be done with 'Nectar collector' after this shift&#13;
Worker #2 finished the job&#13;
Worker #2 is not working&#13;
Worker #3 is doing 'Sting patrol' for 3 more shifts&#13;
Worker #4 is doing 'Baby bee tutoring' for 6 more shifts
                        "/>
</StackPanel>
```

Here's the margin we gave you. Leaving off the right and bottom margin causes them to mirror the top and left (120 and 0).

Does your XAML code look different from ours? There are many ways to display very similar (or even identical) pages in XAML.

This is the header for the second group, with a 40 pixel margin above and 20 pixel margin below.

Here's the dummy data we used to populate the shift report. The Content property ignores line breaks—we added them to make the solution easier to read.

```
<Grid Grid.Row="1" Margin="120,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>

    <TextBlock Text="Lumberjack name" Margin="0,0,0,10"
               Style="{StaticResource BodyTextBlockStyle}"/>
    <TextBox Grid.Row="1"/>

    <TextBlock Grid.Row="2" Text="Breakfast line" Margin="0,20,0,10"
               Style="{StaticResource BodyTextBlockStyle}"/>
    <ListBox Grid.Row="3">
        <ListBoxItem Content="1. Ed"/>
        <ListBoxItem Content="2. Billy"/>
        <ListBoxItem Content="3. Jones"/>
        <ListBoxItem Content="4. Fred"/>
        <ListBoxItem Content="5. Johansen"/>
        <ListBoxItem Content="6. Bobby, Jr."/>
    </ListBox>

    <TextBlock Grid.Row="4" Text="Feed a lumberjack" Margin="0,20,0,10"
               Style="{StaticResource BodyTextBlockStyle}"/>
    <StackPanel Grid.Row="5" Orientation="Horizontal">
        <TextBox Text="2" Margin="0,0,20,0"/>
        <ComboBox SelectedIndex="0" Margin="0,0,20,0">
            <ComboBoxItem Content="Crispy"/>
            <ComboBoxItem Content="Soggy"/>
            <ComboBoxItem Content="Browned"/>
            <ComboBoxItem Content="Banana"/>
        </ComboBox>
        <Button Content="Add flapjacks" Style="{StaticResource TextBlockButtonStyle}"/>
    </StackPanel>

    <ScrollViewer Grid.Row="6" Margin="0,20,0,0" Content="Ed has 7 flapjacks"
                  BorderThickness="2" BorderBrush="White"/>

    <StackPanel Grid.Row="7" Orientation="Horizontal" Margin="0,40,0,0">
        <Button Content="Add Lumberjack" Margin="0,0,20,0" />
        <Button Content="Next Lumberjack" />
    </StackPanel>

</Grid>
```

We removed line breaks to make the row definitions fit on this page.

Just to be 100% clear, we asked you to add these dummy items **as part of the exercise**, to make the form look like it's being used. You're about to learn how to bind controls like this ListBox to properties in your classes.

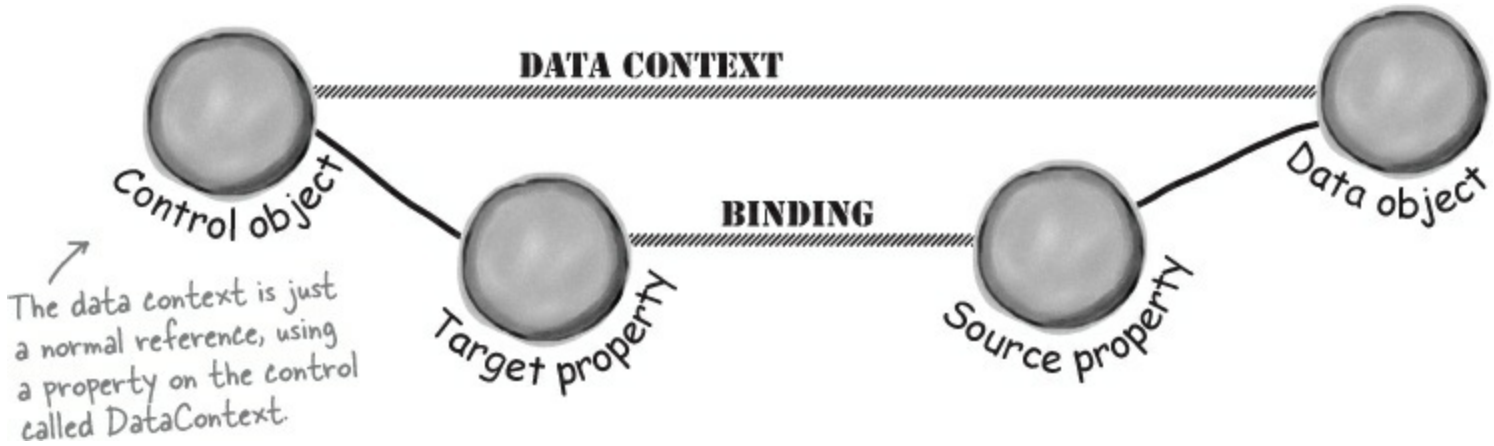
More dummy content...

BRAIN POWER

What do you think of this page layout? Would it make more sense to move the Add and Next buttons into a standard Windows 8 app bar?

Data binding connects your XAML pages to your classes

Your TextBlocks, ScrollViewers, TextBoxes, and other controls are built for displaying data. When you were using WinForms, you had to use properties to display text or add list items. That will work with XAML too, but there's another way: you can use **data binding** to automatically populate the controls on your page with data. Even better, you can also use data binding to have your controls update properties in your classes.



Data binding only works with properties. If you try to bind to a public field, you won't see anything — and you won't get an error, either!

Context, path, and binding

Data binding in XAML is a relationship between the ***source property*** of an object that feeds data to a control and the ***target property*** of the control that displays the data. To set up data binding, the control's **data context** must be set to a reference to the data object. The **binding** for the control must be set to a **binding path**, which is the property on the object to bind to. Once these things are set, the control will automatically read the source property and display the data as its content.

To set up data binding in XAML, set the property that you want to bind to **`{Binding Path}`**:

The binding path for this TextBlock control is the Cash property. It will display the value of Cash for whatever object it's bound to.

```
<TextBlock x:Name="walletTextBlock" Text="{Binding Cash}"/>
```

Then you just need an object to bind to — in this case, a `Guy` object named `joe` whose `Cash` property is set to the decimal value 325.50. Giving the TextBlock's `DataContext` a reference to the `Guy` object sets up the context.

```
Guy joe = new Guy("Joe", 47, 325.50M);

walletTextBlock.DataContext = joe;
```

The data context for this TextBlock is a reference to a Guy object. The TextBlock will read any bound properties from the Guy object.

Now your binding is set up! You've set the data context to an instance of Guy, you've set the binding path to the `Cash` property. The TextBlock sees its binding is set to `Cash`, then **looks for a property called `Cash`** on its data object.

You can actually leave out the path and just set the property to `{Binding}`. In this case, it will call the `Guy` object's `ToString()` method.

Two-way binding can get or set the source property

Binding can read data from the data object. It can also use **two-way binding** to modify the source property:

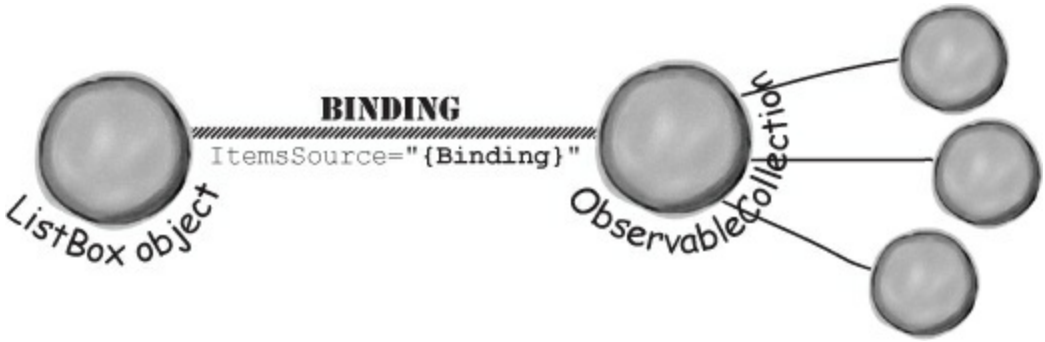
```
<TextBox x:Name="ageTextBox" Text="{Binding Age, Mode=TwoWay}"/>
```

This TextBox's binding path is the `Age` property, and the binding is set to two-way mode. When the page is displayed, the TextBox will show the value of the `Age` property of whatever object it's bound to. If you change the value in the TextBox, the control will call the `Age` property's set accessor to update the value.

Data binding is built to cause you as few headaches as possible. If you set the binding path to a property that isn't in the data context, it won't display or set any data, but it also won't cause your program to break.

Bind to collections with ObservableCollection

Some controls like TextBlock or TextBox display a string. Other controls like ScrollViewer display content from an object. But you've also seen controls that display a collection: ListBox and ComboBox. That's why .NET comes with `ObservableCollection<T>`, a collection class that's built specifically for data binding. It works a lot like `List<T>` (you'll see it in action on the next page).



NOTE

When you bind the ListBox's ItemsSource property to an ObservableCollection, it displays all of the items in the collection.

Use code for binding (without using any XAML at all!)

If you examine a control, you won't actually see a property called `Binding`. There's no direct way in C# to get a reference to a property on an object, just the whole object. When you create the XAML code for a data binding, it sets up the binding using **an instance of a `Binding` object** that stores the name of the target property as a string. Here's code-behind that creates a `Guy` object, then sets up binding for a TextBlock called `walletTextBlock` so its `Text` property is bound to the `Guy` object's `Cash` property.

```
Guy joe = new Guy("Joe", 47, 325.50M);
Binding cashBinding = new Binding();
cashBinding.Path = new PropertyPath("Cash");
cashBinding.Source = joe;
walletTextBlock.SetBinding(TextBlock.TextProperty, cashBinding);
```

There's a class called DependencyProperty, and the TextBlock class has a whole bunch of static properties that are instances of it. One of them is called TextProperty.

XAML controls can contain text...and more

Let's talk a little more about XAML **markup** (that's what the M in XAML stands for, and it refers to the tags that define the page) and **code-behind** (the code in the *.cs* file that's joined with the markup).

When you use a Grid or StackPanel control, you add the controls that they contain between the opening and closing tags. You can also use the same thing for other kinds of controls. For controls like TextBlock and TextBox, you can set the `Text` property by adding text and a closing tag:

```
<TextBlock>This is the text to display</TextBlock>
```

This is just like using the Text property.

When you do this, you'll use `<LineBreak/>` instead of `
` to add line breaks. What you're really doing here is specifying the Unicode character U+0013, which is interpreted as a line break. You can also specify it in hex: `
` gives you a line break, `£` gives you a £ character (remember Charmap?).

```
<TextBlock>First line<LineBreak/>Second line</TextBlock>
```

Try adding that TextBlock to a XAML page, then use Edit Text to edit it and press Shift-Enter to add a break. The IDE will add this:

```
<TextBlock>
    <Run Text="First line"/>
    <LineBreak/>
    <Run Text="Second line"/>
</TextBlock>
```

All three of those options may look the same on the screen, but they create different object graphs. Each `<Run>` tag is turned into its own string object, and each of those strings can be given its own name:

```
<Run Text="First line" x:Name="firstLine" />
```

You can use this to modify that string in your C# code behind the XAML form:

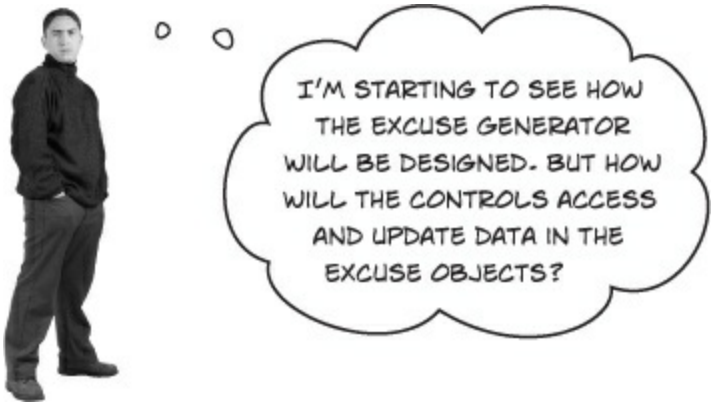
```
firstLine.Text = "This is new text for the first line";
```

Content controls like ScrollViewer have a `Content` property (instead of a `Text` property) that doesn't have to be text — it can contain any control. And there are many content controls. One useful one is `Border`, which you can use to add a background and border to controls like TextBlock that don't have one:

```
<Border Background="Blue"
        BorderBrush="Green" BorderThickness="3">
</Border>
```

NOTE

ScrollViewer inherits from ContentControl, which is the same control you used to create your enemy in Save the Humans. Your ContentControl contained a Grid, which contained three Ellipses.


I'M STARTING TO SEE HOW THE EXCUSE GENERATOR WILL BE DESIGNED. BUT HOW WILL THE CONTROLS ACCESS AND UPDATE DATA IN THE EXCUSE OBJECTS?

THERE ARE NO DUMB QUESTIONS

Q: **Q: My page had a Grid that contained another Grid, which contained a StackPanel. Is there a limit to how many controls can live inside other controls?**

A: **A:** No. You can nest controls inside of other controls, and those controls can in turn contain additional controls. In fact, later on in the book you'll learn about how to build up your own controls by starting with a container and adding content to it. You can put a Grid into *any* content control — you already did this once when you created the enemy out of a Grid and three Ellipse controls in *Save the Humans*. That's one of the strengths of using XAML to design your apps: it gives you the ability to create complex pages out of simple controls.

Q: **Q: If I can lay out the same page using either a Grid or a StackPanel, which one should I use?**

A: **A:** It depends a lot on the situation. There is no "right" answer: sometimes it makes more sense to use a StackPanel, sometimes it makes sense to use a Grid, and sometimes it makes sense to combine them. And those aren't the only options, either. You used a Canvas in *Save the Humans*, which is a container control that allows you to use the `Canvas.Left` and `Canvas.Top` properties to position controls at specific coordinates. All three of these controls are subclasses of `Panel`, and among the behaviors they inherit from that base class is the ability to host multiple other controls.

Q: **Q: Does that mean there are controls that can only host a single control?**

A: **A:** Yes. Try adding a ScrollViewer to a page. Then nest two other controls inside it. Here's what you'll see:

```
<ScrollViewer>
    <TextBox/>
    <Button/>
</Sc
```

The property 'Content' is set more than once.

That's because this XAML sets the `Content` property on the `ScrollViewer` object, and that property is of type `object`. If you replace the ScrollViewer tags with Grid tags:

```
<Grid>
    <TextBox/>
    <Button/>
</Grid>
```

This will work just fine, because the contained controls are added to a collection called `Children`. (Your code in *Save the Humans* used the `Children` collection to add enemies.)

Q: **Q: Why do some controls like TextBlock have a `Text` property instead of a `Content` property?**

A: **A:** Because those controls can only host text, so they have a `string` property called `Text` instead of an `object` property called `Content`. This is called the **default property** of the control. The default property of a Grid or StackPanel is its `Children` collection.

Q: **Q: Should I be typing in my XAML code, or using the IDE's designer to drag controls out of the toolbox?**

A: **A:** You should try both, and do what's most comfortable to you. A lot of developers rely heavily on the designer in the IDE, but many developers rarely use the designer at all because they find it faster to type the XAML. The IDE's IntelliSense makes it especially easy to type XAML.

Q: **Q: Remind me again why I had to learn WinForms? Why couldn't I just jump straight to XAML and Windows Store apps?**

A: **A:** Because there are a lot of concepts that make XAML much easier to understand. Take the `Children` collection, for example. If you didn't understand collections, would the answer to the third question on this page make sense? Maybe. But it's a lot more obvious once you do understand collections. On the other hand, it's really easy to drag controls out of the toolbox and onto the form. There's a lot less depth to WinForms than there is to page design with XAML (which makes sense, since XAML is a much newer and more flexible technology). Spending several chapters on WinForms made it easy for you to get the hang of designing visual applications and building interesting projects. That, in turn, helped you get many of these concepts into your brain. You'll absorb XAML much faster now that you have them there. There's also a lot of value in seeing the same project done two different ways. That's why we're revisiting some of the projects from previous chapters: you'll understand more about both WinForms and Windows Store apps by seeing the same app done in both.

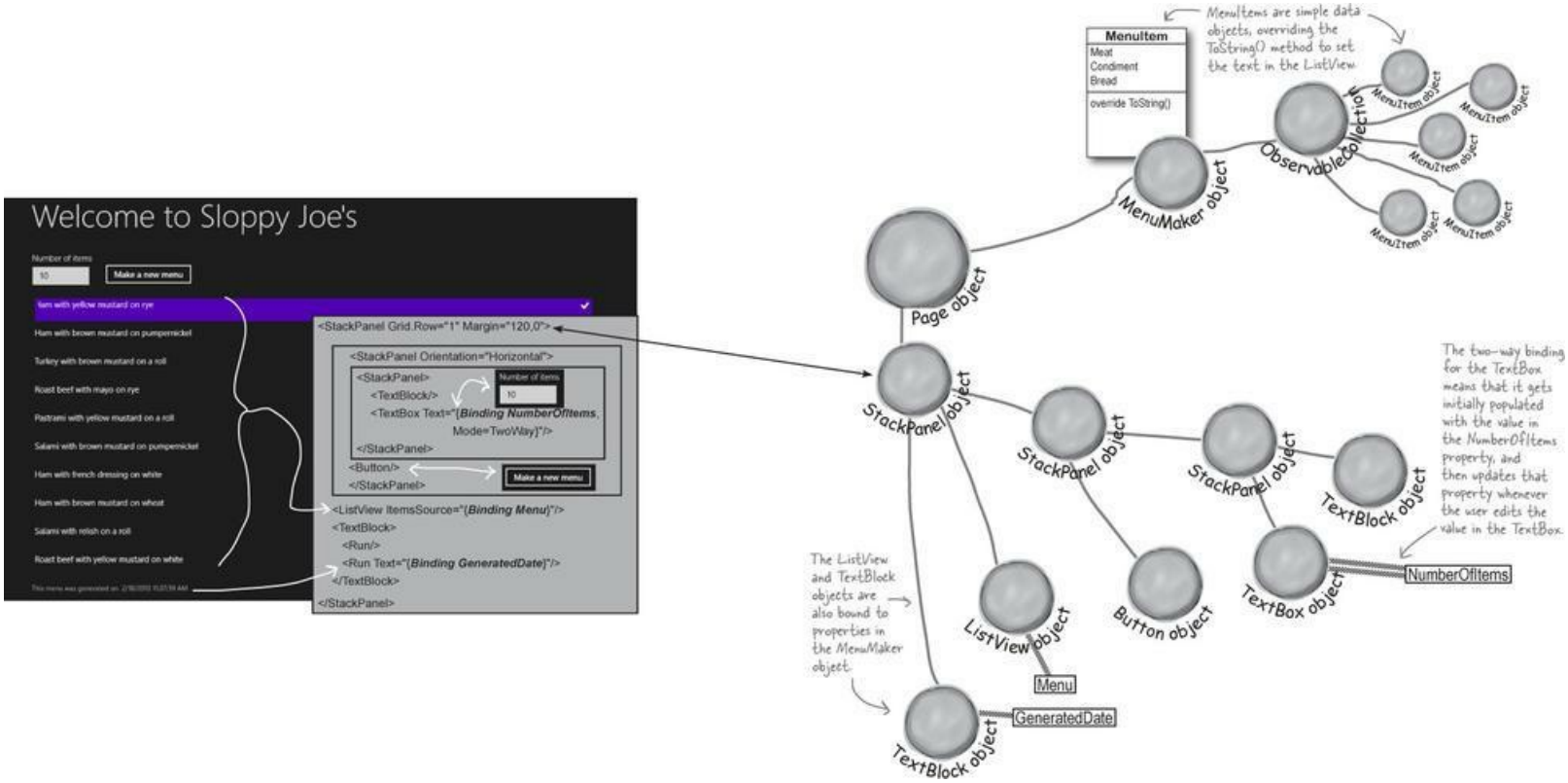
WinForms is a great tool for learning and exploring C#, but XAML is a much more capable tool for building flexible and effective apps.

Use data binding to build Sloppy Joe a better menu

Remember Sloppy Joe from Chapter 4? Well, he's using Windows 8 now, and he wants a Window Store app for his sandwich menu. Let's build him one.

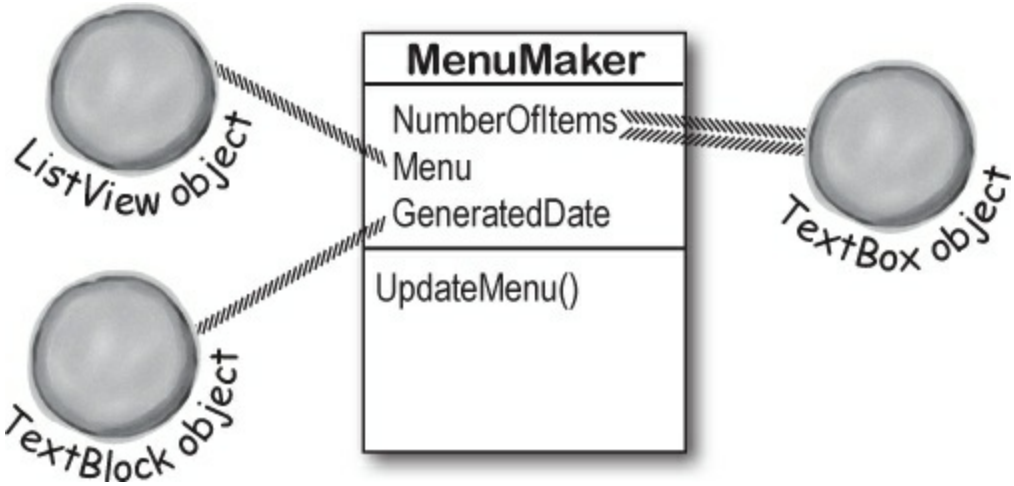
Here's the page we're going to build.

It uses one-way data binding to populate a ListView and a Run inside a TextBlock, and it uses two-way data binding for a TextBox, using one of its `<Run>` tags to do the actual binding.



We'll need an object with properties to bind to.

The `Page` object will have an instance of the `MenuMaker` class, which has three public properties: an int called `NumberOfItems`, an `ObservableCollection` of menu items called `Menu`, and a `DateTime` called `GeneratedDate`.



The `Page` object creates an instance of `MenuMaker` and uses it for the data context.

The constructor for the `Page` object will set the StackPanel's `DataContext` property to an instance of `MenuMaker`. The binding will all be done in XAML.

The TextBox uses two-way binding to set the number of menu items.

That means the TextBox doesn't need an `x:Name` property. Since it's bound to the `NumberOfItems` property in the `MenuMaker` object, we don't need to write any C# code that refers to it.

The button tells the `MenuMaker` to update.

The button calls the `MenuMaker`'s `UpdateMenu()` method, which updates its menu by clearing the `ObservableCollection` and then adding new `MenuItems` to it. The ListView will automatically update any time the `ObservableCollection` changes.

Here's a coding challenge. Based on what you've read so far, how much of the new and improved Sloppy Joe app can you build before you flip the page and see the code for it?

DO THIS!

- **❶ Create the new project and replace MainPage.xaml with a Basic Page.**
  Create a **new Windows Store app**. Then delete *MainPage.xaml*, and **add a new Basic Page called *MainPage.xaml* to replace it**. You'll need to rebuild the project after you replace the page. This is exactly the same thing you did with *Save the Humans* (flip back to Chapter 1 if you need a refresher).
- **❷ Add the new and improved `MenuMaker` class.**
  You've come a long way since Chapter 4. Let's build a well-encapsulated class that lets you set the number of items with a property. You'll create an `ObservableCollection` of `MenuItem` in its constructor, which is updated every time the `UpdateMenu()` is called. That method will also update a `DateTime` property called `GeneratedDate` with a timestamp for the current menu. Add this `MenuMaker` class to your project:

Just right-click on the project name in the Solution Explorer and add a new class, just like you did with other projects.

```
using System.Collections.ObjectModel;

class MenuMaker {
    private Random random = new Random();
    private List<String> meats = new List<String>()
                { "Roast beef", "Salami", "Turkey", "Ham", "Pastrami" };
    private List<String> condiments = new List<String>() { "yellow mustard",
             "brown mustard", "honey mustard", "mayo", "relish", "french dressing" };
    private List<String> breads = new List<String>() { "rye", "white", "wheat",
            "pumpernickel", "italian bread", "a roll" };

    public ObservableCollection<MenuItem> Menu { get; private set; }
    public DateTime GeneratedDate { get; private set; }
    public int NumberOfItems { get; set; }

    public MenuMaker() {
        Menu = new ObservableCollection<MenuItem>();
        NumberOfItems = 10;
        UpdateMenu();
    }
    private MenuItem CreateMenuItem() {
        string randomMeat = meats[random.Next(meats.Count)];
        string randomCondiment = condiments[random.Next(condiments.Count)];
        string randomBread = breads[random.Next(breads.Count)];
        return new MenuItem(randomMeat, randomCondiment, randomBread);
    }
    public void UpdateMenu() {
        Menu.Clear();
        for (int i = 0; i < NumberOfItems; i++) {
            Menu.Add(CreateMenuItem());
        }
        GeneratedDate = DateTime.Now;
    }
}
```

You'll need this using line because ObservableCollection<T> is in this namespace.

You'll use data binding to display data from these properties on your page. You'll also use two-way binding to update NumberOfItems.

The new CreateMenuItem() method returns MenuItem objects, not just strings. That will make it easier to change the way items are displayed if we want.

Take a closer look at how this works. It never actually creates a new MenuItem collection. It updates the current one by clearing it and adding new items.

What happens if the NumberOfItems is set to a negative number?

USE DATETIME TO WORK WITH DATES

You've already seen the DateTime type that lets you store a date. You can also use it to create and modify dates and times. It has a static property called Now that returns the current time. It also has methods like AddSeconds() for adding and converting seconds, milliseconds, days, etc., and properties like Hour and DayOfWeek to break down the date. How timely!

- **❸ Add the `MenuItem` class.**
  You've already seen how you can build more flexible programs if you use classes instead of strings to store data. Here's a simple class to hold a menu item — add it to your project, too:

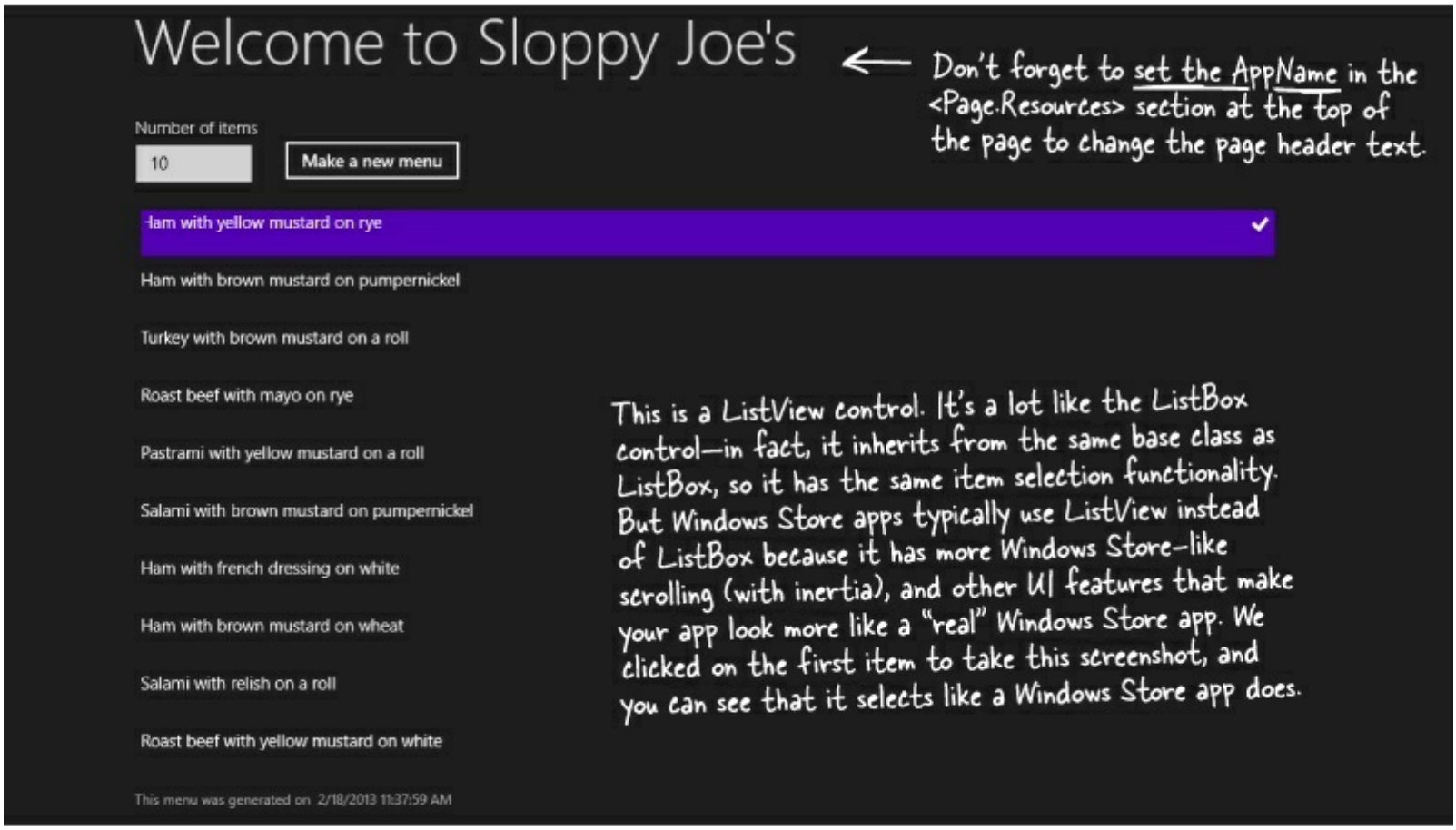
```
class MenuItem {
    public string Meat { get; private set; }
    public string Condiment { get; private set; }
    public string Bread { get; private set; }

    public MenuItem(string meat, string condiment, string bread) {
        Meat = meat;
        Condiment = condiment;
        Bread = bread;
    }

    public override string ToString() {
        return Meat + " with " + Condiment + " on " + Bread;
    }
}
```

The three strings that make up the item are passed into the constructor and held in read-only automatic properties.

Override the ToString() method so the MenuItem knows how to display itself.

- **❹ Build the XAML page.**
  Here's the screenshot. Can you build it using StackPanels? The TextBox has a width of 100. The bottom TextBlock has the style `CaptionTextBlockStyle`, and it has two `<Run>` tags (the second one just holds the date).

> **NOTE**
>
> Don't add dummy data this time. We'll let data binding do that for us.



Can you build this page on your own just from the screenshot before you see the XAML?

- **❺ Add object names and data binding to the XAML.**
  Here's the XAML that gets added to *MainPage.xaml*. Make sure you **add it to the outermost grid just above the `<!-- Back button and page title -->` XAML comment**, just like you did in the *Save the Humans* main page. We named the button `newMenu`. Since we used data binding of the ListView, TextBlock, and TextBox, we didn't need to give them names. *(Here's a shortcut. We didn't even really need to name the button, we did it just to get the IDE to automatically add an event handler named newMenu_Click when we double-clicked it in the IDE. Try it out!)*

```
<StackPanel Grid.Row="1" Margin="120,0" x:Name="pageLayoutStackPanel">
    <StackPanel Orientation="Horizontal" Margin="0,0,0,20">
        <StackPanel Margin="0,0,20,0">
            <TextBlock Style="{StaticResource BodyTextBlockStyle}"
                Text="Number of items" Margin="0,0,0,10" />
            <TextBox Width="100" HorizontalAlignment="Left"
                     Text="{Binding NumberOfItems, Mode=TwoWay}" />
        </StackPanel>
        <Button x:Name="newMenu" VerticalAlignment="Bottom" Click="newMenu_Click"
        Content="Make a new menu" Margin="0,0,20,0"/>
    </StackPanel>
    <ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0" />
    <TextBlock Style="{StaticResource CaptionTextBlockStyle}">
        <Run Text="This menu was generated on " />
        <Run Text="{Binding GeneratedDate}"/>
    </TextBlock>
</StackPanel>
```

We need two-way data binding to both get and set the number of items with the TextBox.

Here's that ListView control. Try swapping it out for ListBox to see how it changes your page.

This is where <Run> tags come in handy. You can have a single TextBlock but only bind part of its text.

- **❻ Add the code-behind for the page to MainPage.xaml.cs.**
  The page constructor creates the menu collection and the `MenuMaker` instance, and sets the data contexts for the controls that use data binding. It also needs a `MenuMaker` field called `menuMaker`.

```
MenuMaker menuMaker = new MenuMaker();

public MainPage() {
    this.InitializeComponent();

    pageLayoutStackPanel.DataContext = menuMaker;
    // ... the rest of the MainPage constructor stays the same ...
```

Your main page's class in MainPage.xaml.cs gets a MenuMaker field, which is used as the data context for the StackPanel that contains all of the bound controls.

You just need to set the data context for the outer StackPanel. It will pass that data context on to all of the controls contained inside it.

Finally, double-click on the button to generate a method stub for its `Click` event handler. Here's the code for it — it just updates the menu:

```
private void newMenu_Click(object sender, RoutedEventArgs e) {
    menuMaker.UpdateMenu();
}
```

There's an easy way to rename an event handler so that it updates XAML and C# code at the same time. Flip to leftover #8 in the appendix to learn more about the refactoring tools in

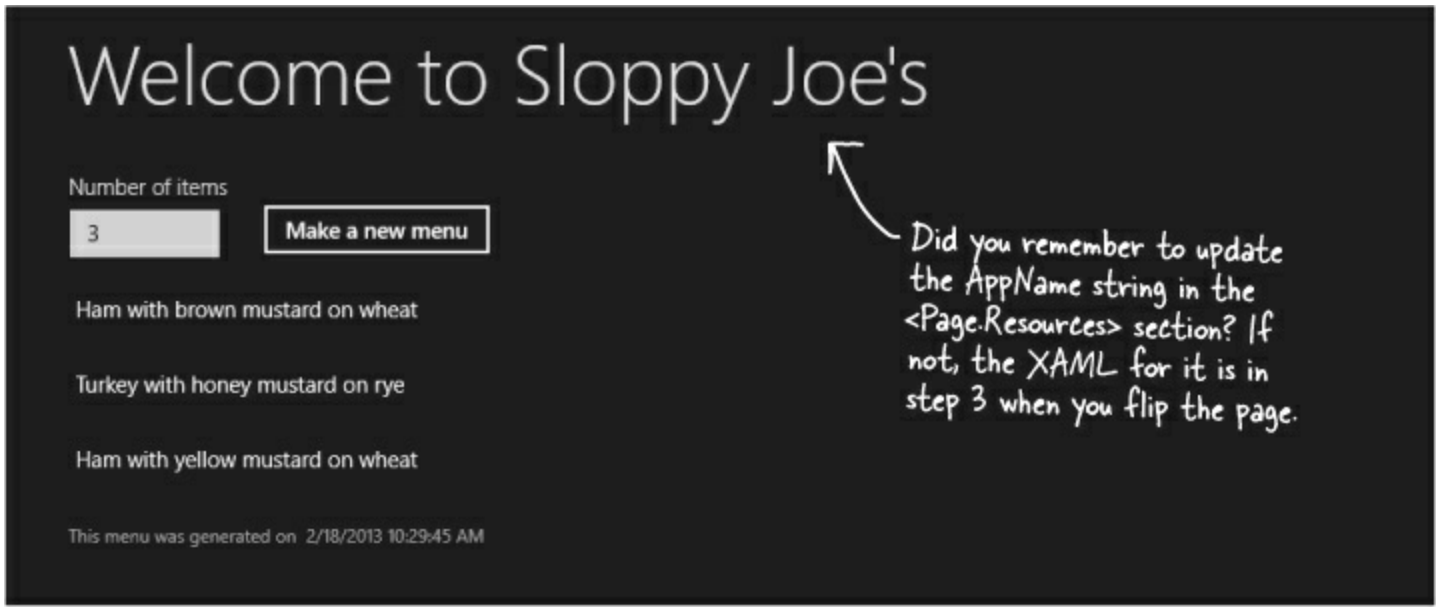
the IDE.

RELAX

There's a lot of code-behind in the Blank Page template.

Open up MainPage.xaml.cs — there's a lot of C# code in there already! You'll learn what it does later in the book. Just add your code-behind methods, fields, etc., just like you did in Chapter2..

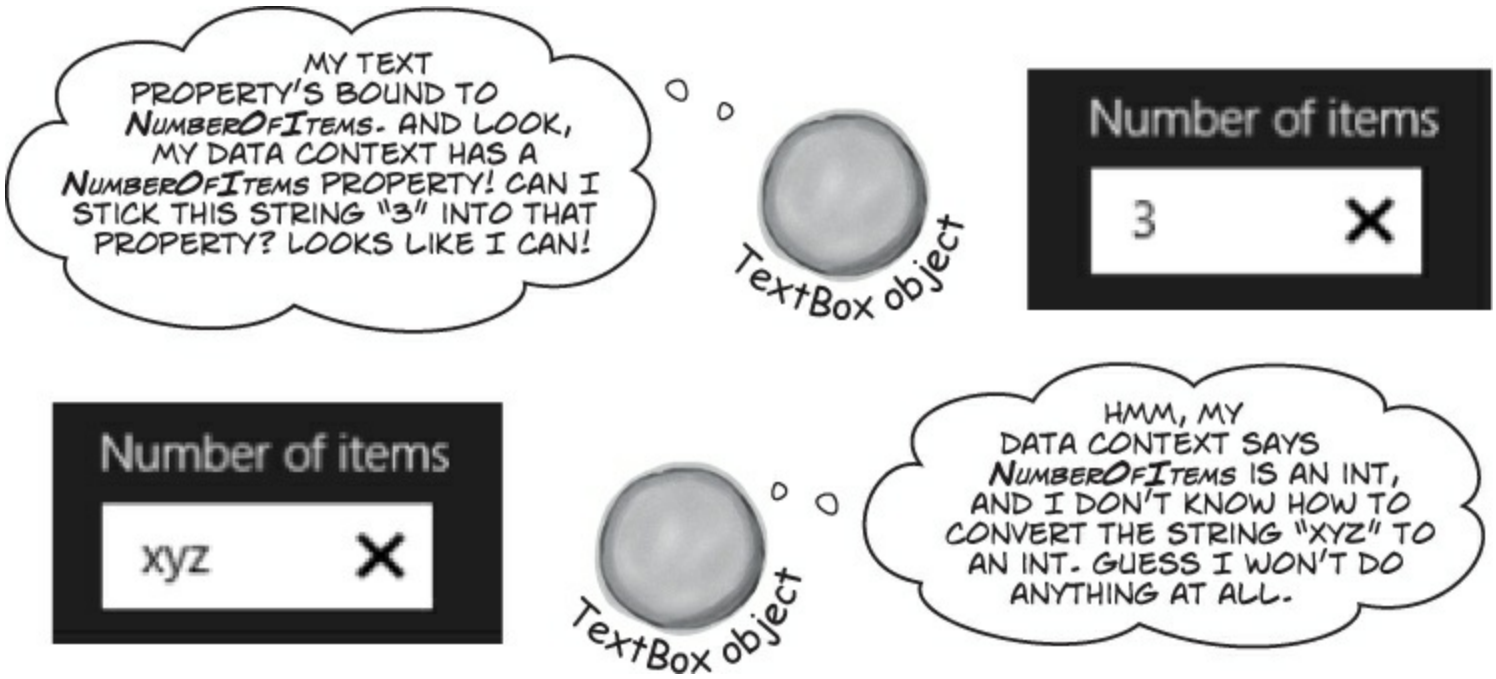
Now run your program! Try changing the TextBox to different values. Set it to 3, and it generates a menu with three items:



Now you can play with binding to see just how flexible it is. Try entering "xyz" or no data at all into the TextBox. Nothing happens! When you enter data into the TextBox, you're giving it a string. The TextBox is pretty smart about what it does with that string. It knows that its binding path is `NumberOfItems`, so it looks in its data context to see if there are any properties with that name, and then does its best to convert the string to whatever that property's type is.

NOTE

Keep your eye on the generated date. It's not updating, even though the menu updates. Hmm, maybe there's still something we need to do.


MY TEXT PROPERTY'S BOUND TO NumberOfItems. AND LOOK, MY DATA CONTEXT HAS A NumberOfItems PROPERTY! CAN I STICK THIS STRING "3" INTO THAT PROPERTY? LOOKS LIKE I CAN!
TextBox object
Number of items
3
Number of items
xyz
TextBox object
HMM, MY DATA CONTEXT SAYS NumberOfItems IS AN INT, AND I DON'T KNOW HOW TO CONVERT THE STRING "XYZ" TO AN INT. GUESS I WON'T DO ANYTHING AT ALL.

Use static resources to declare your objects in XAML

When you build a page with XAML, you're creating an object graph with objects like StackPanel, Grid, TextBlock, and Button. And you've seen that there's no magic or mystery to any of that — when you add a `<TextBox>` tag to your XAML, then your page object will have a TextBox field with a reference to an instance of TextBox. And if you give it a name using the `x:Name` property, your code-behind C# code can use that name to access the TextBox.

You can do exactly the same thing to create instances of *almost any* class and store them as fields in your page by adding a **static resource** to your XAML. And data binding works particularly well with static resources, especially when you combine it with the visual designer in the IDE. Let's go back to your program for Sloppy Joe and move the `MenuMaker` to a static resource.

THERE ARE NO DUMB QUESTIONS

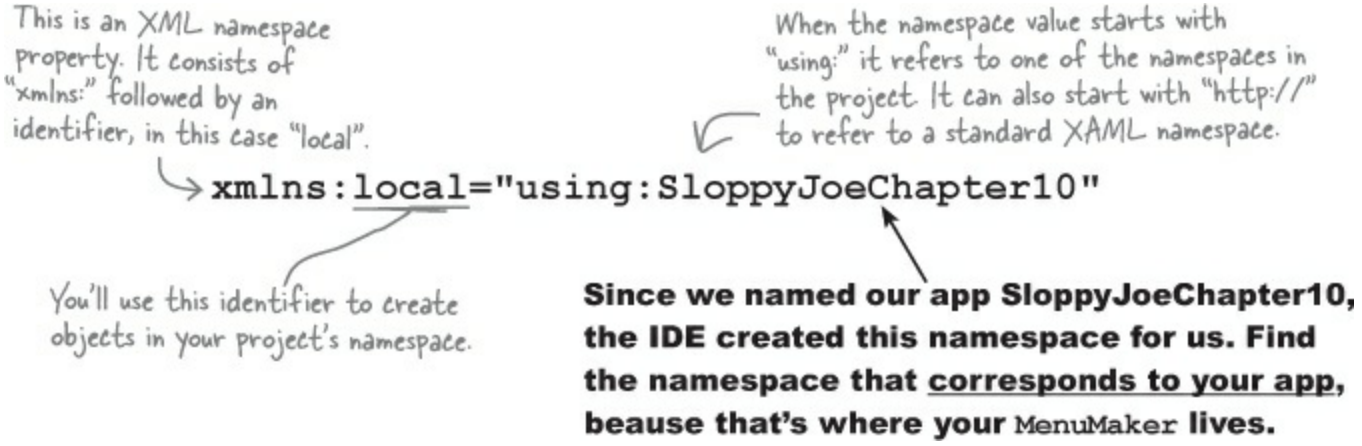
Q: **Q: Hey, there's no Close button! How do I quit my app?**

A: **A:** Windows Store apps don't have Close buttons by default, because you typically never quit most apps. Windows Store apps follow an **application lifecycle** with three states: not running, running, and suspended. Apps can be suspended if the user switches away or Windows enters a low power state. And if it needs to reclaim the memory, Windows can terminate it. Later in the book you'll learn how to make your app work with this lifecycle.

- **❶ DELETE THE MENUMAKER FIELD FROM THE CODE-BEHIND.**
  You're going to be setting up the `MenuMaker` class and the data context in the XAML, so delete these lines from your C# code:

```
MenuMaker menuMaker = new MenuMaker();

public MainPage() {
    this.InitializeComponent();

    pageLayoutStackPanel.DataContext = menuMaker;
```

- **❷ TAKE A CLOSE LOOK AT THE NAMESPACES FOR YOUR PAGE.**
  Look at the top of the XAML code for your page, and you'll see that the page's opening tag has a set of `xmlns` properties. Each of these properties defines a namespace. Look for the one that starts with `xmlns:local` and has your project's namespace. It should look like this:



- **❸ ADD THE STATIC RESOURCE TO YOUR XAML AND SET THE DATA CONTEXT.**
  Find the `<Page.Resources>` section of your page and type **`<local:`** to pop up an IntelliSense window:

```
<Page.Resources>

    <local:
            App
            MenuItem
            MenuMaker

    <!-- TODO: Delete this line if the key AppName is declared in App.xaml -->
    <x:String x:Key="AppName">Welcome to Sloppy Joe's</x:String>
</Page.Resources>
```

You can only add static resources if their classes have parameterless consructors. This makes sense! If the constructor has a parameter, how would the XAML page know what arguments to pass to it?

The window shows all of the classes in the namespace that you can use. Choose `MenuMaker`, and give it the name `menuMaker`:

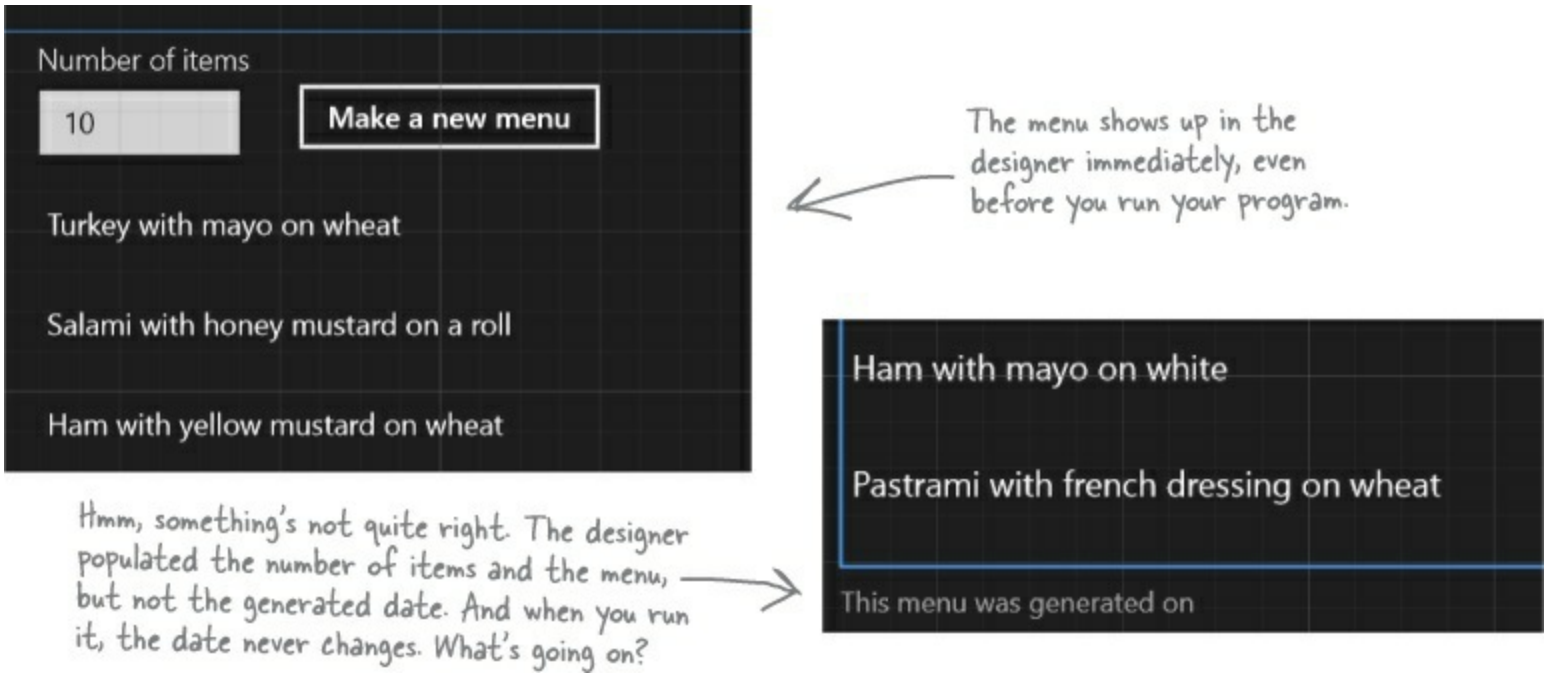
```
<local:MenuMaker x:Name="menuMaker"/>
```

Now your page has a static `MenuMaker` resource called `menuMaker`.

- **❹ SET THE DATA CONTEXT FOR YOUR STACKPANEL AND ALL OF ITS CHILDREN.** Then go to the outermost StackPanel, remove the `x:Name` tag, and set its `DataContext` property:

```
<StackPanel Grid.Row="1" Margin="120,0"
      DataContext="{StaticResource ResourceKey=menuMaker}">
```

Your program will still work, just like before. But did you notice what happened in the IDE when you added the data context to the XAML? As soon as you added it, the IDE created an instance of MenuMaker and used its properties to populate all of the controls that were bound to it. You got a menu generated immediately, right there in the designer — before you even ran your program. Neat!



Use a data template to display objects

When you show items in a list, you're showing contents of ListViewItem (which you use for ListViews), ListBoxItem, or ComboBoxItem controls, which get bound to objects in an `ObservableCollection`. Each ListViewItem in the Sloppy Joe menu generator is bound to a `MenuItem` object in its `Menu` collection. The ListViewItem objects call the `MenuItem` objects' `ToString()` methods by default, but you can use a **data template** that uses data binding to display data from the bound object's properties.

Modify the `<ListView>` tag to add a basic data template. It uses the basic `{Binding}` to call the item's `ToString()`.

Leave the ListView tag intact, but replace /> with > and add a closing </ListView> tag at the bottom. Then add the ListView.ItemTemplate tag to contain the data template.

This is a really basic data template, and it looks just like the default one used to display the ListViewItems.

```
<ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0">
    <ListView.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding}"/>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Adding a {Binding} without a path just calls the ToString() method of the bound object.

Change your data template to add some color to your menu.

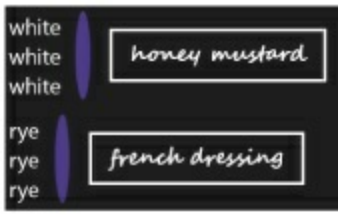
Replace the <DataTemplate>, but leave the rest of the ListView intact.

You can bind individual Run tags. You can change each tag's color, font, and other properties too.

```
<DataTemplate>
    <TextBlock>
        <Run Text="{Binding Meat}" Foreground="Blue"/><Run Text=" on "/>
        <Run Text="{Binding Bread}" FontWeight="Light"/><Run Text=" with "/>
        <Run Text="{Binding Condiment}" Foreground="Red" FontWeight="ExtraBold"/>
    </TextBlock>
</DataTemplate>
```

Go crazy! The data template can contain any controls you want.

The DataTemplate object's Content property can only hold one object, so if you want multiple controls in your data template, you'll need a container like StackPanel.




```
<DataTemplate>
    <StackPanel Orientation="Horizontal">
        <StackPanel>
            <TextBlock Text="{Binding Bread}"/>
            <TextBlock Text="{Binding Bread}"/>
            <TextBlock Text="{Binding Bread}"/>
        </StackPanel>
        <Ellipse Fill="DarkSlateBlue" Height="Auto" Width="10" Margin="10,0"/>
        <Button Content="{Binding Condiment}" FontFamily="Segoe Script"/>
    </StackPanel>
</DataTemplate>
```

THERE ARE NO DUMB QUESTIONS

Q: **Q: So I can use a StackPanel or a Grid to lay out my page. I can use XAML static resources, or I can use fields in code-behind. I can set properties on controls, or I can use data binding. Why are there so many ways to do the same things?**

A: **A:** Because C# and XAML are extremely flexible tools for building apps. That flexibility makes it possible to design very detailed pages that work on many different devices and displays. This gives you a very large toolbox that you can use to get your pages *just right*. So don't look at it as a confusing set of choices; look at it as many different options that you can choose from.

Q: **Q: I'm still not clear on how static resources work. What happens when I add a tag inside `<Page.Resources>`?**

A: **A:** When you add that tag, it udpates the `Page` object. Find the `AppName` resource that you changed to set the page header:

```
<x:String x:Key="AppName">Welcome to Sloppy
Joe's</x:String>
```

Now go through the code that the IDE added as part of the Basic Page template to find where it uses the resource:

```
<TextBlock x:Name="pageTitle"
      Grid.Column="1"
      Text="{StaticResource AppName}"
      Style="{StaticResource ...
```

The page uses this static resource to set the text. So what's going on behind the scenes? You can use the IDE to see what's going on. Put a breakpoint in your button event handler, then run the code and press the button. Add `this.Resources["AppName"]` to the Watch window, and you'll see that it contains a reference to a string. And every static resource works the same way — when you add a static resource to the code, it creates an object and adds it to a collection called `Resources`.

Q: **Q: Can I use that `{StaticResource}` syntax in my own code, or is it just for templates like Blank Page?**

A: **A:** Absolutely, you can set up resources and use them just like that. There's nothing special about the Blank Page template, or any other templates you'll use in this book. They just use regular XAML and C#, and they don't do anything that you can't do yourself.

Q: **Q: I used `x:Name` to set my `MenuMaker` resource's name, but the `AppName` resource uses `x:Key`. What's the difference?**

A: **A:** When you use the `x:Key` property in a static resource, it adds the resource to the `Resources` collection using that key, but it doesn't create a field (so you can't enter `AppName` into your C# code, you can only access it using the `Resources` collection). When you use the `x:Name` property, it adds it to the `Resources` collection, but it also adds a field to the `Page` object. That's how you were able to call the `UpdateMenu()` method on the `MenuMaker` static resource.

Q: **Q: Does my binding path have to be a string property?**

A: **A:** No, you can bind a property of any type. If it can be converted between the source and property types, then the binding will work. If not, the data will be ignored. And remember, not all properties on your controls are text, either. Let's say you've got a bool in your data context called `EnableMyObject`. You can bind it to any Boolean property, like `IsEnabled`. This will enable or disable the control based on the value of the `EnableMyObject` property:

```
IsEnabled="{Binding EnableMyObject}"
```

Of course, if you bind it to a text property it'll just print `True` or `False` (which, if you think about it, makes perfect sense).

Q: **Q: Why did the IDE display the data in my form when I added the static resource and set the data context in XAML, but not when I did it in C#?**

A: **A:** Because the IDE understands your XAML, which has all of the information that it needs to create the objects to render your page. As soon as you added the `MenuMaker` resource to your XAML code, the IDE created an instance of `MenuMaker`. But it couldn't do that from the `new` statement in its constructor, because there could be many other statements in the constructor, and they would need to be run. The IDE only runs the code-behind C# code when the program is executed. But if you add a static resource to the page the IDE will create it, just like it creates instances of TextBlock, StackPanel, and the other controls on your page. It sets the controls' properties to show them in the designer, so when you set up the data context and binding paths, those got set as well, and your menu items showed up in the IDE's designer.

The static resources in your page are instantiated when the page is first loaded and can be used at any time by the objects in the app.

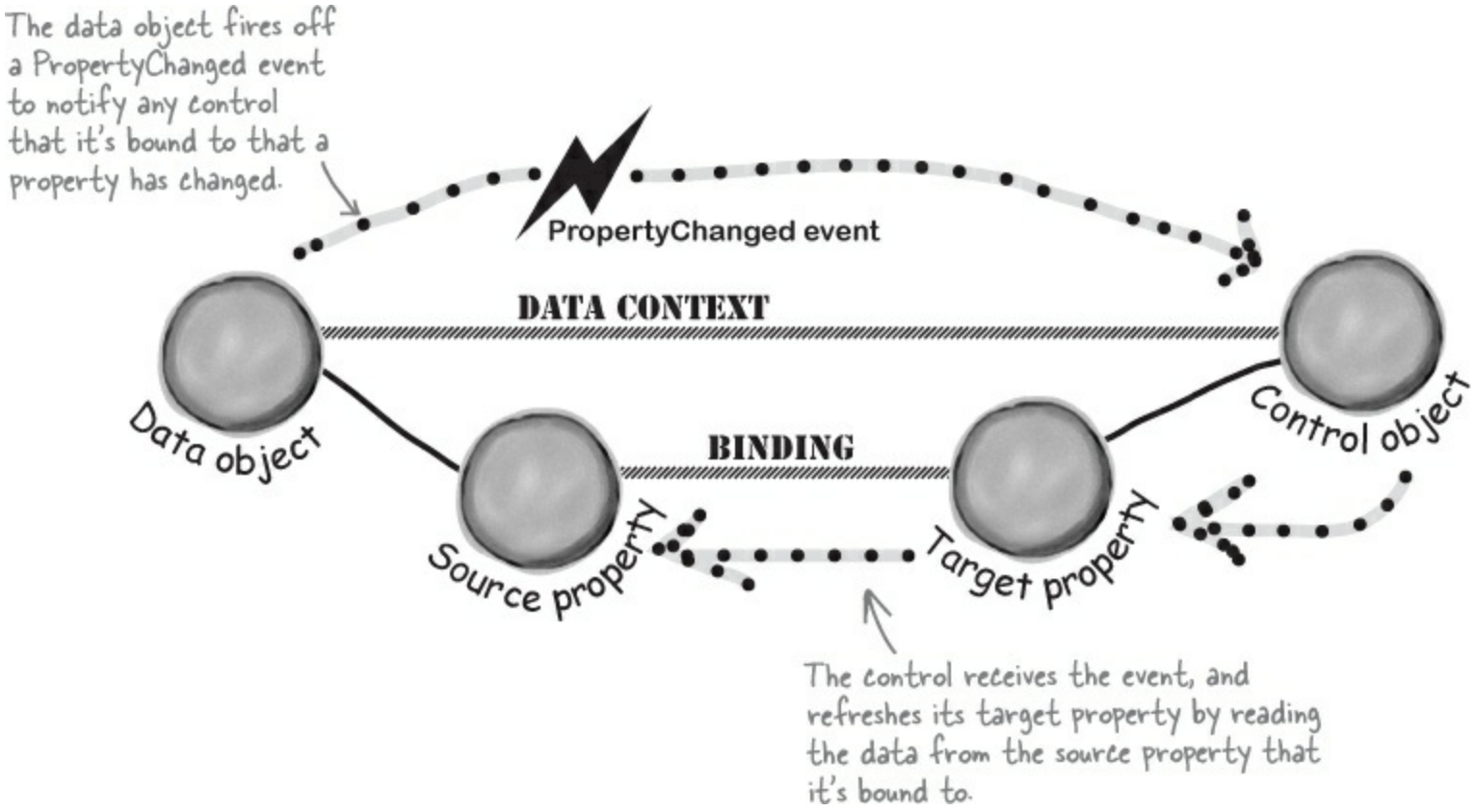
NOTE

The name "static resource" is a little misleading. Static resources are definitely created for each instance; they're not static fields!

INotifyPropertyChanged lets bound objects send updates

When the `MenuMaker` class updates its menu, the ListView that's bound to it gets updated. But the `MenuMaker` updates the `GeneratedDate` property at the same time. Why doesn't the TextBlock that's bound to it get updated too? The reason is that every time an `ObservableCollection` changes, it **fires off an event** to tell any bound control that its data has changed. This is just like how a `Button` control raises a `Click` event when it's clicked, or a `Timer` raises a `Tick` event when its interval elapses. Whenever you add, remove, or delete items from an `ObservableCollection`, it raises an event.

You can make your data objects notify their target properties and bound controls that data has changed, too. All you need to do is **implement the `INotifyPropertyChanged` interface**, which contains a single event called `PropertyChanged`. Just fire off that event whenever a property changes, and watch your bound controls update themselves automatically.



WATCH IT!

Collections work *almost* the same way as data objects.

The ObservableCollection<T> object doesn't actually implement INotifyPropertyChanged. Instead, it implements a closely related interface called **INotifyCollectionChanged** *that fires off a CollectionChanged event instead of a PropertyChanged event. The control knows to look for this event because ObservableCollection implements the INotifyCollectionChanged interface. Setting a ListView's DataContext to an INotifyCollectionChanged object will cause it to respond to these events.*

Modify MenuMaker to notify you when the GeneratedDate property changes

`INotifyPropertyChanged` is in the `System.ComponentModel` namespace, so start by adding this `using` statement to the top of the `MenuMaker` class file:

```
using System.ComponentModel;
```

Update the `MenuMaker` class to implement `INotifyPropertyChanged`, and then use the IDE to automatically implement the interface:

RELAX

This is the first time you're raising events.

You've been writing event handler methods since Chapter 1, but this is the first time you're firing an event. You'll learn all about how this works and what's going on in Chapter 15. For now, all you need to know is that an interface can include an event, and that your `OnPropertyChanged()` method is following a standard C# pattern for raising events to other objects.

This will be a little different than what you saw in Chapter 7 and Chapter 8. It won't add any methods or properties. Instead, it will add an event:

```
public event PropertyChangedEventHandler PropertyChanged;
```

Next, add this `OnPropertyChanged()` method, which you'll use to raise the `PropertyChanged` event.

```
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

This is a standard .NET pattern for raising events.

Now all you need to do to notify a bound control that a property is changed is to call `OnPropertyChanged()` with the name of the property that's changing. We want the TextBlock that's bound to `GeneratedDate` to refresh its data every time the menu is updated, so all we need to do is add one line to the end of `UpdateMenu()`:

```
public void UpdateMenu() {
    Menu.Clear();
    for (int i = 0; i < NumberOfItems; i++) {
        Menu.Add(CreateMenuItem());
    }
    GeneratedDate = DateTime.Now;

    OnPropertyChanged("GeneratedDate");
}
```

Now the date should change when you generate a menu.

WATCH IT!

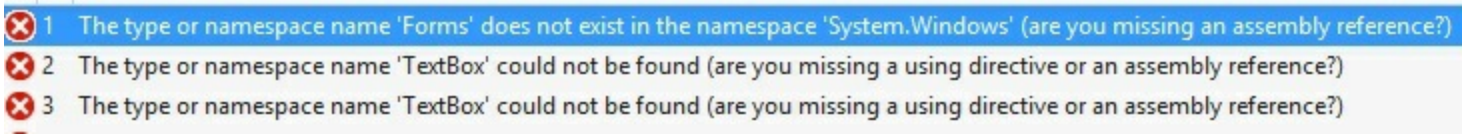
Don't forget to implement `INotifyPropertyChanged`.

Data binding only works when the controls implement that interface. If you leave `: INotifyPropertyChanged` out of the class declaration, your bound controls won't get updated — even if the data object fires PropertyChanged events.

EXERCISE

Finish porting the Go Fish! game to a Windows Store app. You'll need to modify the XAML from earlier in this chapter to add data binding, copy all of the classes and enums from the Go Fish! game in Chapter 8 (or download them from our website), and update the `Player` and `Game` classes.

- **❶ Add the existing class files and change their namespace to match your app.**
  Add these files to your project from the Chapter 8 Go Fish! code: *Values.cs*, *Suits.cs*, *Card.cs*, *Deck.cs*, *CardComparer_bySuit.cs*, *CardComparer_byValue.cs*, *Game.cs*, and *Player.cs*. You can use the Add Existing Item option in the Solution Explorer, but you'll need to **change the namespace** in each of them to match your new project's namespace (just like you did with multipart projects earlier in the book).
  Try building your project. You should get errors in *Game.cs* and *Player.cs* that look like this:



- **❷ Remove all references to WinForms classes and objects; add `using` lines to `Game`.**
  You're not in the WinForms world anymore, so delete `using System.Windows.Forms;` from the top of *Game.cs* and *Player.cs*. You'll also need to remove all mentions of TextBox. You'll need to modify the `Game` class to use `INotifyPropertyChanged` and `ObservableCollection<T>`, so add these `using` lines to the top of *Game.cs*:

```
using System.ComponentModel;
using System.Collections.ObjectModel;
```

- **❸ Add an instance of `Game` as a static resource and set up the data context.**
  Modify your XAML to add an instance of `Game` as a static resource and use it as the data context for the grid that contains the Go Fish! page you built earlier in the chapter. Here's the XAML for the static resource: `<local:Game x:Name="game"/>` — and you're going to need a new constructor because you can only include resources that have parameterless constructors:

```
public Game() {
    PlayerName = "Ed";
    Hand = new ObservableCollection<string>();
    ResetGame();
}
```

- **❹ Add public properties to the `Game` class for data binding.**
  Here are the properties you'll be binding to properties of the controls in the page:

```
public bool GameInProgress { get; private set; }
public bool GameNotStarted { get { return !GameInProgress; } }
public string PlayerName { get; set; }
public ObservableCollection<string> Hand { get; private set; }
public string Books { get { return DescribeBooks(); } }
public string GameProgress { get; private set; }
```

- **❺ Use binding to enable or disable the TextBox, ListBox, and Buttons.**
  You want the "Your Name" TextBox and the "Start the game!" Button to be enabled only when the game is not started, and you want the "Your hand" ListBox and "Ask for a card" Button to be enabled only when the game is in progress. You'll add code to the `Game` class to set the `GameInProgress` property. Have a look at the `GameNotStarted` property. Figure out how it works, then add the following property bindings to the TextBox, ListBox, and two Buttons:

You'll need two of each of these.

```
IsEnabled="{Binding GameInProgress}"   IsEnabled="{Binding GameNotStarted}"
IsEnabled="{Binding GameInProgress}"   IsEnabled="{Binding GameNotStarted}"
```

- **❻ Modify the `Player` class so it tells the `Game` to display the game's progress.**
  The WinForms version of the `Player` class takes a TextBox as a parameter for its constructor. Change that to take a reference to the `Game` class and store it in a private field. (Look at the `StartGame()` method below to see how this new constructor is used when adding players.) Find the lines that use the TextBox reference and replace them with calls to the `Game` object's `AddProgress()` method.
- **❼ Modify the `Game` class.**
  Change the `PlayOneRound()` method so that it's void instead of returning a Boolean, and have it use the `AddProgress()` method instead of the TextBox to display progress. If a player won, display that progress, reset the game, and return. Otherwise, refresh the `Hand` collection and describe the hands.
  You'll also need to add/update these four methods, and figure out what they do and how they work.

```
public void StartGame() {
    ClearProgress();
    GameInProgress = true;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    Random random = new Random();
    players = new List<Player>();
    players.Add(new Player(PlayerName, random, this));
    players.Add(new Player("Bob", random, this));
    players.Add(new Player("Joe", random, this));
    Deal();
    players[0].SortHand();
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
    OnPropertyChanged("Books");
}

  public void ClearProgress() {
      GameProgress = String.Empty;
      OnPropertyChanged("GameProgress");
  }

public void AddProgress(string progress)
{
    GameProgress = progress +
           Environment.NewLine +
           GameProgress;
    OnPropertyChanged("GameProgress");
}
```

> You'll **also** need to implement the `INotifyPropertyChanged` interface and add the same `OnPropertyChanged()` method that you used in the `MenuMaker` class. The updated methods use it, and your modified `PullOutBooks()` method will also use it.

```
public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}
```

EXERCISE SOLUTION

Here's all of the code-behind that you had to write:

```
private void startButton_Click(object sender, RoutedEventArgs e) {
    game.StartGame();
}
private void askForACard_Click(object sender, RoutedEventArgs e) {
```

```
        if (cards.SelectedIndex >= 0)
            game.PlayOneRound(cards.SelectedIndex);
    }
    private void cards_DoubleTapped(object sender, DoubleTappedRoutedEventArgs e) {
        if (cards.SelectedIndex >= 0)
            game.PlayOneRound(cards.SelectedIndex);
    }
```

These are the changes needed for the `Player` class:

```
    class Player {
        private string name;
        public string Name { get { return name; } }
        private Random random;
        private Deck cards;
        private Game game;
        public Player(String name, Random random, Game game) {
            this.name = name;
            this.random = random;
            this.game = game;
            this.cards = new Deck(new Card[] { });
            game.AddProgress(name + " has just joined the game");
        }
        public Deck DoYouHaveAny(Values value)
        {
            Deck cardsIHave = cards.PullOutValues(value);
            game.AddProgress(Name + " has " + cardsIHave.Count + " " + Card.Plural(value));
            return cardsIHave;
        }
        public void AskForACard(List<Player> players, int myIndex, Deck stock, Values value) {
            game.AddProgress(Name + " asks if anyone has a " + value);
            int totalCardsGiven = 0;
            for (int i = 0; i < players.Count; i++) {
                if (i != myIndex) {
                    Player player = players[i];
                    Deck CardsGiven = player.DoYouHaveAny(value);
                    totalCardsGiven += CardsGiven.Count;
                    while (CardsGiven.Count > 0)
                        cards.Add(CardsGiven.Deal());
                    }
            }
            if (totalCardsGiven == 0) {
                game.AddProgress(Name + " must draw from the stock.");
                cards.Add(stock.Deal());
            }
        }
        // ... the rest of the Player class is the same ...
```

These are the changes needed for the XAML:

```
<Grid Grid.Row="1" Margin="120,0,60,60" DataContext="{StaticResource ResourceKey=game}" >
    <TextBlock Text="Your Name" Margin="0,0,0,20"
        Style="{StaticResource SubheaderTextBlockStyle}"/>
    <StackPanel Orientation="Horizontal" Grid.Row="1">
        <TextBox x:Name="playerName" FontSize="24" Width="500" MinWidth="300"
            Text="{Binding PlayerName, Mode=TwoWay}" IsEnabled="{Binding GameNotStarted}" />
        <Button x:Name="startButton" Margin="20,0" IsEnabled="{Binding GameNotStarted}"
            Content="Start the game!" Click="startButton_Click" />
    </StackPanel>
    <TextBlock Text="Game progress"
        Style="{StaticResource SubheaderTextBlockStyle}" Margin="0,20,0,20" Grid.Row="2"/>
    <ScrollViewer Grid.Row="3" FontSize="24" Background="White" Foreground="Black"
        Content="{Binding GameProgress}" />
    <TextBlock Text="Books" Style="{StaticResource SubheaderTextBlockStyle}"
        Margin="0,20,0,20" Grid.Row="4"/>
    <ScrollViewer FontSize="24" Background="White" Foreground="Black"
        Grid.Row="5" Grid.RowSpan="2" Content="{Binding Books}" />
    <TextBlock Text="Your hand" Style="{StaticResource SubheaderTextBlockStyle}"
        Grid.Row="0" Grid.Column="2" Margin="0,0,0,20"/>
    <ListBox Background="White" FontSize="24" Height="Auto" Margin="0,0,0,20"
        x:Name="cards" Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"
        ItemsSource="{Binding Hand}" IsEnabled="{Binding GameInProgress}"
        DoubleTapped="cards_DoubleTapped" />
    <Button x:Name="askForACard" Content="Ask for a card" HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch" Grid.Row="6" Grid.Column="2"
        Click="askForACard_Click" IsEnabled="{Binding GameInProgress}" />
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="5*"/>
        <ColumnDefinition Width="40"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto" MinHeight="150" />
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
</Grid>
```

The data context for the grid is the Game class, since all of the binding is to properties on that class.

The TextBox has a two-way binding to PlayerName.

Here's the Click event handler for the Start button.

The Game Progress and Books ScrollViewers bind to the Progress and Books properties.

The IsEnabled property enables or disables the control. It's a Boolean property, so you can bind it to a Boolean property to turn the control on or off based on that property.

EXERCISE SOLUTION

Here's everything that changed in the `Game` class, including the code we gave you with the instructions.

```
using System.ComponentModel;
using System.Collections.ObjectModel;

class Game : INotifyPropertyChanged {
    private List<Player> players;
    private Dictionary<Values, Player> books;
    private Deck stock;

    public bool GameInProgress { get; private set; }
    public bool GameNotStarted { get { return !GameInProgress; } }
    public string PlayerName { get; set; }
    public ObservableCollection<string> Hand { get; private set; }
    public string Books { get { return DescribeBooks(); } }
    public string GameProgress { get; private set; }

    public Game() {
        PlayerName = "Ed";
        Hand = new ObservableCollection<string>();
        ResetGame();
    }

    public void AddProgress(string progress) {
        GameProgress = progress + Environment.NewLine + GameProgress;
        OnPropertyChanged("GameProgress");
    }

    public void ClearProgress() {
        GameProgress = String.Empty;
        OnPropertyChanged("GameProgress");
    }

    public void StartGame() {
        ClearProgress();
        GameInProgress = true;
        OnPropertyChanged("GameInProgress");
        OnPropertyChanged("GameNotStarted");
        Random random = new Random();
        players = new List<Player>();
        players.Add(new Player(PlayerName, random, this));
        players.Add(new Player("Bob", random, this));
        players.Add(new Player("Joe", random, this));
        Deal();
        players[0].SortHand();
        Hand.Clear();
        foreach (String cardName in GetPlayerCardNames())
            Hand.Add(cardName);
        if (!GameInProgress)
            AddProgress(DescribePlayerHands());
        OnPropertyChanged("Books");
    }
```

You need these lines for INotifyPropertyChanged and ObservableCollection.

These properties are used by the XAML data binding.

Here's the new Game constructor. We only create one collection and just clear it when the game is reset. If we created a new object, the form would lose its reference to it, and the updates would stop.

These methods make the game progress data binding work. New lines are added to the top so the old activity scrolls off the bottom of the ScrollViewer.

Here's the StartGame() method that we gave you. It clears the progress, creates the players, deals the cards, and then updates the progress and books.

Every program you've written in the book so far can be adapted or rewritten as a Windows Store app using XAML. But there are so many ways to write them, and that's especially true when you're using XAML! That's why we gave you so much of the code for this exercise.

This used to return a Boolean value so the form could update its progress. Now it just needs to call AddProgress, and data binding will take care of the updating for us.

```
public void PlayOneRound(int selectedPlayerCard) {
    Values cardToAskFor = players[0].Peek(selectedPlayerCard).Value;
    for (int i = 0; i < players.Count; i++) {
        if (i == 0)
            players[0].AskForACard(players, 0, stock, cardToAskFor);
        else
            players[i].AskForACard(players, i, stock);
        if (PullOutBooks(players[i])) {
            AddProgress(players[i].Name + " drew a new hand");
            int card = 1;
            while (card <= 5 && stock.Count > 0) {
                players[i].TakeCard(stock.Deal());
                card++;
            }
        }
        OnPropertyChanged("Books");
        players[0].SortHand();
        if (stock.Count == 0) {
            AddProgress("The stock is out of cards. Game over!");
            AddProgress("The winner is... " + GetWinnerName());
            ResetGame();
            return;
        }
    }
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
}
```

The books changed, and the form needs to know about the change so it can refresh its ScrollViewer.

Here are the modifications to the PlayOneRound() method that update the progress when the game is over, or update the hand and the books if it's not.

This is the ResetGame() method from the instructions. It clears the books, stock, and hand.

```
public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}

public event PropertyChangedEventHandler PropertyChanged;
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}

// ... the rest of the Game class is the same ...
```

This is the standard PropertyChanged event pattern from earlier in the chapter.

Chapter 11. Async, Await, and Data Contract Serialization: Pardon the interruption

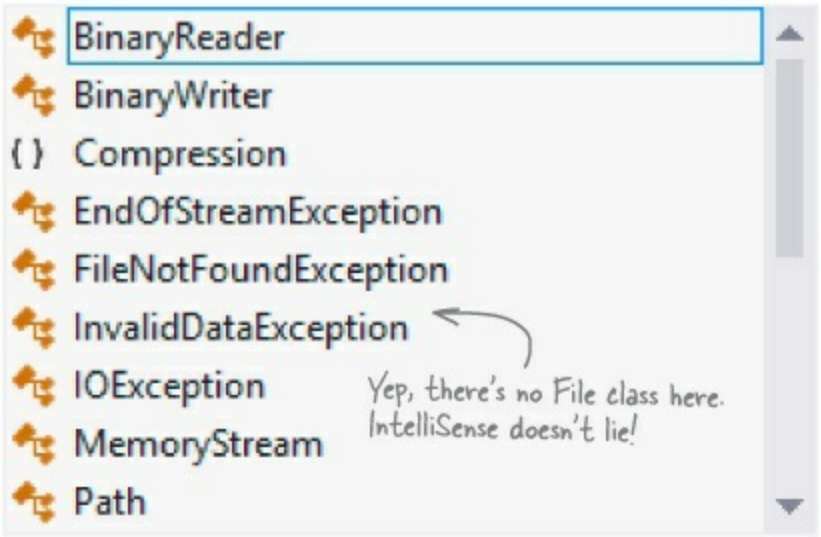
Nobody likes to be kept waiting...especially not users.

Computers are good at doing lots of things at once, so there's no reason your apps shouldn't be able to as well. In this chapter, you'll learn how to keep your apps responsive by **building asynchronous methods**. You'll also learn how to use the **built-in file pickers and message dialogs** and **asynchronous file input and output** without freezing up your apps. Combine this with **data contract serialization**, and you've got the makings of a thoroughly modern app.

Brian runs into file trouble

Brian's got his XAML, he's got his data binding, and he's all ready to start porting his Excuse Manager to a Windows Store app. Everything's going great, until...

```
System.IO.
```



Windows Store apps have superior I/O tools.

When you build a Windows Store app, it needs to be responsive, intuitive, and consistent. That's why the .NET Framework for Windows Store Apps includes classes and methods that let you display file dialogs and do file I/O **asynchronously** — which means they don't lock up your app while dialogs are displayed or files are written. And by using **data contracts** for serialization, your apps can write files that are easier to work with, and much clearer to understand.

NOTE

When you see an hourglass, that means you're using a program that's locked up and has become unresponsive...and users hate that! (Don't you?)

Windows Store apps use await to be more responsive

What happens when you call `MessageBox.Show()` from a WinForms program? Everything stops, and your program freezes until the dialog disappears. That's literally the most unresponsive that a program can be! Windows Store apps should always be responsive, even when they're waiting for feedback from a user. But some things — like waiting for a dialog, or reading or writing all the bytes in a file — take a long time. When a method sits there and makes the rest of the program wait for it to complete, programmers call that **blocking**, and it's one of the biggest causes of program unresponsiveness.

Windows Store apps use **the `await` operator and the `async` modifier** to keep from becoming unresponsive during operations that block. You can see how it works by looking at how Windows Apps pop up dialogs without blocking the app by using the `MessageDialog` class:

You create a MessageDialog object the same way that you'd instantiate any other class.

Configure the MessageDialog by giving it a message and adding responses. Each response is a UICommand object.

```
MessageDialog dialog = new MessageDialog("Message");
dialog.Commands.Add(new UICommand("Response #1"));
dialog.Commands.Add(new UICommand("Response #2"));
dialog.Commands.Add(new UICommand("Response #3"));
dialog.DefaultCommandIndex = 1;
UICommand result = await dialog.ShowAsync() as UICommand;
```

The **`await`** operator causes the method that's running this code to stop and wait until the `ShowAsync()` method completes — and that method will block until the user chooses one of the commands. In the meantime, the rest of the program ***will keep responding to other events***. As soon as the `ShowAsync()` method returns, the method that called it will pick up where it left off (although it may wait until after any other events that started up in the meantime have finished).

If your method uses the `await` operator, then it **must be declared with the `async` modifier**:

```
public async void ShowADialog() {
    // ... some code ...
    UICommand result = await dialog.ShowAsync() as UICommand;
    // ... some more code:
}
```

When a method is declared with `async`, you have some options with how you call it. You can call the method as usual. When you do, as soon as it hits the `await` statement it returns, which keeps the blocking call from freezing your app.

DO THIS!

You can see exactly how this works by **creating a new Blank App** and adding the following XAML:

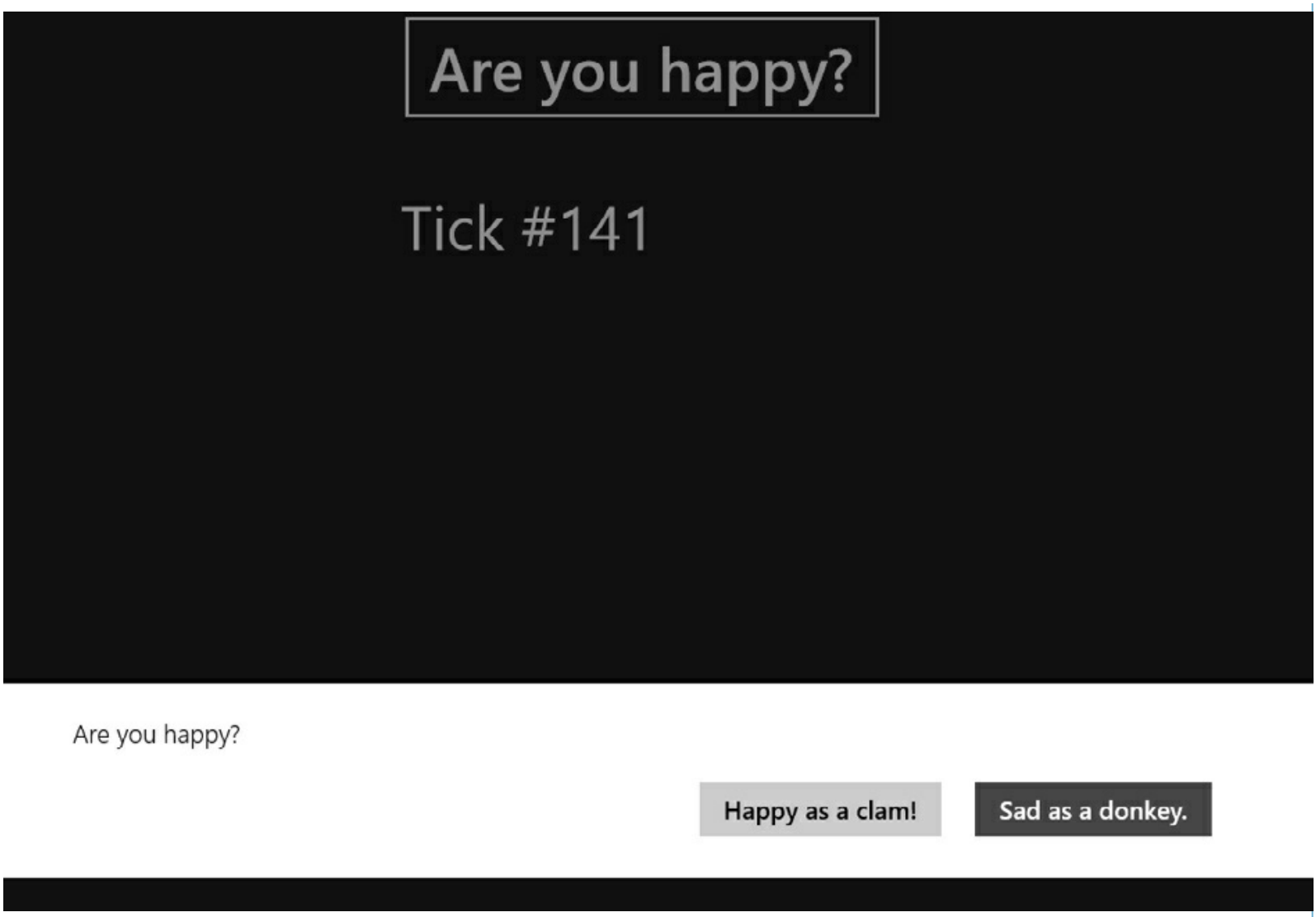
```
<StackPanel VerticalAlignment="Top" HorizontalAlignment="Center">
    <Button Click="Button_Click_1" FontSize="36">Are you happy?</Button>
    <TextBlock x:Name="response" FontSize="36"/>
    <TextBlock x:Name="ticker" FontSize="36"/>
</StackPanel>
```

Here's the code-behind. You'll need to add `using Windows.UI.Popups;` because `MessageDialog` and `UICommand` are in that namespace.

```
DispatcherTimer timer = new DispatcherTimer();
private void Button_Click_1(object sender, RoutedEventArgs e) {
    timer.Tick += timer_Tick;
    timer.Interval = TimeSpan.FromMilliseconds(50);
    timer.Start();
    CheckHappiness();
}
int i = 0;
void timer_Tick(object sender, object e) {
    ticker.Text = "Tick #" + i++;
}
private async void CheckHappiness() {
    MessageDialog dialog = new MessageDialog("Are you happy?");
    dialog.Commands.Add(new UICommand("Happy as a clam!"));
    dialog.Commands.Add(new UICommand("Sad as a donkey."));
    dialog.DefaultCommandIndex = 1;
    UICommand result = await dialog.ShowAsync() as UICommand;
    if (result != null && result.Label == "Happy as a clam!")
        response.Text = "The user is happy";
    else
        response.Text = "The user is sad";
    timer.Stop();
}
```

Try moving the timer.Stop() line here. The timer will stop ticking immediately, because the async method returns as soon as it hits the await operator.

When you run the program, you can see the timer ticking while the dialog is open. Your app remains responsive! It doesn't stop ticking until after you click on one of the dialog options, at which point the method resumes.



Are you happy?

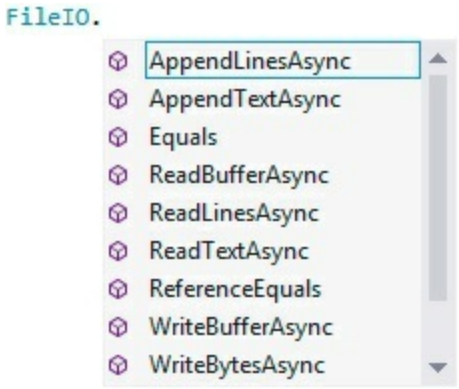
Happy as a clam!

Sad as a donkey.

Use the FileIO class to read and write files

WinForms use the `System.IO.File` class to read and write files, but you've already seen that class doesn't exist in the .NET Framework for Windows Store apps. And that's a good thing! If you use `File.WriteAllText()` to write a giant file that will fill up a big portion of your hard drive, it will block and cause your program to become unresponsive.

Windows Store apps can **use `Windows.Storage` classes to read and write files**. That namespace includes a class called `FileIO`, which has some familiar-looking methods that pop up in its IntelliSense window.



These methods look similar to the ones in the `File` class. The `FileIO` class has `AppendLinesAsync()` and `ReadTextAsync()`, where the `File` class had `AppendLines()` and `ReadText()`. The difference is that each of these methods is declared using the `async` modifier, and uses the `await` operator to do the actual file reading. That lets you write code that can read and write files without blocking.

Use the file pickers to locate file paths

MessageBoxes aren't the only kinds of dialogs that cause your WinForms programs to become unresponsive. File dialogs do exactly the same thing. Windows Store apps have their own file pickers to access files and folders, and they're **asynchronous**, too (so they don't block). Here's how to create and use a `FileOpenPicker` to find a file to open, and `ReadTextAsync()` to read the text from it into a file:

```
FileOpenPicker picker = new FileOpenPicker {
    ViewMode = PickerViewMode.List,
    SuggestedStartLocation = PickerLocationId.DocumentsLibrary
};
picker.FileTypeFilter.Add(".txt");
IStorageFile file = await picker.PickSingleFileAsync();
if (file != null) {
    string fileContents = await FileIO.ReadTextAsync(file);
}
```

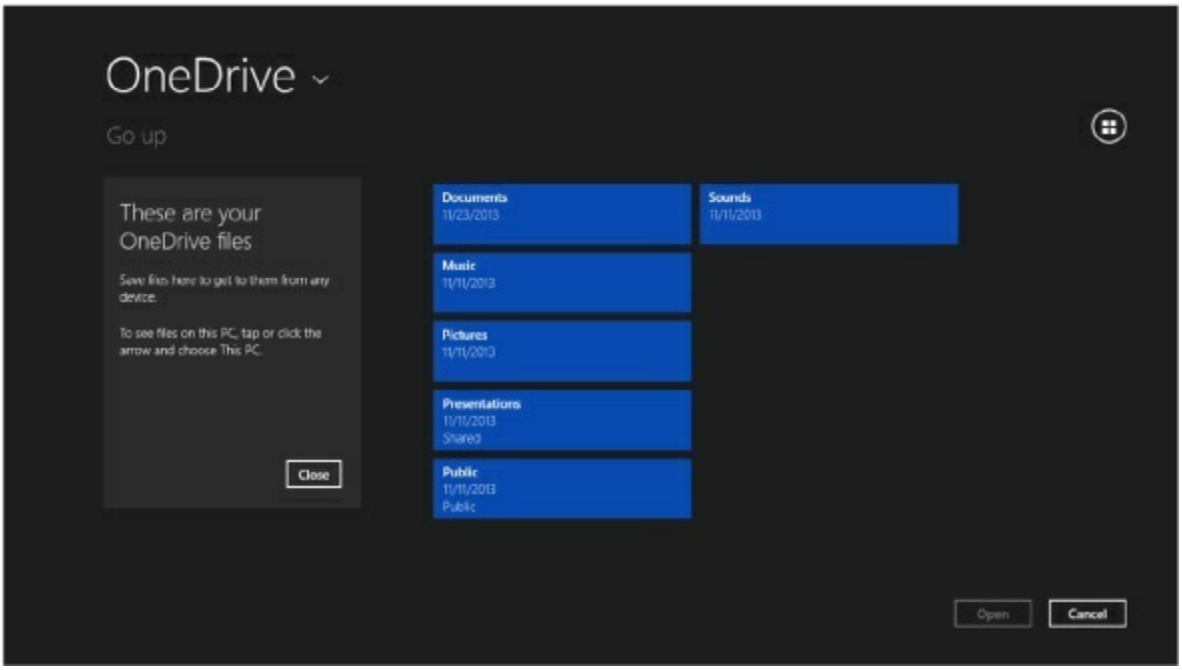
You can configure the properties on the picker using an object initializer. This FileOpenPicker is configured to display files as a list, and start in the user's documents library folder.

This picker has a collection called FileTypeFilter that has the types of files that it can load.

The file picker returns an IStorageFile when you pick a single file. You'll read a lot more about it in a few pages.

You can pass the IStorageFile reference straight into FileIO.ReadTextAsync() to read the contents of the file.

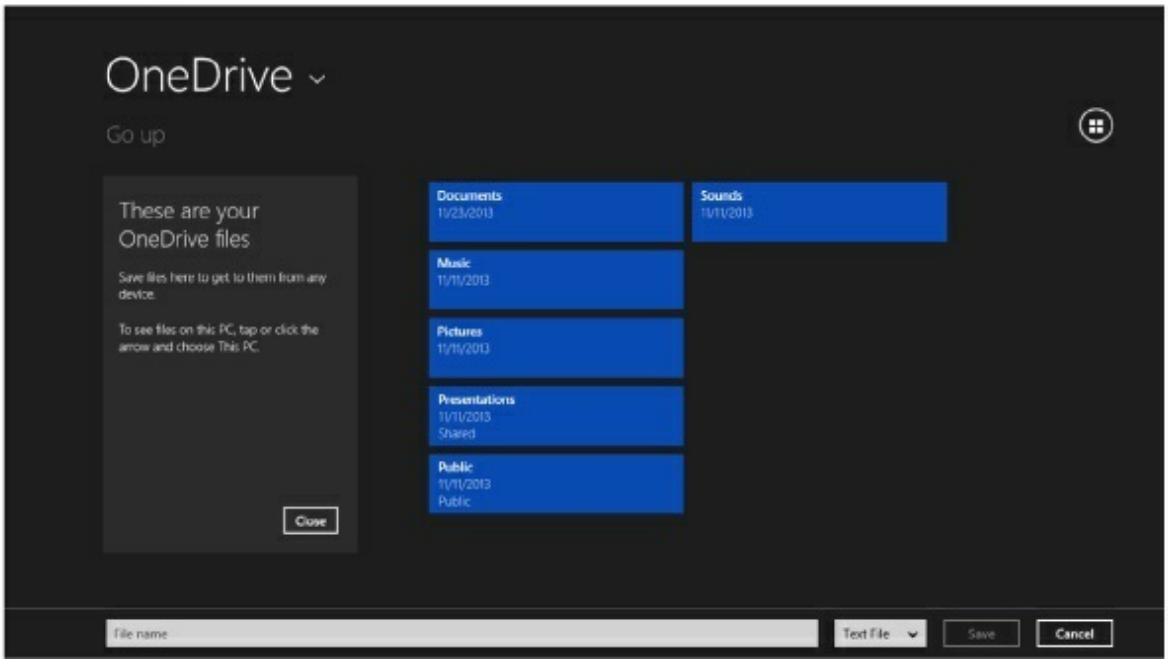
Here's what the FileOpenPicker looks like when it's open.



The `FileSavePicker` lets the user pick a file to save. Here's how it can be used in conjunction with `FileIO.WriteTextAsync()` to write text to a file:

The FileSavePicker returns an IStorageFile, too. It contains all of the information needed to read or write to a file, and can be passed straight to WriteTextAsync().

```
FileSavePicker picker = new FileSavePicker {
    DefaultFileExtension = ".txt",
    SuggestedStartLocation = PickerLocationId.DocumentsLibrary
};
picker.FileTypeChoices.Add("Text File", new List<string>() { ".txt" });
picker.FileTypeChoices.Add("Log File",
                           new List<string>() { ".log", ".dat" });
IStorageFile saveFile = await picker.PickSaveFileAsync();
if (saveFile == null) return;
await FileIO.WriteTextAsync(saveFile, textToWrite);
```



Build a slightly less simple text editor

DO THIS!

Let's rebuild the Simple Text Editor from Chapter 9 as a Windows Store app. You'll use the `FileIO` class, a `FileOpenPicker`, and a `FileSavePicker` to load and save the files. But first you'll build the main page. And since this is a Windows Store app that can open and save files, it should have **an app bar with Open and Save buttons**, so you'll use the IDE to add one.

An AppBar control is a lot like a ScrollViewer or Border, because it can contain another control. It knows how to hide and show itself, and acts just like any other app bar. All you need to do is add it to the `<BottomAppBar>` or `<TopAppBar>` section of a page.

- ❶ Create a new Windows Store Blank App project, and replace *MainPage.xaml* with a new Basic Page. Here's XAML for the page contents:

```
<Grid Grid.Row="1" Margin="120,0,60,60">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <TextBlock x:Name="filename" Margin="10"
               Style="{StaticResource TitleTextBlockStyle}">
        Untitled
    </TextBlock>
    <Border Margin="10" Grid.Row="1">
        <TextBox x:Name="text" AcceptsReturn="True"
             ScrollViewer.VerticalScrollBarVisibility="Visible"
             ScrollViewer.HorizontalScrollBarVisibility="Visible"
             TextChanged="text_TextChanged" />
    </Border>
</Grid>
```

The AcceptsReturn property makes the TextBox accept multiline input.

The TextBox control can display horizontal and vertical scrollbars. These properties turn them on.

  Right-click on `text_TextChanged` and choose Go To Definition F12 from the menu. The IDE will create the `TextChanged` event handler for your TextBox.

- ❷ Use the Document Outline to select the Page (it's named `pageRoot` — or select any control and press Escape a few times). Go to the Properties window, expand the Common section, and find the `BottomAppBar` property:

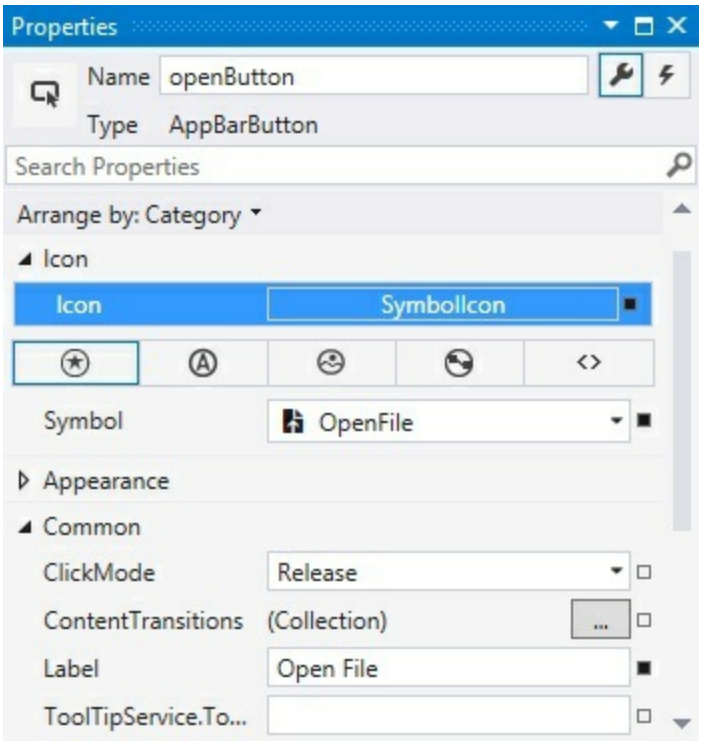
  Click the New button — this will pop up a Select Object window so you can choose what type of object to add. Choose AppBar to add a bottom app bar. The IDE will add this code to your page:

```
<Page.BottomAppBar>
    <AppBar/>
</Page.BottomAppBar>
```

- ❸ The AppBar is a container control. When you display a page's app bar, its AppBar control displays itself and any controls that it contains. But most of the time it's hidden. **Click anywhere in the page outside of the app bar** and you'll see it disappear from the designer — just like it

does in a typical Windows Store app.
You want your AppBar to contain buttons, and you want those buttons to look like normal, everyday app bar buttons. .NET for Windows Store has a control for exactly that: the **AppBarButton** control. But your app bar disappeared when yo clicked outside of it! No problem — just **select the app bar** by either clicking on `<AppBar/>` in the XAML designer or using the Document Outline to select it, and the app bar will appear again.
Next, **drag an** AppBarButton **out of the Toolbox** and onto your app bar. (You can also double-click it in th Toolbox.) The default app bar button has a smiley face icon and the label "AppBarButton," so use the Properties window to change its name, icon and label. You can also just edit the XAML to change the `x:Name`, `Label`, and `Icon` properties:

```
<Page.BottomAppBar>
    <AppBar>
        <AppBarButton x:Name="openButton"
                Label="Open File" Icon="OpenFile"/>
    </AppBar>
</Page.BottomAppBar>
```



- ❹ You also need a Save button, so try dragging another AppBarButton out of the toolbar onto the app bar. Uh-oh — something's not right! You should see two error messages, one for the AppBar telling you that the property 'Content' can only be set once, and one for the AppBarButton telling you that property is set more than once. That's beause when a containr like an AppBar, Button, or Border bas a Content property, that property can only contain a single control. So how do we get two AppBarButton controls into our AppBar?
  You probably already guessed the answer: we'll use a horizontal StackPanel. **Add a StackPanel to the AppBar** with horizontal orientation, and set its horizontal alignment to `Right` (so the buttons are on the right-hand side of the page when the app bar pops up), and put the Open File AppBarButton inside of it. Next, **add a second AppBarButton** to the StackPanel. Give it the Save icon, the label Save, name it saveButton, and set its IsEnabled property to **`false`** so it's disabled when the app first starts. Finally, **add Click event handlers** to both app bar buttons (call them **`openButton_Click()`** and `saveButton_Click()`). Here's the complete app bar XAML:

```
<Page.BottomAppBar>
    <AppBar>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
            <AppBarButton x:Name="openButton" Click="openButton_Click"
                          Label="Open File" Icon="OpenFile"/>
            <AppBarButton x:Name="saveButton" IsEnabled="false"
                          Label="Save" Icon="Save" Click="saveButton_Click"/>
        </StackPanel>
    </AppBar>
</Page.BottomAppBar>
```

Now your app has a working app bar!

- ❺ Here's the code-behind for the entire program. It uses the `TextBox.Text` property to modify the text in the textbox. We're modifying a property on the object instead of using data binding in order to keep the code in this program as similar as possible to the Simple Text Editor in Chapter 9. That will give you a reference point for comparison if you want to flip back and forth to see how things change between WinForms and Windows Store apps. You'll also need these `using` statements at the top of the file:

```
using Windows.System;
using Windows.Storage;
using Windows.Storage.Pickers;
using Windows.UI.Popups;
```

Here's the rest of the code. It should all go into the `MainPage` class.

```
bool textChanged = false;
bool loading = false;
IStorageFile saveFile = null;
```

You'll need these three fields. The booleans are used to add the * to the end of the filename. The IStorageFile keeps track of the file being saved so it doesn't have to keep displaying the save file picker.

When you have an `await` in your method, you must have an `async` in the method's declaration.

```
private async void openButton_Click(object sender, RoutedEventArgs e) {
    if (textChanged) {
        MessageDialog overwriteDialog = new MessageDialog(
           "You have unsaved changes. Are you sure you want to load a new file?");
        overwriteDialog.Commands.Add(new UICommand("Yes"));
        overwriteDialog.Commands.Add(new UICommand("No"));
        overwriteDialog.DefaultCommandIndex = 1;
        UICommand result = await overwriteDialog.ShowAsync() as UICommand;
        if (result != null && result.Label == "No")
            return;
    }
    OpenFile();
}
```

The Open button displays a dialog if there are unsaved changes. If the user confirms, then it calls OpenFile() to display a picker and open the file.

The Save button just calls the SaveFile() method.

```
private void saveButton_Click(object sender, RoutedEventArgs e) {
    SaveFile();
}

private void text_TextChanged(object sender, TextChangedEventArgs e) {
    if (loading) {
        loading = false;
        return;
    }
    if (!textChanged) {
        filename.Text += "*";
        saveButton.IsEnabled = true;
        textChanged = true;
    }
}
```

Once the text changes, a * should be added to the end of the filename—but it should only be added once. The textChanged field keeps track if the text has changed.

The loading field keeps it from adding that * immediately after it's loaded (because the text changes, which triggers the event). See if you can figure out how it works.

```
private async void OpenFile() {
    FileOpenPicker picker = new FileOpenPicker {
        ViewMode = PickerViewMode.List,
        SuggestedStartLocation = PickerLocationId.DocumentsLibrary
    };
    picker.FileTypeFilter.Add(".txt");
    picker.FileTypeFilter.Add(".xml");
    picker.FileTypeFilter.Add(".xaml");
    IStorageFile file = await picker.PickSingleFileAsync();
    if (file != null) {
        string fileContents = await FileIO.ReadTextAsync(file);
        loading = true;
        text.Text = fileContents;
        textChanged = false;
        filename.Text = file.Name;
        saveFile = file;
    }
}
private async void SaveFile() {
    if (saveFile == null) {
        FileSavePicker picker = new FileSavePicker {
            DefaultFileExtension = ".txt",
            SuggestedStartLocation = PickerLocationId.DocumentsLibrary
        };
        picker.FileTypeChoices.Add("Text File", new List<string>() { ".txt" });
        picker.FileTypeChoices.Add("XML File", new List<string>() { ".xml", ".xaml" });
        saveFile = await picker.PickSaveFileAsync();
        if (saveFile == null) return;
    }
    await FileIO.WriteTextAsync(saveFile, text.Text);
    await new MessageDialog("Wrote " + saveFile.Name).ShowAsync();
    textChanged = false;
    filename.Text = saveFile.Name;
}
```

The OpenFile() and SaveFile() methods are really similar to the code on the previous page. They display the picker, then use the FileIO methods to load or save the file.

You can hold the Windows key and press Z to show the app bar for the current app.

You're all done. Fire it up!

Simple Text Editor

emailToCaptainAmazing.txt

To: CaptainAmazing@objectville.net
From: Commissioner@objectiville.net
Subject: Can you save the day... again?

We've discovered the Swindler's plan:
The plan -> How I'll defeat Captain Amazing
The plan -> Another genius secret plan by The Swindler
The plan -> I'll create an army of clones and unleash them upon the citizens of Objectville.
The plan -> Clone #0 attacks the mall
The plan -> Clone #1 attacks downtown
The plan -> Clone #2 attacks the mall
The plan -> Clone #3 attacks downtown
The plan -> Clone #4 attacks the mall
The plan -> Clone #5 attacks downtown
The plan -> Clone #6 attacks the mall

Can you help us?

You can bring up the app bar by touching or clicking, then dragging up from the bottom of the page. You can also use the Windows–Z keyboard shortcut.

Open File Save

There is! It's called data contract serialization.

Writing text files is great, because you can just open up a file in Notepad and see what's in it. But text files are also pretty lousy, because you need to write a lot of code to parse your data.

Binary serialization with a BinaryFormatter is great because it's so convenient. But it's pretty lousy in its own way! Binary files are **fragile**. Make one tiny change to your class, and suddenly you can't load any of your files anymore! And you've already seen the mess that appears when you open up binary files in Notepad. Good luck getting a human to read or edit a binary file.

Data contract serialization is the best of both worlds. It's true serialization, so entire object graphs are automatically written out for you. But it generates XML files, which turn out to be really easy to read and can even be edited by hand (especially if you're used to working with XAML!).

NOTE

When you use binary serialization, you're writing "pure"(-ish) data: actual bytes in memory get glued together and written to a file, along with just enough information for the binary formatter to figure out which bytes go with which class members in the object graph. One little change to just one class, and suddenly none of the bytes line up anymore, and when you try to deserialize you'll get an error.

A data contract is an abstract definition of your object's data

A data contract is a **formal agreement** that's attached to your class. The contract uses the `[DataContract]` and `[DataMember]` attributes to define exactly what data gets read or written during serialization.

If you want to serialize instances of a class, you can set up a data contract for it by adding the `[DataContract]` attribute to the top, and `[DataMember]` attributes to each class member to be serialized. Here's a simple `Guy` class with a data contract:

```
using System.Runtime.Serialization;

[DataContract]
class Guy {

    [DataMember]
    public string Name { get; private set; }

    [DataMember]
    public int Age { get; private set; }

    [DataMember]
    public decimal Cash { get; private set; }

    public Guy(string name, int age, decimal cash) {
        Name = name; Age = age; Cash = cash;
    }
}
```

The [DataContract] and [DataMember] attributes are in the System.Runtime.Serialization namespace.

The [DataContract] attribute establishes the data contract for this class.

Every class member that needs to be saved or retrieved during serialization is added to the contract with a [DataMember] attribute.

In the XML snippet for <Guy> below, xmlns is called an attribute, not a property. In your XAML files you'll find tags with attributes like Fill, Text, and x:Name. The designer in the IDE calls them properties because they're used to define properties on objects.

Data contract serialization uses XML files

Luckily, you already know a lot about XML files, because XAML is an XML-based language. All XML files use opening tags, closing tags, and attributes to define data. Each member gets a name, but the contract itself also needs a name — or, more specifically, **a unique namespace** — because the serializer needs to be able to distinguish the data files for a contract from other XML files. Here's the XML file that's created when the `Guy` class on this page is serialized. As usual, we added spaces and line breaks to make it easier to read:

```
<Guy xmlns="http://schemas.datacontract.org/2004/07/XamlGuySerializer"
     xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
   <Age>37</Age>
   <Cash>164.38</Cash>
   <Name>Joe</Name>
</Guy>
```

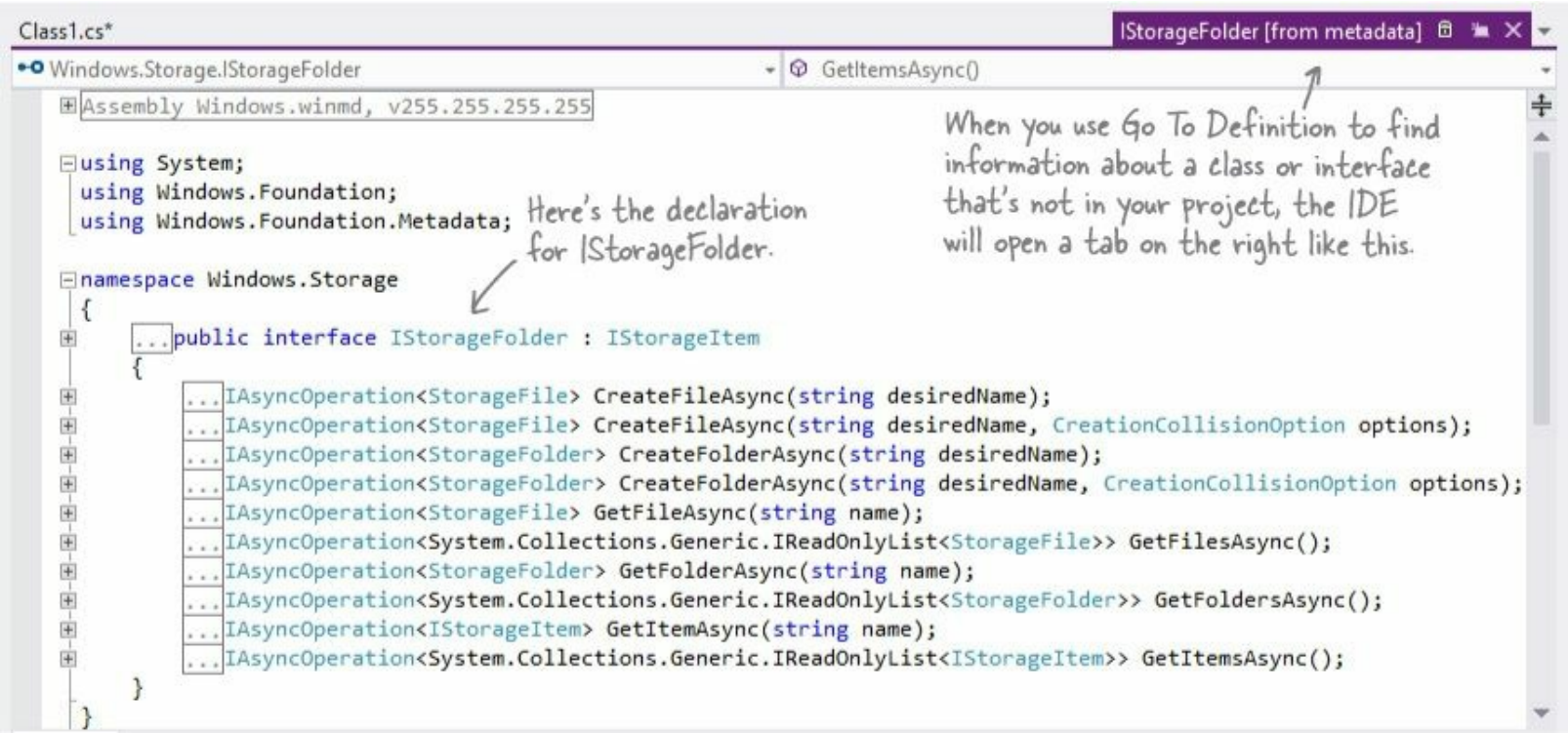
Each of the data members gets its own tag. This is way more readable than binary files!

We named our project XamlGuySerializer, and that was turned into the namespace for the contract.

Use async methods to find and open files

Data contract serialization works a lot like binary serialization. You need to open a file, create a stream for reading or writing, and then call methods to read or write objects. But there are differences, too: Windows Store apps have async methods for opening files. They're based around the `IStorageFile` and `IStorageFolder` interfaces. You can use the IDE to explore these interfaces and discover their members.

Go to any line in any method and type **`Windows.Storage.IStorageFolder`**, then right-click on `IStorageFolder` and choose Go To Definition (F12) to see the definition in the IDE:



Each `IStorageFolder` object represents a folder in the filesystem, with methods to work with its files, including:

- `CreateFileAsync()` is an async method to create a file in the folder.
- `CreateFolderAsync()` is an async method to create a subfolder.
- `GetFileAsync()` gets a file in the folder and returns an `IStorageFile` object.
- `GetFolderAsync()` gets a subfolder and returns another `IStorageFolder` object.
- `GetItemAsync()` gets either a file or a folder, and returns an `IStorageItem` object.
- `GetFilesAsync()`, `GetFoldersAsync()`, and `GetItemsAsync()` return collections of items — these methods return collections of type **`IReadOnlyList`**, a very simple kind of collection that lets you get items by index but doesn't have methods to add, sort, or compare.

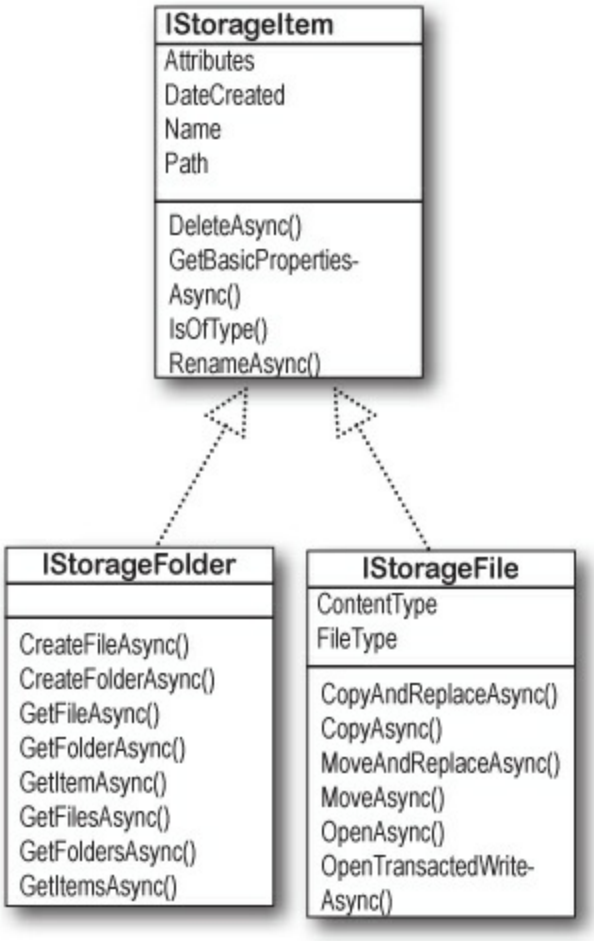
WINDOWS STORE APPS PROTECT YOUR FILESYSTEM

Flip back to the first code sample in Chapter 9. We warned you that it's probably not a good idea to write to the C:\ folder, so hopefully you picked a safe folder to write to. Hopefully. It's really easy for Windows Desktop programs to damage important system files. That's one reason that every Windows Store app gets its own folder to store its files where it's safe to read and write files.

The `Windows.Storage` namespace has two additional interfaces to help you manage items in your filesystem. The `IStorageFile` interface and the objects that implement it (of course!) move, copy,

and open files. And if you look closely at the declaration for `IStorageFolder`, you'll see that it extends the `IStorageItem` interface. `IStorageFile` extends the same interface, which makes sense if you think about the operations that apply to both files and folders: deleting, renaming, and getting the name, creation date, path, and attributes.

Every Windows Store app has a local folder where it's safe to read and write files, which you can access using an `IStorageFolder` called `ApplicationData.Current.LocalFolder`. Then you can **use an `IStorageFile` object to open files for reading and writing** by calling its `OpenAsync()` method (which returns an `IRandomAccessStream`).



Once you have a data contract and a stream, you just need a new `DataContractSerializer`, and you can read and write objects to XML files:

```
using Windows.Storage;
using Windows.Storage.Streams;
using System.Runtime.Serialization;
```

You'll need these using statements.

```
Guy joe = new Guy("Joe", 37, 164.38M);
```

Here's a Guy with a data contract from the previous page.

```
DataContractSerializer serializer =
       new DataContractSerializer(typeof(Guy));

IStorageFolder localFolder =
       ApplicationData.Current.LocalFolder;
```

The data contract serializer needs to know what type it's serializing. Here's how you tell it to serialize Guy objects and their graphs.

```
IStorageFile guyFile = await localFolder.CreateFileAsync("Joe.xml",
                                CreationCollisionOption.ReplaceExisting);

using (IRandomAccessStream stream =
          await guyFile.OpenAsync(FileAccessMode.ReadWrite))

using (Stream outputStream = stream.AsStreamForWrite()) {

          serializer.WriteObject(outputStream, joe);

}
Guy copyOfJoe;

using (IRandomAccessStream stream =
       await guyFile.OpenAsync(FileAccessMode.ReadWrite))

using (Stream inputStream = stream.AsStreamForRead()) {

    copyOfJoe = serializer.ReadObject(inputStream) as Guy;

}
```

You can pass the CreateFileAsync() a filename and a parameter to replace, open, fail, or generate a unique name if the file already exists.

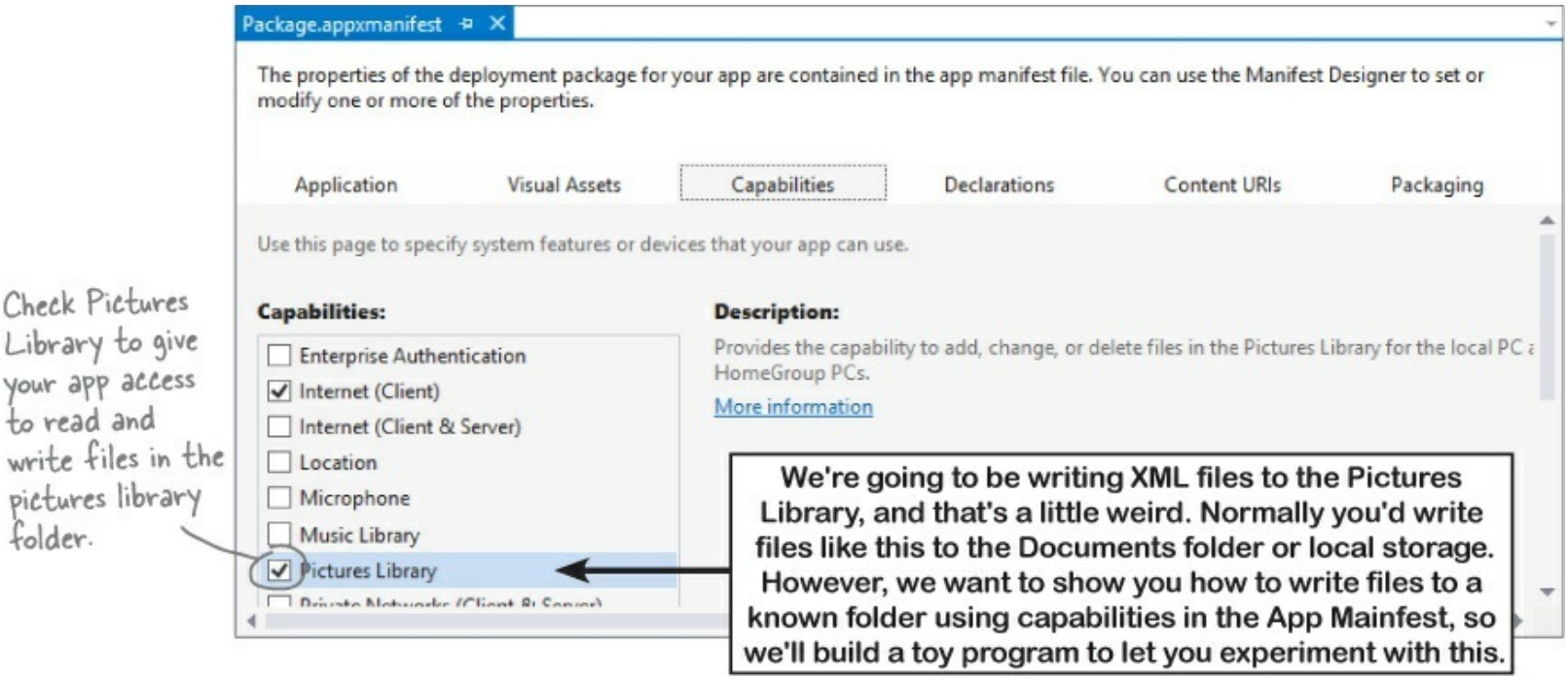
Now that you have input and output streams, you can serialize your objects.

KnownFolders helps you access high-profile folders

The `Windows.Storage` namespace includes the `KnownFolders` class, which has properties to help you access the documents library, music library, or other standard folder for a typical Windows account. `KnownFolders.PicturesLibrary` is a `StorageFolder` object (which implements `IStorageFolder`) that you can use to access the current user's pictures library. It also has properties for the other known folders, including the music and video libraries, removable and media server devices, camera roll, and home group.

But there's a catch. Windows Store apps are free to read and write to the local storage folder. But if you want your app to write to another folder, you'll need to give it special permission by **adding capabilities to the package manifest**. When you explicitly allow your app to read and write to the local folder, anyone who installs it from the Windows Store can see that it has this capability.

To add the Pictures Library capability to your app, **double-click on *Package.appxmanifest*** in the Solution Explorer, click on the Capabilities tab, and check Pictures Library.



WATCH IT!

Your apps should always use a FolderPicker to access folders unless you have a *really good reason* not to.

Technically, you can manually edit the App Manifest and add the Documents Library capability. If you do that, don't bother trying to get your app certified for the Windows Store — it'll almost certainly fail certification.

Take a close look at the Package.appxmanifest screenshot. Do you notice anything missing? What about a Documents Library capability? That's right — you can't write directly to the Documents library unless you go through a FolderPicker. If your Windows Store app needs to write files, there's a good chance that some of your users will want to write to their OneDrive folders, while others may want to write to the local hard drive. So do your users a favor... use a FolderPicker and give them a choice!

The whole object graph is serialized to XML

When the data contract serializer writes an object, it goes through the entire object graph. Every instance of a class with a data contract is written to the XML output. You can customize the XML output by choosing a namespace and naming members using parameters of the `DataContract` and `DataMember` attributes.

```
[DataContract(Namespace = "http://www.headfirstlabs.com/Chapter11")]
class Guy {
    public Guy(string name, int age, decimal cash){
        Name = name;
        Age = age;
        Cash = cash;
        TrumpCard = Card.RandomCard();
    }

    [DataMember]
    public string Name { get; private set; }

    [DataMember]
    public int Age { get; private set; }

    [DataMember]
    public decimal Cash { get; private set; }

    [DataMember(Name = "MyCard")]
    public Card TrumpCard { get; set; }

    public override string ToString() {
        return String.Format("My name is {0}, I'm {1}, I have {2} bucks, "
            + "and my trump card is {3}", Name, Age, Cash, TrumpCard);
    }
}
```

Here's the XML for the serialized `Guy`:

```
<Guy
    xmlns="http://www.headfirstlabs.com/Chapter11"
    xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
    <Age>37</Age>
    <MyCard>
        <Suit>Hearts</Suit>
        <Value>Three</Value>
    </MyCard>
    <Cash>176.22</Cash>
    <Name>Joe</Name>
</Guy>
```

The Guy contains a reference to a Card object with a data contract, so it gets included in the XML as a <Card> tag.

```
[DataContract(Namespace = "http://www.headfirstlabs.com/Chapter11")]
class Card {
    [DataMember]
    public Suits Suit { get; set; }

    [DataMember]
    public Values Value { get; set; }

    public Card(Suits suit, Values value) {
        this.Suit = suit;
        this.Value = value;
    }

    private static Random r = new Random();

    public static Card RandomCard() {
        return new Card((Suits)r.Next(4), (Values)r.Next(1, 14));
    }

    public string Name {
        get { return Value.ToString() + " of " + Suit.ToString(); }
    }

    public override string ToString() { return Name; }
}
```

Both contracts are in the same namespace, which becomes the xmlns property of the <Guy> tag in the serialized XML.

Data contract member names **don't** need to match property names. This `Guy` class has a property called `TrumpCard`, but we used the `Name` parameter of the `DataMember` attribute to give it the name `MyCard`. That's what shows up in the serialized XML.

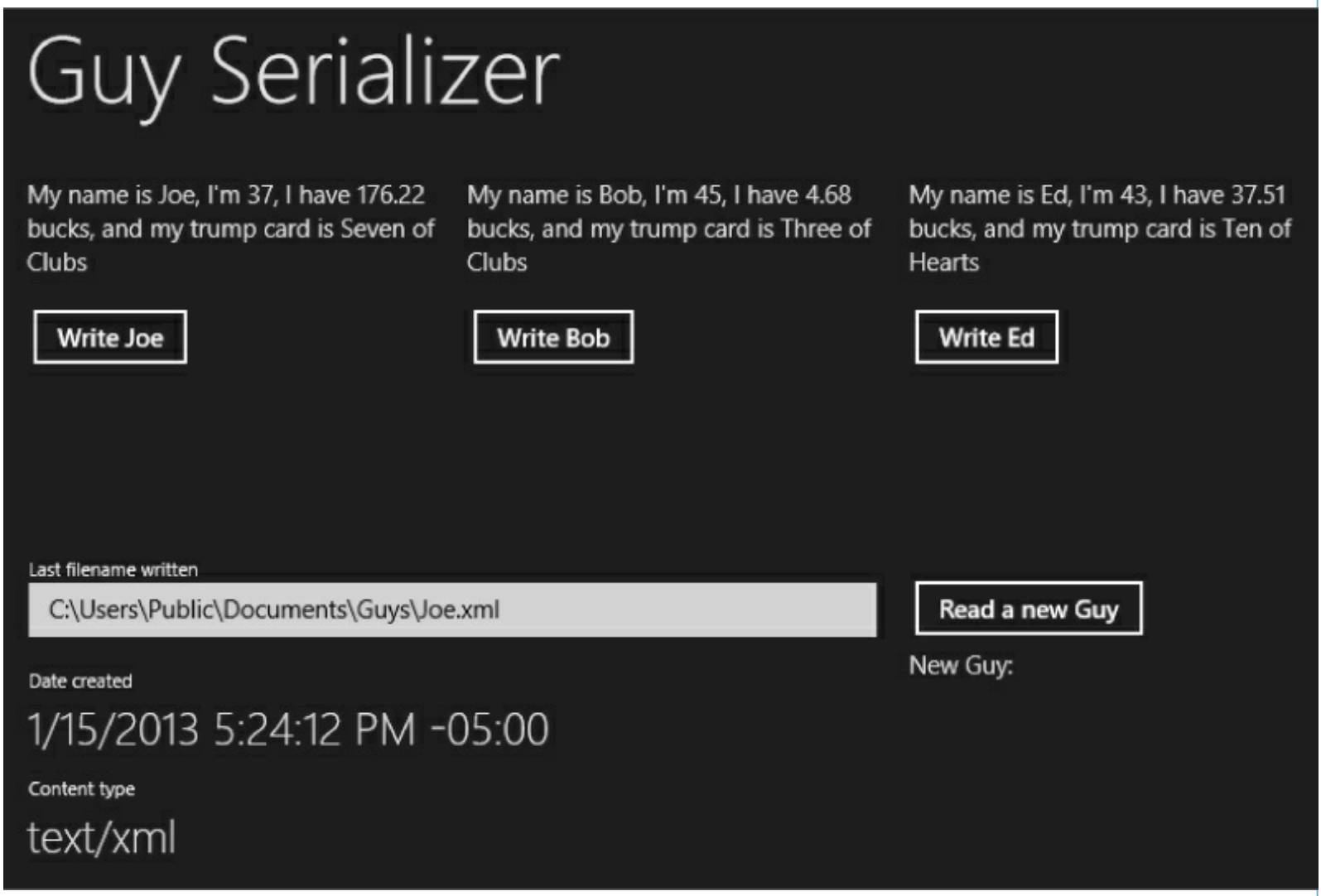
Did you notice that the serialized XML *does not contain the* `Card` *type*? That's because you can add these data contract attributes to any class with compatible members — like the `Suit` and `Value`

properties of the `Card` class, which the serializer knew how to set using values like `Hearts` and `Three` by matching with corresponding enum values.

Stream some Guy objects to XML files

DO THIS!

Here's a project to help you experiment with data contract serialization. **Create a new Windows Store app** and replace the *MainPage.xaml* with a new Basic Bage. Then **open *Package.appxmanifest***, enable access to the Pictures Library. **Add both classes** with the data contracts from the previous page (you'll need `using System.Runtime.Serialization;` in each of them). And add the familiar `Suits` and `Values` enums, too (for the `Card` class). Here's the page you'll build next:



- ❶ Add a static `GuyManager` resource to the page (and set the app name). You'll add the `GuyManager` class on the next page.

```
<Page.Resources>
    <local:GuyManager x:Name="guyManager"/>
    <x:String x:Key="AppName">Guy Serializer</x:String>
</Page.Resources>
```

You can add an empty GuyManager class now to get rid of the IDE error for this tag—you'll fill it in on the next page. Don't forget to rebuild the solution after you add the empty class to get rid of any error messages in the designer.

- ❷ Here's the XAML to add to the page.

```
<Grid Grid.Row="1" DataContext="{StaticResource guyManager}" Margin="120,0">
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

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

    <StackPanel>
        <TextBlock Text="{Binding Joe}" Style="{StaticResource CaptionTextBlockStyle}"
            Margin="0,0,0,20"/>
        <Button x:Name="WriteJoe" Content="Write Joe" Click="WriteJoe_Click"/>
    </StackPanel>

    <StackPanel Grid.Column="1">
        <TextBlock Text="{Binding Bob}" Style="{StaticResource CaptionTextBlockStyle}"
            Margin="0,0,0,20"/>
        <Button x:Name="WriteBob" Content="Write Bob" Click="WriteBob_Click"/>
    </StackPanel>

    <StackPanel Grid.Column="2">
        <TextBlock Text="{Binding Ed}" Style="{StaticResource CaptionTextBlockStyle}"
            Margin="0,0,0,20"/>
        <Button x:Name="WriteEd" Content="Write Ed" Click="WriteEd_Click"/>
    </StackPanel>

    <StackPanel Grid.Row="1" Grid.ColumnSpan="2" Margin="0,0,20,0">
        <TextBlock>Last filename written</TextBlock>
        <TextBox Text="{Binding Path, Mode=TwoWay}" Margin="0,0,0,20"/>
        <TextBlock>Date created</TextBlock>
        <TextBlock Text="{Binding LatestGuyFile.DateCreated}" Margin="0,0,0,20"
            Style="{StaticResource SubheaderTextBlockStyle}"/>
        <TextBlock>Content type</TextBlock>
        <TextBlock Text="{Binding LatestGuyFile.ContentType}"
                Style="{StaticResource SubheaderTextBlockStyle}"/>
    </StackPanel>

    <StackPanel Grid.Row="1" Grid.Column="2">
        <Button x:Name="ReadNewGuy" Content="Read a new Guy" Click="ReadNewGuy_Click"
        Margin="0,10,0,0"/>
        <TextBlock Style="{StaticResource CaptionTextBlockStyle}" Margin="0,0,0,20">
    <Run>New Guy: </Run>
    <Run Text="{Binding NewGuy}"/>
        </TextBlock>
    </StackPanel>
</Grid>
```

The page has three columns and two rows.

The grid's data context is the GuyManager static resource. Hey, did you notice that we left out "resourceKey"? This XAML still works.

The LatestGuyFile property is an IStorageFile, which has a DateCreated property. Try adding a TextBlock that binds to it—instead of displaying a date, it displays a type. Can you figure out how to use a DatePicker and TimePicker to display the file creation time? (Hint: you'll need to bind the controls' Date and Time properties to DateCreated.)

Each column in the top row has a StackPanel with a TextBlock and a Button.

This TextBlock is bound to the Ed property in GuyManager.

The first cell in the bottom row spans two columns. It has several controls bound to properties. Why do you think we used a TextBox for the path?

You can bind a control to a property on an object. `LatestGuyFile` is an `IStorageFile`, and these TextBlock controls are bound to its properties.

We're not done yet — flip the page!

You'll need these using statements for the GuyManager class.

```
using System.ComponentModel;
using Windows.Storage;
using Windows.Storage.Streams;
using System.IO;
using System.Runtime.Serialization;
```

- ❸ Add the `GuyManager` class.

```
class GuyManager : INotifyPropertyChanged
{
    private IStorageFile latestGuyFile;
    public IStorageFile LatestGuyFile { get { return latestGuyFile; } }

    private Guy joe = new Guy("Joe", 37, 176.22M);
    public Guy Joe
    {
        get { return joe; }
    }

    private Guy bob = new Guy("Bob", 45, 4.68M);
    public Guy Bob
    {
        get { return bob; }
    }

    private Guy ed = new Guy("Ed", 43, 37.51M);
    public Guy Ed
    {
        get { return ed; }
    }

    public Guy NewGuy { get; private set; }

    public string Path { get; set; }

    public async void ReadGuyAsync()
    {
        if (String.IsNullOrEmpty(Path))
            return;
        latestGuyFile = await StorageFile.GetFileFromPathAsync(Path);

        using (IRandomAccessStream stream =
                    await latestGuyFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead())
        {
            DataContractSerializer serializer = new DataContractSerializer(typeof(Guy));
            NewGuy = serializer.ReadObject(inputStream) as Guy;
        }
        OnPropertyChanged("NewGuy");
        OnPropertyChanged("LatestGuyFile");
    }
```

The backing field of this property is set by the ReadGuyAsync() method, and TextBlocks are bound to its DateCreated and ContentType properties.

There are three read-only Guy properties with private backing fields. The XAML has a TextBlock bound to each of them.

A fourth TextBlock is bound to this Guy property, which is set by the ReadGuyAsync() method.

You can use the static `StorageFile.GetFileFromPathAsync()` method to create an `IStorageFile` from a string path.

NOTE

The ReadGuyAsync() method uses the path in the TextBox to set the latestGuyFile IStorgeFile field. It uses the serializer to read the objects from the XML file, then fires off PropertyChanged events for properties that use IStorageFile attributes.

The String class also has a method called `IsNullOrWhiteSpace()` that detects nulls, empty strings, or strings with only whitespace characters. Would it make more sense to use it in the ReadGuyAsync() method instead of `IsNullOrEmpty()`?

This creates a folder called Guys in the pictures library to hold the XML files. If it already exists, the existing folder is opened.

This code creates the XML file, opens a stream, and writes the Guy object graph to it.

```
public async void WriteGuyAsync(Guy guyToWrite)
{
    IStorageFolder guysFolder =
        await KnownFolders.PicturesLibrary.CreateFolderAsync("Guys",
                                                CreationCollisionOption.OpenIfExists);

    latestGuyFile =
            await guysFolder.CreateFileAsync(guyToWrite.Name + ".xml",
                                    CreationCollisionOption.ReplaceExisting);

    using (IRandomAccessStream stream =
                        await latestGuyFile.OpenAsync(FileAccessMode.ReadWrite))
    using (Stream outputStream = stream.AsStreamForWrite())
    {
        DataContractSerializer serializer = new DataContractSerializer(typeof(Guy));
        serializer.WriteObject(outputStream, guyToWrite);
    }

    Path = latestGuyFile.Path;

    OnPropertyChanged("Path");
    OnPropertyChanged("LatestGuyFile");
}

public event PropertyChangedEventHandler PropertyChanged;

private void OnPropertyChanged(string propertyName)
{
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null)
    {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}
}
```

The WriteGuyAsync() method writes a guy to an XML file in a Guys folder inside the documents library. It sets the latestGuyFile IStorageFile field to point to the file it wrote, then it fires off property changed events for the properties that use that field.

Here's the same code you used earlier to implement INotifyPropertyChanged and fire off PropertyChanged events.

- ❹ Here are the event handler methods for *MainPage.xaml.cs*:

```
private void WriteJoe_Click(object sender, RoutedEventArgs e) {
    guyManager.WriteGuyAsync(guyManager.Joe);
}
private void WriteBob_Click(object sender, RoutedEventArgs e) {
    guyManager.WriteGuyAsync(guyManager.Bob);
}
private void WriteEd_Click(object sender, RoutedEventArgs e) {
    guyManager.WriteGuyAsync(guyManager.Ed);
}
```

```
private void ReadNewGuy_Click(object sender, RoutedEventArgs e) {
    guyManager.ReadGuyAsync();
}
```

Keep in mind that this is a toy program to help you learn how file access works in Windows 8. You should always use a file picker unless you have a very specific need to add a specific capability to the App Manifest.

Take your Guy Serializer for a test drive

Use the Guy Serializer to experiment with data contract serialization:

- Write each `Guy` object to the Document Library folder. Click the ReadGuy button to read the guy that was just written. It uses the path in the TextBox to read the file, so try updating that path to read a different guy. Try reading a file that doesn't exist. What happens?
- Open up the Simple Text Editor you built earlier in the chapter. You added XML files as options for the open and save file pickers, so you can use it to edit Guy files. Open one of the Guy files, change it, save it, and read it back into your Guy Serializer. What happens if you add invalid XML? What if you change the card suit or value so it doesn't match a valid enum value?
- Your Simple Text Editor doesn't have a New button that resets it to untitled. Can you figure out how to add one? (You can also just restart it.) Try copying a Guy file, then pasting it into a new XML file in the *Guys* folder. What happens when you try to read it into the Guy Serializer?
- Try adding or removing the `DataMember` names (`[DataMember(Name="...")]`). What does that do to the XML? What happens when you update the contract and then try to load a previously saved XML file? Can you fix the XML file to make it work?
- Try changing the namespace of the `Card` data contract. What happens to the XML?

THERE ARE NO DUMB QUESTIONS

Q: **Q: I didn't set the app capabilities in my Simple Text Editor. Why was it able to write to my documents library?**

A: **A:** When your app uses the File Picker, the user can gain access to files and folders **without** setting app capabilities in the package manifest because the File Picker is *built to keep your filesystem safe*: the pickers won't let you access install folders, local folders, temporary folders, and a lot of other unsafe locations in your filesystem that your app could accidentally damage. You only need to set capabilities if you need to write code to access locations directly.

Q: **Q: Sometimes I make a change in my XAML or my code, and the IDE's designer gives me a message that I need to rebuild. What's going on?**

A: **A:** The XAML designer in the IDE is really clever. It's able to show you an updated page in real time as you make changes to your XAML code. You already know that when the XAML uses static resources, that adds object references to the `Page` class. Well, those objects need to get instantiated in order for them to be displayed in the designer. If you make a change to the class that's being used for a static resource, the designer doesn't get updated until you rebuild that class. That makes sense — the IDE only rebuilds your project when you ask it to, and until you do that it doesn't actually have the compiled code in memory that it needs to instantiate the static resources.

You can use the IDE to see exactly how this works. Open your Guy Serializer and edit the `Guy.ToString()` method to add some extra words to the return value. Then go back to the main page designer. It's still showing the old output. Now choose Rebuild from the Build menu. The designer will update itself as soon as the code finishes rebuilding. Try making another change, but don't rebuild yet. Instead, add another TextBlock that's bound to a `Guy` object. The IDE will use the old version of the object until you rebuild.

Q: **Q: I'm confused about namespaces. How is the namespace in the program different from the one in an XML file?**

A: **A:** Let's take a step back and understand why namespaces are necessary. C#, XML files, the Windows filesystem, and web pages all use different (but often related) naming systems to give each class, XML document, file, or web page its own unique name. So why is this important? Well, let's say back in Chapter 9, you created a class called `KnownFolders` to help Brian keep track of excuse folders. Uh oh! Now you find out that the .NET Framework already has a `KnownFolders` class. No worries. The .NET `KnownFolders` class is in the Windows.Storage namespace, so it can exist happily alongside your class with the same name, and that's called **disambiguation**.

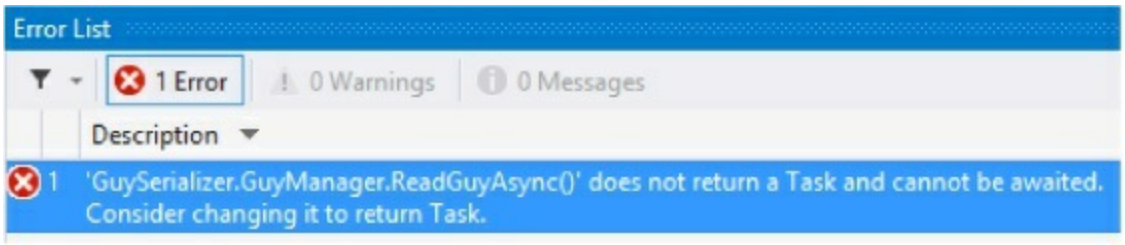
Data contracts also need to disambiguate. You've seen several different versions of a `Guy` class throughout this book. What if you wanted to have two different contracts to serialize different versions of `Guy`? You can put them in different namespaces to disambiguate them. And it makes sense that these namespaces would be separate from the ones for your classes, because you can't really confuse classes and contracts.

Use a Task to call one async method from another

When you mark a method with the `async` modifier, that method can also be awaited by other async methods. But you'll need to make one change to an asynchronous method in order to do that. Try adding this method to your *GuyManager.cs*:

```
private async void MethodThatReadsGuys()
{
    await ReadGuyAsync();
}
```

You'll get an error, with a squiggly underline — and a very useful error message in the Error List window:



The IDE is telling you exactly what you need to do to fix the problem.

In order to make one async method call another, the method being called has to have the return type be the `Task` class (or its subclass, `Task<T>`, if the method needs to return a value). Since `ReadGuyAsync()` has a void return value, all you need to do is **replace `void` with `Task` in the declaration**:

```
public async Task ReadGuyAsync()
{
    // Same as on the previous page
}
```

The recommended naming convention is to add `Async` to the end of the method name for any asynchronous method that should be called with the `await` operator.

Now the method can be called with the `await` operator, and it will act just like any other asynchronous method and return control when it hits an asynchronous operation. If you wanted the method to return a value, you'd make it type `Task<T>`. For example, if you wanted `ReadGuyAsync()` to return the `Guy` object that it read, you would change its return type to `Task<Guy>`.

Yes! The `Task` class represents an asynchronous operation.

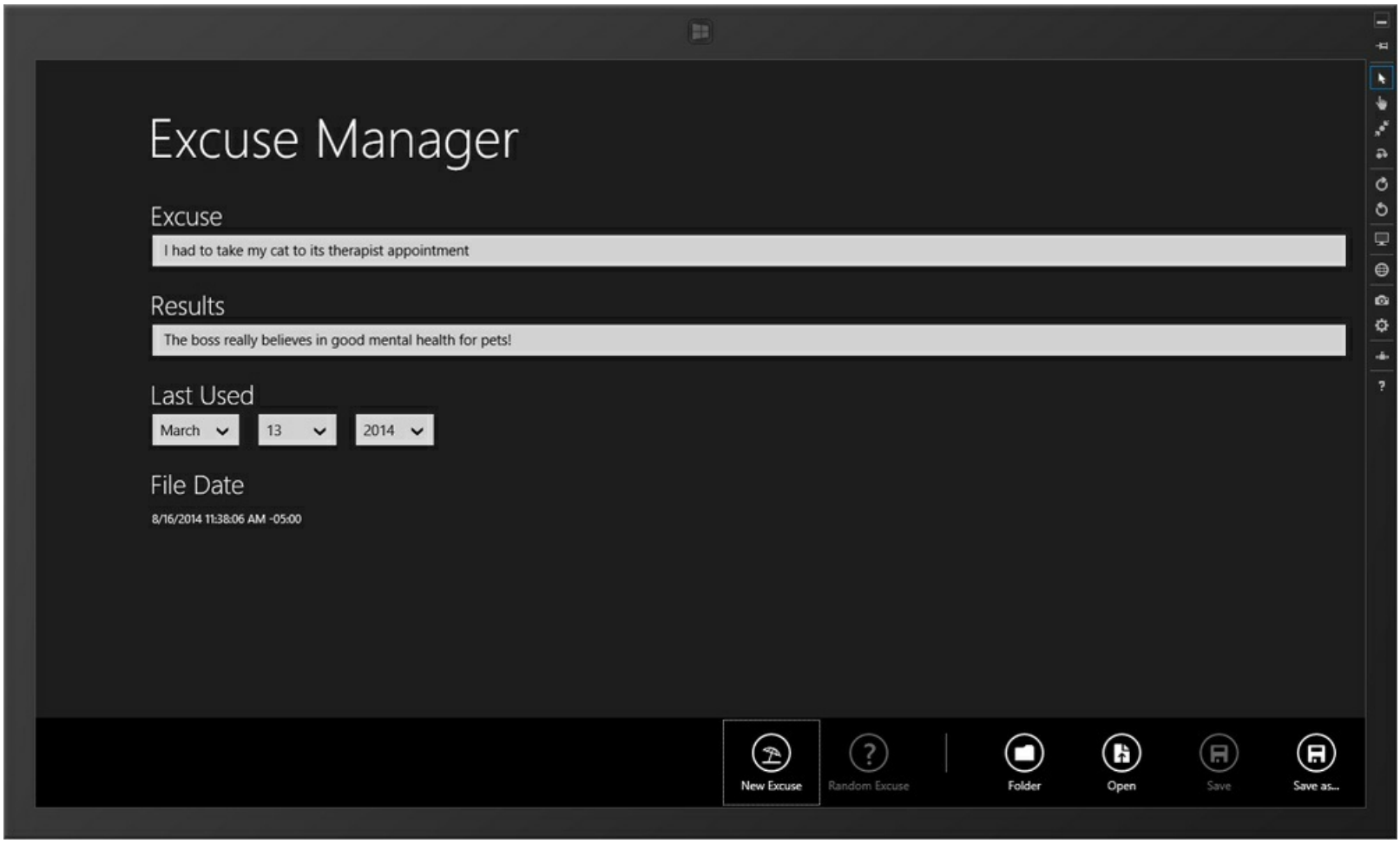
The `async` modifier, `await` keyword, and `Task` class make writing asynchronous code easier, and the way they do that is by encaspulating all of the work of yielding control into that `Task` class. Use "Go to Definition" to have a quick look at the properties and methods of the Task class. It has methods like `Run()`, `Continue()`, and `Wait()`, and properties like `IsCompleted` and `IsFaulted`. This should give you a hint about what's going on behind the scenes...and all of the things it does automatically in order to make it easier to write asynchronous methods.

You can read more about asynchronous programming here: http://msdn.microsoft.com/en-us/library/vstudio/hh191443.aspx

Build Brian a new Excuse Manager app

You know how to build XAML pages, read and write files, and serialize objects. It's time to put all of the pieces together and rebuild Brian his Excuse Manager as a Windows Store app.

Here's the main page:



RUN WINDOWS STORE APPS IN THE VISUAL STUDIO SIMULATOR

We captured the screenshot on this page using the simulator built into the IDE. The simulator is a desktop application that's installed with Visual Studio that lets you run your apps full-screen in a simulated device. This is really useful if you want to see how it responds to touch and hardware events, which can be really handy for testing. (This is a simulator, not an emulator.)

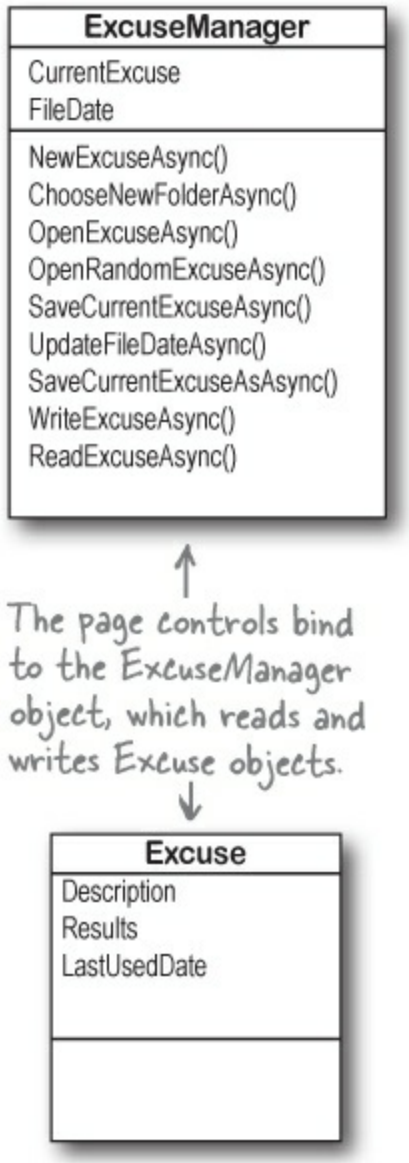
To start the simulator, click the drop-down arrow next to Local Machine and choose Simulator when you run your program. Now your app will launch in a simulator that shows how it will respond to full-screen touch and hardware events.

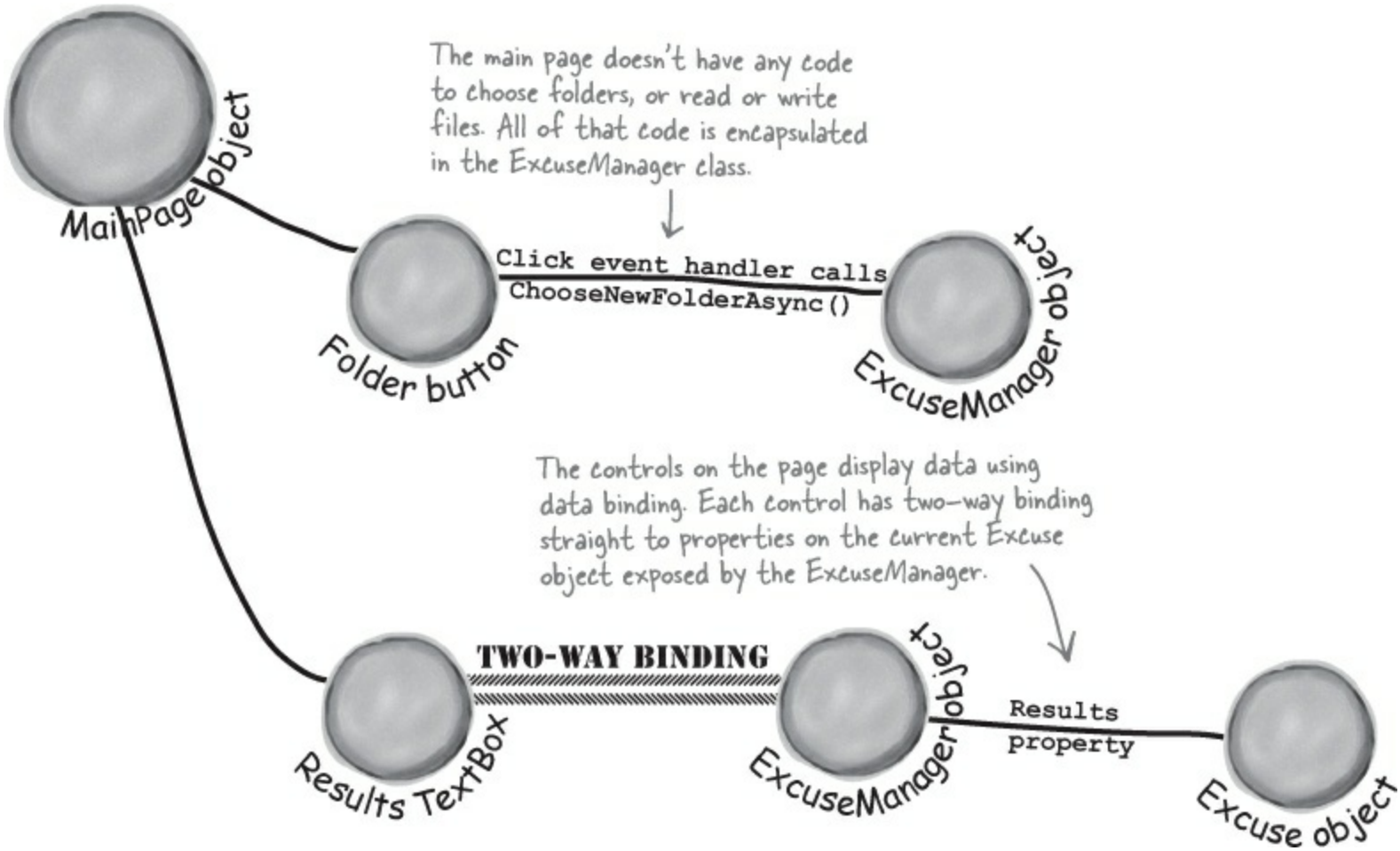
Learn more about navigating the simulator here: http://msdn.microsoft.com/en-us/library/windows/apps/hh441475.aspx.

Separate the page, excuse, and Excuse Manager

Your old `Excuse` object knew how to read and write itself, and that's not a bad way to design your objects. But there are other design choices that you can make. Your Guy Serializer app had the information about the `Guy` in one class, and methods to read and write `Guy` objects in the `GuyManager` class. You'll follow the same pattern for the new Excuse Manager app.

That's another example of the **separation of concerns** design principle that we talked about back in Chapter 5 and Chapter 6. The `Guy` just needs to expose the data contract; it's up to another class like `GuyManager` to determine what to do with that contract. And neither of those classes has any code for updating the user interface, because they're not concerned with displaying the excuse — that's the `MainPage` object's job.





BRAIN POWER

The `Excuse` and `ExcuseManager` classes don't have any code for updating the user interface... and you *can use data contract serialization or asynchronous programming in a WinForms program.* Could you use them to adapt the Windows Forms version of Brian's Excuse Manager to read and write the same excuse files as your new Windows Store Excuse Manager?

Create the main page for the Excuse Manager

DO THIS!

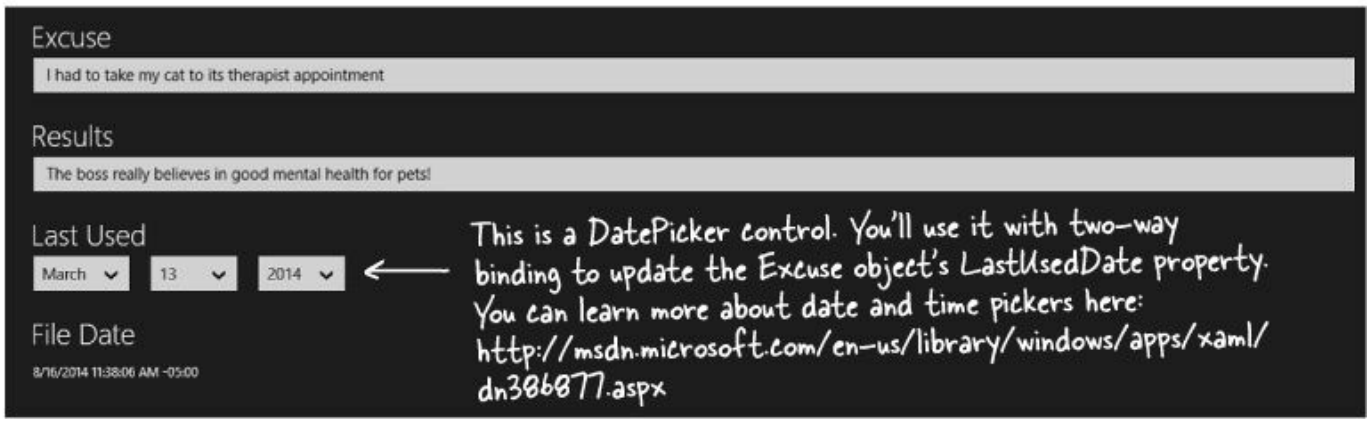
Create a new Windows Store app project and **replace *MainPage.xaml* with a new Basic Page**. You'll need a static `ExcuseManager` resource. Add an empty `ExcuseManager` class so your code compiles, then add it as a static resource to `<Page.Resources>`:

```
<Page.Resources>
    <local:ExcuseManager x:Name="excuseManager"/>
    <x:String x:Key="AppName">Excuse Manager</x:String>
</Page.Resources>
```

Here's the XAML for the page contents — it's a simple StackPanel-based layout. It sets the data context for the StackPanel to the `ExcuseManager` resource.

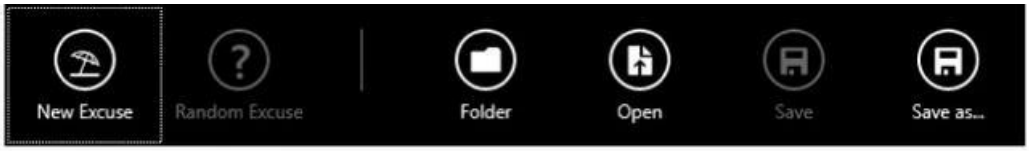
```
<StackPanel Grid.Row="1" Margin="120,0,0,0"
            DataContext="{StaticResource ResourceKey=excuseManager}">

    <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
               Text="Excuse" Margin="0,0,0,10"/>

    <TextBox Text="{Binding CurrentExcuse.Description, Mode=TwoWay}" Margin="0,0,20,20"/>

    <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
               Text="Results" Margin="0,0,0,10"/>

    <TextBox Text="{Binding CurrentExcuse.Results, Mode=TwoWay}" Margin="0,0,20,20"/>

    <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
               Text="Last Used" Margin="0,0,0,10"/>

    <DatePicker Date="{Binding CurrentExcuse.LastUsedDate, Mode=TwoWay}" Margin="0,0,0,20"/>

    <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
               Text="File Date" Margin="0,0,0,10"/>

    <TextBlock Text="{Binding FileDate}" Style="{StaticResource CaptionTextBlockStyle}"/>

</StackPanel>
```

The TextBox and DatePicker controls have two-way data binding to properties on the CurrentExcuse object in the ExcuseManager class. The TextBlock for the file date is bound to the ExcuseManager's FileDate property.



Add the app bar to the main page

Add a bottom app bar to the page, just like you did with the Guy Serializer earlier in the chapter. You'll need to uncomment `OpenFileAppBarButtonStyle`, `SaveAppBarButtonStyle`, and `FolderppBarButtonStyle` for the Open, Save, and Folder buttons.

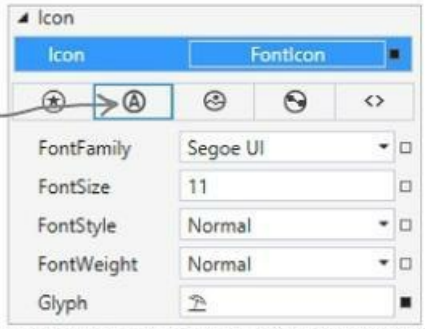


You'll need to add a BottomAppBar to the page, just like you did with the Simple Text Editor.

```
<Page.BottomAppBar>
    <AppBar>
      <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
        <AppBarButton Label="New Excuse" Click="NewClick">
          <AppBarButton.Icon>
            <FontIcon Glyph="&#x26F1;"/>
          </AppBarButton.Icon>
        </AppBarButton>
        <AppBarButton Icon="Help" Label="Random Excuse"
          Click="RandomClick" x:Name="randomButton" IsEnabled="False"/>
        <AppBarSeparator/>
        <AppBarButton Icon="Folder" Label="Folder" Click="FolderClick"/>
        <AppBarButton Icon="OpenFile" Label="Open" Click="OpenClick"/>
        <AppBarButton Icon="Save" Label="Save" x:Name="saveButton" IsEnabled="False" Click="SaveClick"/>
        <AppBarButton Icon="Save" Label="Save as..." Click="SaveAsClick"/>
      </StackPanel>
    </AppBar>
</Page.BottomAppBar>
```

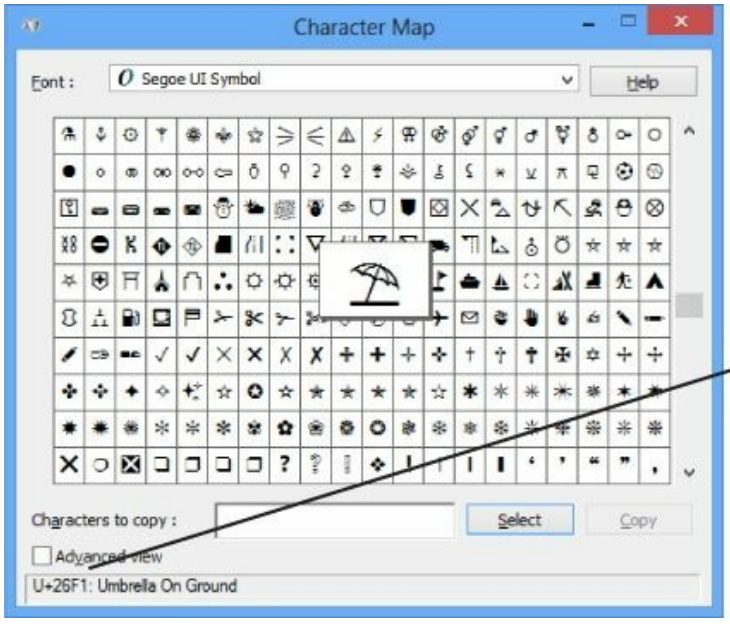
AppBar separators draw vertical bars to separate your AppBar buttons.

The New Excuse button uses a Font icon to make it display an umbrella.



The Random Excuse and Save buttons are disabled.

So how did that XAML change the picture in the New Excuse button? The icon is just text in the Segoe UI Symbol font. and the icon is a Unicode character in that font. XAML is based on XML, and you can insert any character into a string in XML with **`&#x`** followed by its Unicode number and a semicolon.



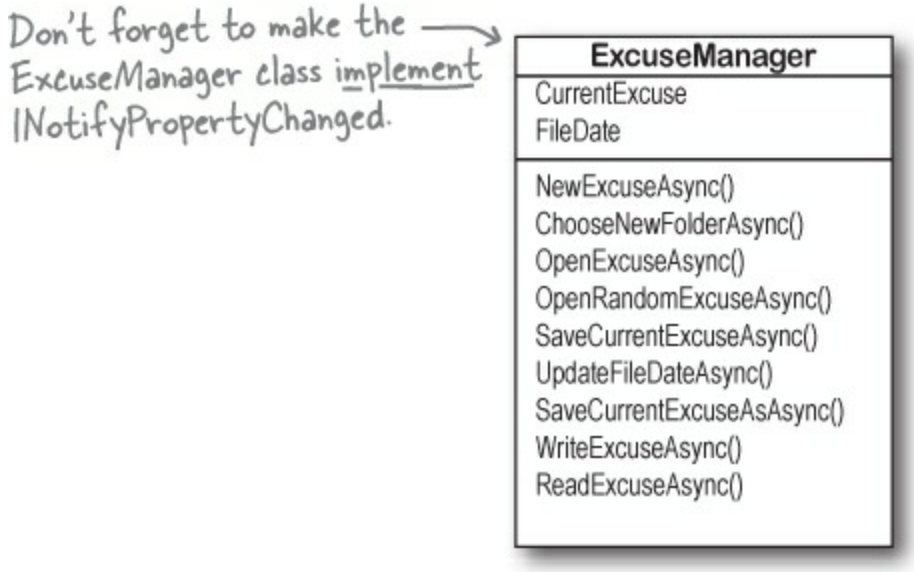
You can also copy the glyph out of Character Map and paste it into the Properties window. But you can also paste the glyph directly into the XAML editor, just like you did with the Hebrew letters back in Chapter 9. But if the Unicode character is isn't in the Segoe UI Symbol font, then the Glyph might not display the way you expect.

`Content="⛱"`

Next comes the ExcuseManager class

Build the ExcuseManager class

Here's *most* of the code for the `ExcuseManager` class — you'll finish the rest of the class and build the `Excuse` class as an exercise. It has two public properties for binding: `CurrentExcuse` is the currently loaded `Excuse` object, and `FileDate` is a string that either shows the file date or the string `"(no file loaded)"` (if the current excuse hasn't been saved or loaded). Since it's firing a PropertyChanged event, make sure you implement `INotifyPropertyChanged`.



The `ChooseNewFolderAsync()` method shows a folder picker, and returns true only if the user chose a folder. Since it's an async method that returns a bool value, its return type is `Task<bool>`.

```
public Excuse CurrentExcuse { get; set; }

public string FileDate { get; private set; }

private Random random = new Random();

private IStorageFolder excuseFolder = null;

private IStorageFile excuseFile;

public ExcuseManager() {
    NewExcuseAsync();
}
```

The excuseFile IStorageFile property keeps track of the current excuse file. It's reset to null if the current excuse hasn't been loaded or saved.

You'll need these using statements:

```
using System.ComponentModel;
using System.IO;
using System.Runtime.Serialization;
using Windows.Storage;
using Windows.Storage.Streams;
using Windows.Storage.FileProperties;
using Windows.Storage.Pickers;
using Windows.UI.Popups;
```

Task is in the System.Threading.Tasks namespace, but the IDE already added that using statement.

```
async public void NewExcuseAsync() {
    CurrentExcuse = new Excuse();
    excuseFile = null;
    OnPropertyChanged("CurrentExcuse");
    await UpdateFileDateAsync();
}
```

When the user clicks the New Excuse button, the ExcuseManager resets its current excuse, then calls UpdateFileDateAsync() to update the FileDate property.

You can call an async method like `NewExcuseAsync()` from a regular, non-asynchronous method. Just leave off the `await` keyword and the method will block. The IDE will give you a warning to make sure this is what you want to do.

```
public async Task<bool> ChooseNewFolderAsync() {
    FolderPicker folderPicker = new FolderPicker() {
        SuggestedStartLocation = PickerLocationId.DocumentsLibrary
    };
    folderPicker.FileTypeFilter.Add(".xml");
    IStorageFolder folder = await folderPicker.PickSingleFolderAsync();
    if (folder != null) {
        excuseFolder = folder;
        return true;
    }
    MessageDialog warningDialog = new MessageDialog("No excuse folder chosen");
    await warningDialog.ShowAsync();
    return false;
}
```

This asynchronous method returns a bool value, so its return type is Task<bool>.

If the user picked a folder, the method returns true. An async method that returns a Task<bool> just returns the bool value as usual.

The `FolderPicker` is another picker that lets you choose a folder. It works just like the other pickers you've seen. Have a look at all of the pickers in the `Windows.Storage.Pickers` namespace:
http://msdn.microsoft.com/library/windows/apps/BR207928

```
public async void OpenExcuseAsync() {
    FileOpenPicker picker = new FileOpenPicker {
        SuggestedStartLocation = PickerLocationId.DocumentsLibrary,
        CommitButtonText = "Open Excuse File"
    };
    picker.FileTypeFilter.Add(".xml");
    excuseFile = await picker.PickSingleFileAsync();
    if (excuseFile != null)
        await ReadExcuseAsync();
}

public async void OpenRandomExcuseAsync() {
    IReadOnlyList<IStorageFile> files = await excuseFolder.GetFilesAsync();
    excuseFile = files[random.Next(0, files.Count())];
    await ReadExcuseAsync();
}

public async Task UpdateFileDateAsync() {
    if (excuseFile != null) {
        BasicProperties basicProperties = await excuseFile.GetBasicPropertiesAsync();
        FileDate = basicProperties.DateModified.ToString();
    }
    else
        FileDate = "(no file loaded)";
    OnPropertyChanged("FileDate");
}

public async void SaveCurrentExcuseAsync() {
    if (CurrentExcuse == null) {
        await new MessageDialog("No excuse loaded").ShowAsync();
        return;
    }
    if (String.IsNullOrEmpty(CurrentExcuse.Description)) {
        await new MessageDialog("Current excuse does not have a description").ShowAsync();
        return;
    }
    if (excuseFile == null)
        excuseFile = await excuseFolder.CreateFileAsync(CurrentExcuse.Description + ".xml",
                        CreationCollisionOption.ReplaceExisting);

    await WriteExcuseAsync();
}

public async Task ReadExcuseAsync() {
    // You'll write this method
}

public async Task WriteExcuseAsync() {
    // You'll write this method
}

public async void SaveCurrentExcuseAsAsync() {
    // You'll write this method
}
```

The OpenExcuseAsync() method is just like the ReadGuyAsync() method in the Guy Serializer.

Uh oh! There's a bug somewhere around here. Can you spot it? You'll fix it in the next chapter.

UpdateFileDateAsync() sets the FileDate property to the last modified date of the current excuse file. If there's no excuse loaded, it sets it to a string. It's an async method that gets called by another async method, so it returns a Task.

The `IStorageFile.GetBasicPropertiesAsync()` method returns a BasicProperties object with read-only `DateModified` and `Size` properties that contain the modified date and size of the file. Its `DateModified` property has type DateTimeOffset, and your Excuse class will use the same type for its LastUsedDate property because that's what the DatePicker control binds to.

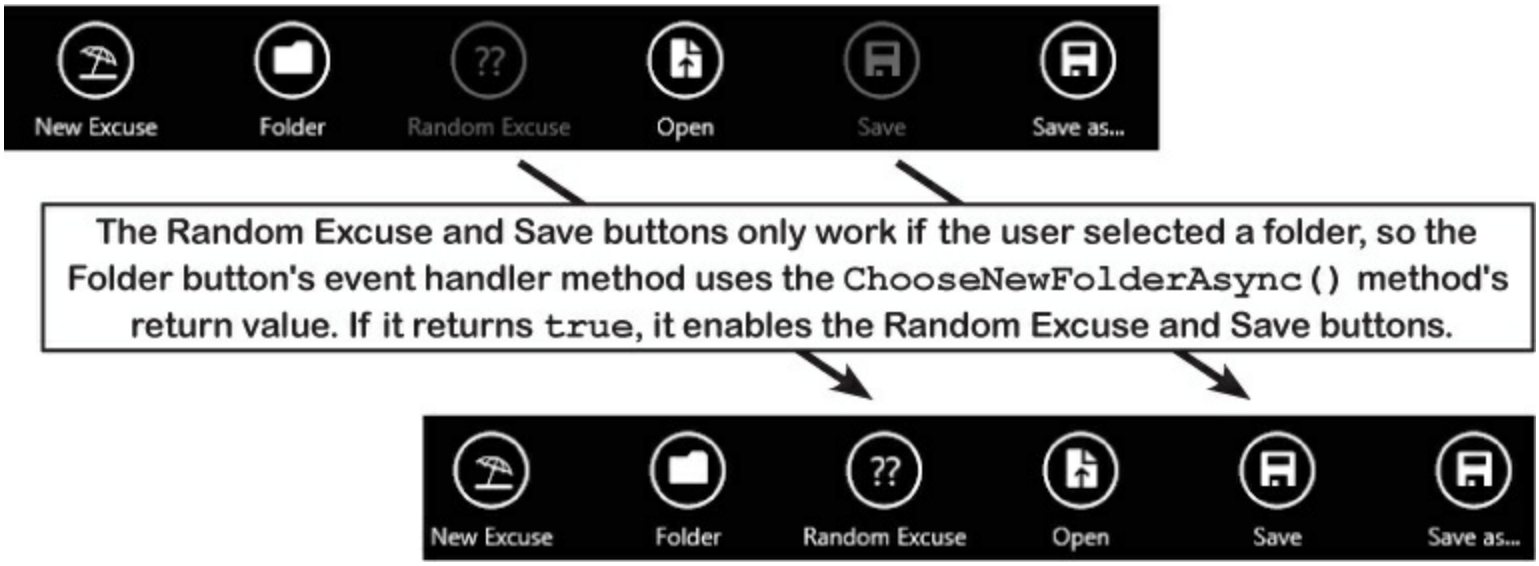
The SaveCurrentExcuseAsync() method first checks if the current excuse is null or if it has an empty description, and displays a warning message. If there's a valid excuse, it calls WriteExcuseAsync() to write the excuse. If there's no excuse file yet, it calls the folder's CreateFileAsync() method to create it.

Flip the page to finish porting the app

Add the code-behind for the page

This is all the code-behind you need. The event handlers for the buttons just call methods in the `ExcuseManager`. This is a benefit of separating the concerns about managing excuses from the concerns about displaying the user interface. Your user interface code tends to be very simple, because the other classes do most of the work.

Still thinking about that bug on the previous page? Here's a hint: what happens when you try to read a random excuse from a folder that you created back in Chapter 9? What about an empty folder?




```
private void OpenClick(object sender, RoutedEventArgs e) {
    excuseManager.OpenExcuseAsync();
}

private void SaveClick(object sender, RoutedEventArgs e) {
    excuseManager.SaveCurrentExcuseAsync();
}

private void NewClick(object sender, RoutedEventArgs e) {
    excuseManager.NewExcuseAsync();
}

private void SaveAsClick(object sender, RoutedEventArgs e) {
    excuseManager.SaveCurrentExcuseAsAsync();
}

private void RandomClick(object sender, RoutedEventArgs e) {
    excuseManager.OpenRandomExcuseAsync();
}

private async void FolderClick(object sender, RoutedEventArgs e) {
    bool folderChosen = await excuseManager.ChooseNewFolderAsync();
    if (folderChosen) {
        saveButton.IsEnabled = true;
        randomButton.IsEnabled = true;
    }
}
```

EXERCISE

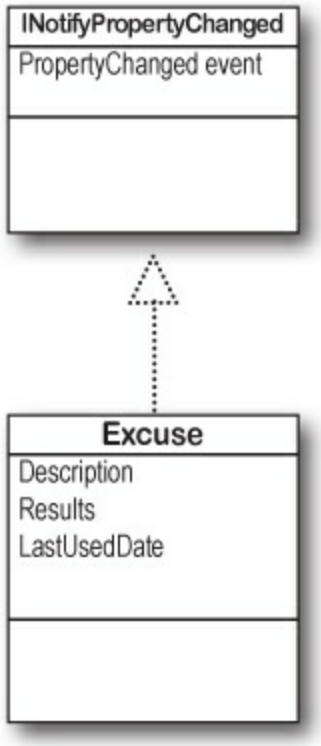
Finish the `Excuse` and `ExcuseManager` classes for Brian's new XAML Excuse Manager.

- **❶ Build the `Excuse` class.**
  It needs a data contract with the `http://www.headfirstlabs.com/ExcuseManager` namespace and three data members. The

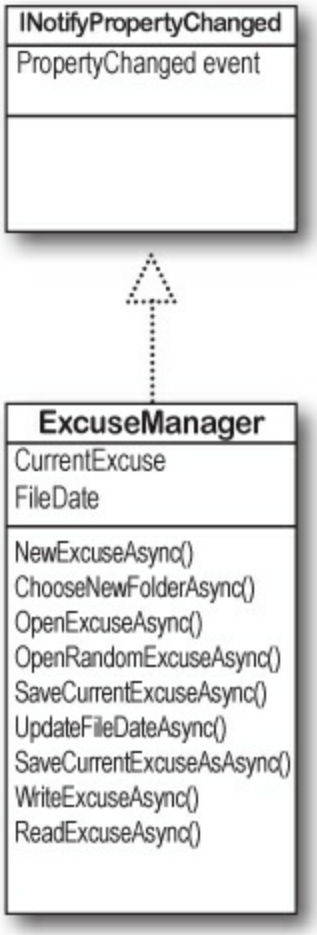
first two data members are the **`Description`** and **`Results`** automatic string properties. The third is a `DateTimeOffset` automatic property called **`LastUsedDate`**. Take another look at the XAML — it uses three controls that bind direcly to these properties like this:

```
{Binding CurrentExcuse.Description, Mode=TwoWay}
```

When you build your Excuse class, make sure you add a constructor that uses `DateTimeOffset`. Now to set the LastUsedDate property to the current date.



- **❷ Implement the `ExcuseManager.WriteExcuseAsync()` method.**
  This method opens a stream and serializes the current excuse to the excuse file managed by the `IStorageFile` currently stored in the `excuseFile` field. Then it displays a message that the excuse was written correctly, and calls `UpdateFileDateAsync()` to update the `FileDate` property.
- **❸ Implement the `ExcuseManager.ReadExcuseAsync()` method.**
  This method opens a stream and deserializes a new `Excuse` object from the excuse file managed by `excuseFile`. It fires a `PropertyChanged` event to let the page know that the `CurrentExcuse` was updated, then calls the `UpdateFileDateAsync()` method. You'll also need to **implement `INotifyPropertyChanged`** and add the `OnPropertyChanged()` method.



- ❹ **Implement the `ExcuseManager.SaveCurrentExcuseAsAsync()` method.**
  This method displays a `FileSavePicker` to let the user choose an XML file to save. If the user chooses one, it calls the `WriteExcuseAsync()` method to save the file.

WINDOWS STORE APPS USE DATETIMEOFFSET

When you built Windows Forms applications, you used DateTime values to store dates and times. Windows Store apps typically use DateTimeOffset values instead, because that's what controls like DatePicker and properties like BasicProperties.DateModified use. You can learn more about DateTimeOffset here: http://msdn.microsoft.com/en-us/library/system.datetimeoffset.aspx

EXERCISE SOLUTION

Here are the methods that you needed to add to the `ExcuseManager` class. Make sure the class extends `INotifyPropertyChanged`.

```
public async Task ReadExcuseAsync() {
    using (IRandomAccessStream stream =
            await excuseFile.OpenAsync(FileAccessMode.Read))
    using (Stream inputStream = stream.AsStreamForRead())     {
        DataContractSerializer serializer = new DataContractSerializer(typeof(Excuse));
        CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
    }

    await new MessageDialog("Excuse read from " + excuseFile.Name).ShowAsync();
    OnPropertyChanged("CurrentExcuse");
    await UpdateFileDateAsync();
}

public async Task WriteExcuseAsync() {
    using (IRandomAccessStream stream =
            await excuseFile.OpenAsync(FileAccessMode.ReadWrite))
    using (Stream outputStream = stream.AsStreamForWrite()) {
        DataContractSerializer serializer = new DataContractSerializer(typeof(Excuse));
        serializer.WriteObject(outputStream, CurrentExcuse);
    }
    await new MessageDialog("Excuse written to " + excuseFile.Name).ShowAsync();
    await UpdateFileDateAsync();
}

public async void SaveCurrentExcuseAsAsync() {
    FileSavePicker picker = new FileSavePicker {
        SuggestedStartLocation = PickerLocationId.DocumentsLibrary,
        SuggestedFileName = CurrentExcuse.Description,
        CommitButtonText = "Save Excuse File"
    };
    picker.FileTypeChoices.Add("XML File", new List<string>() { ".xml" });
    IStorageFile newExcuseFile = await picker.PickSaveFileAsync();
    if (newExcuseFile != null) {
        excuseFile = newExcuseFile;
        await WriteExcuseAsync();
    }
}

public event PropertyChangedEventHandler PropertyChanged;

private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

The methods to read and write Excuse objects are very similar to the corresponding methods in the Guy Serializer.

The SaveCurrentExcuseAsAsync() method shows a picker and then saves the excuse to the file it picked. It updates the excuseFile field to keep track of the new file that was saved (so the Save button saves to this new file).

Here's the normal code to fire the PropertyChanged event.

Here's the new `Excuse` class. It's got a data contract that includes the `Description` and `Results` properties and the `lastUsed` backing field for the `LastUsed` property.

```
using System.ComponentModel;
using System.Runtime.Serialization;

[DataContract(Namespace = "http://www.headfirstlabs.com/ExcuseManager")]
class Excuse : INotifyPropertyChanged
{
    [DataMember]
    public string Description { get; set; }

    [DataMember]
    public string Results { get; set; }

    [DataMember]
    public DateTimeOffset LastUsedDate { get; set; }

    public Excuse()
    {
        LastUsedDate = DateTimeOffset.Now;
    }

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
        if (propertyChangedEvent != null)
        {
            propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

This is the same code to fire a `PropertyChanged` event from earlier in the chapter. But if you copied and pasted it into your `Excuse` or `ExcuseManager` class and forgot to add `: INotifyPropertyChanged` to the class declaration, the controls on the page won't set up data binding. That means your objects will fire their `PropertyChanged` events, but without page controls listening for those events, the data binding will not work. That can be a frustrating bug!

Chapter 12. Exception Handling: Putting out fires gets old

Programmers aren't meant to be firefighters.

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

Brian needs his excuses to be mobile

Brian recently got reassigned to the international division. Now he flies all over the world. But he still needs to keep track of his excuses, so he installed the Excuse Manager app on his laptop and takes it with him everywhere.

But the program isn't working!

Brian chose a brand new, empty folder and clicked the Random Excuse button, and got a pretty nasty-looking error. What gives?

An unhandled exception...must have been a problem we didn't account for.

SHARPEN YOUR PENCIL SOLUTION

Here's another example of some broken code. There are five different exceptions that this code throws, and the error messages are shown on the right. It's your job to match the line of code that has a problem with the exception that line generates. Read the exception messages for a good hint.

```
public static void BeeProcessor() {
    object myBee = new HoneyBee(36.5, "Zippo");
    float howMuchHoney = (float)myBee;

    HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
    double beeName = double.Parse(anotherBee.MyName);

    double totalHoney = 36.5 + 12.5;
    string beesWeCanFeed = "";
    for (int i = 1; i < (int) totalHoney; i++) {
        beesWeCanFeed += i.ToString();
    }
    float f =
      float.Parse(beesWeCanFeed);

    int drones = 4;
    int queens = 0;
    int dronesPerQueen = drones / queens;

    anotherBee = null;
    if (dronesPerQueen < 10) {
        anotherBee.DoMyJob();
    }
}
```

Calling double.Parse("32") will parse a string and return a double value, like 32. The HoneyBee constructor's second parameter sets the MyName property.

An unhandled exception of type 'System.OverflowException' occurred in mscorlib.dll

Additional information: Value was either too large or too small for a Single. (1)

An unhandled exception of type 'System.NullReferenceException' occurred in BeeProcessingSystem.exe

Additional information: Object reference not set to an instance of an object. (2)

An unhandled exception of type 'System.InvalidCastException' occurred in BeeProcessingSystem.exe

Additional information: Specified cast is not valid. (3)

An unhandled exception of type 'System.DivideByZeroException' occurred in BeeProcessingSystem.exe

Additional information: Attempted to divide by zero. (4)

An unhandled exception of type 'System.FormatException' occurred in mscorlib.dll

Additional information: Input string was not in a correct format. (5)

SHARPEN YOUR PENCIL SOLUTION

Your job was to match the line of code that has a problem with the exception that line generates.

```
object myBee = new HoneyBee(36.5, "Zippo");
float howMuchHoney = (float)myBee;
```

C# lets you cast myBee to a float—but there's no way to convert a HoneyBee object to a float value. When your code runs, the CLR has no idea how to actually do that cast, so it throws an InvalidCastException.

An unhandled exception of type 'System.InvalidCastException' occurred in BeeProcessingSystem.exe

Additional information: Specified cast is not valid. (3)

```
HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
double beeName = double.Parse(anotherBee.MyName);
```

The Parse() method wants you to give it a string in a certain format. "Buzzy" isn't a string it knows how to convert to a number. That's why it throws a FormatException.

An unhandled exception of type 'System.FormatException' occurred in mscorlib.dll

Additional information: Input string was not in a correct format. (5)

```
double totalHoney = 36.5 + 12.5;
string beesWeCanFeed = "";
for (int i = 1; i < (int) totalHoney; i++) {
    beesWeCanFeed += i.ToString();
}
float f = float.Parse(beesWeCanFeed);
```

The for loop will create a string called beesWeCanFeed that contains a number with over 60 digits in it. There's no way a float can hold a number that big, and trying to cram it into a float will throw an OverflowException.

An unhandled exception of type 'System.OverflowException' occurred in mscorlib.dll

Additional information: Value was either too large or too small for a Single. (1)

You'd never actually get all these exceptions in a row — the program would throw the first exception and then stop. You'd only get to the second exception if you fixed the first.

```
int drones = 4;
int queens = 0;
int dronesPerQueen = drones / queens;
```

It's really easy to throw a DivideByZeroException. Just divide any number by zero.

An unhandled exception of type 'System.DivideByZeroException' occurred in BeeProcessingSystem.exe

Additional information: Attempted to divide by zero.

4

Dividing any integer by zero always throws this kind of exception. Even if you don't know the value of queens, you can prevent it just by checking the value to make sure it's not zero **before** you divide it into drones.

```
anotherBee = null;
if (dronesPerQueen < 10) {
    anotherBee.DoMyJob();
}
```

Setting the anotherBee reference variable equal to null tells C# that it doesn't point to anything. So instead of pointing to an object, it points to nothing. Throwing a NullReferenceException is C#'s way of telling you that there's no object whose DoMyJob() method can be called.

An unhandled exception of type 'System.NullReferenceException' occurred in BeeProcessingSystem.exe

Additional information: Object reference not set to an instance of an object.

2

That `DivideByZero` error didn't have to happen. You can see just by looking at the code that there's something wrong. The same goes for the other exceptions. These problems were preventable — and the more you know about exceptions, the better you'll be at keeping your code from crashing.

When your program throws an exception, .NET generates an Exception object

You've been looking at .NET's way of telling you something went wrong in your program: an **exception**. In .NET, when an exception occurs, an object is created to represent the problem. It's called — no surprise here — `Exception`.

> **NOTE**
>
> ex-cep-tion, noun.
>
> a person or thing that is excluded from a general statement or does not follow a rule. *While Jim usually hates peanut butter, he made an **exception** for Ken's peanut butter fudge.*

For example, suppose you have an array with four items. Then, you try to access the 16th item (index 15, since we're zero-based here):

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

This code is obviously going to cause problems.

As soon as your program runs into an exception, it generates an object with all the data it has about the problem.

Exception object

The exception object has a message that tells you what's wrong, and a stack trace, or a list of all of the calls that were made leading up to the statement that caused the exception.

Watch 1

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

Locals | Watch 1

When the IDE breaks because of an exception, you can see the details of the execption by **adding `$exception` to the Watch window**. It always shows up in the Locals window too, which is a lot like the Watch window but only shows current local variables.

.NET goes to the trouble of creating an object because it wants to give you all the information about what caused the exception. You may have code to fix, or you may just need to make some changes to how you handle a particular situation in your program.

In this case, an **`IndexOutOfRangeException`** indicates you have a bug: you're trying to access an index in the array that's out of range. You've also got information about exactly where in the code the problem occurred, making it easier to track down (even if you've got thousands of lines of code).

THERE ARE NO DUMB QUESTIONS

Q: **Q: Why are there so many kinds of exceptions?**

A: **A:** There are all sorts of ways that you can write code that C# simply doesn't know how to deal with. It would be difficult to troubleshoot your problems if your program simply gave a generic error message ("A problem occurred at line 37"). It's a lot easier to track down and fix problems in your code when you know specifically what kind of error occurred.

Q: **Q: So what *is* an exception, really?**

A: **A:** It's an object that .NET creates when there's a problem. You can specifically generate exceptions in your code, too (more about that in a minute).

Q: **Q: Wait, what? It's an *object*?**

A: **A:** Yes, an exception is an **object**. The properties in the object tell you information about the exception. For example, it's got a `Message` property that has a useful string like "Specified cast was invalid" or "Value was either too large or too small for a Single", which is what the IDE used to populate the `$exception` watch. The reason that .NET generates it is to give you as much information as it can about exactly what was going on when it executed the statement that threw the exception.

Q: **Q: OK, I still don't get it. Sorry. Why are there so many different kinds of exceptions, again?**

A: **A:** Because there are so many ways that your code can act in unexpected ways. There are a lot of situations that will cause your code to simply crash. It would be really hard to troubleshoot the problems if you didn't know why the crash happened. By throwing different kinds of exceptions under different circumstances, .NET is giving you a lot of really valuable information to help you track down and correct the problem.

Q: **Q: So exceptions are there to help me, not just cause a pain in my butt?**

A: **A:** Yes! Exceptions are all about helping you expect the unexpected. A lot of people get frustrated when they see code throw an exception. But if you think about an exception as .NET's way of helping you track down and debug your program, it really helps out when you're trying to track down what's causing the code to bomb out.

Q: **Q: So when my code throws an exception, it's not necessarily because I did something wrong?**

A: **A:** Exactly. Sometimes your data's different than you expected it to be — like you've got a method that's dealing with an array that's a lot longer or shorter than you anticipated when you first wrote it. And don't forget that human beings are using your program, and they almost always act in an unpredictable way. Exceptions are .NET's way to help you handle those unexpected situations so that your code still runs smoothly and doesn't simply crash or give a cryptic, useless error message.

Q: **Q: Once I knew what I was looking for, it was pretty clear that the code on the previous page was going to crash. Are all exceptions easy to spot?**

A: **A:** No. Unfortunately, there will be times when your code will have problems, and it'll be really hard to figure out what's causing them just by looking at it. That's why the IDE has a **debugger** — to help you get the bugs out by letting you pause your program, execute it statement by statement, and inspect the value of each individual variable and field as you go. That makes it a lot easier for you to figure out where your code is acting in a way that's different from how you expect it to act. That's when you have the best chance of finding and fixing the exceptions — or, even better, preventing them in the first place.

Exceptions are all about helping you find and fix situations where your code behaves in ways you didn't expect.

Brian's code did something unexpected

When Brian wrote his Excuse Manager, he never expected the user to try to pull a random excuse out of an empty directory.

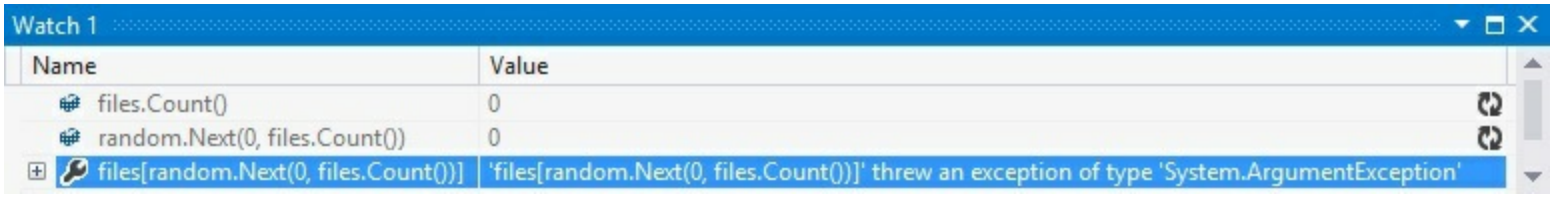
DO THIS!

- ❶ The problem happened when Brian pointed his Excuse Manager program at an empty folder on his laptop and clicked the Random Excuse button. Let's take a look at it and see if we can figure out what went wrong. Here's the unhandled exception window that popped up when he ran the program outside the IDE:

- ❷ OK, that's a good starting point. It's telling us that there's some value that doesn't fall inside some range. Clicking the Break button drops the IDE back into the debugger, with the execution halted on a specific line of code:

```
public async void OpenRandomExcuseAsync()
{
    IReadOnlyList<IStorageFile> files = await excuseFolder.GetFilesAsync();
    excuseFile = files[random.Next(0, files.Count())];
    await ReadExcuseAsync();
}
```

- ❸ Let's use the Watch window to track down the problem. Add a watch for `files.Count()`. Looks like that returns 0. Try adding a watch for `random.Next(0, files.Count())`. That returns 0, too. So add a watch for `files[random.Next(0, files.Count())]`.



NOTE

You can call methods and use indexers in the Watch window. When one of those things throws an exception, you'll see that exception in the Watch window too.

- ❹ So what happened? It turns out that calling `GetFilesAsync()` from an `IStorageFolder` object returns an `IReadOnlyList<IStorageFile>` collection. And like other collections that you've used, if you try to access an element that doesn't exist, it will throw an exception. Try to get the 0th element of an empty collection and your program will throw a `System.ArgumentException`, with the message, "Value does not fall within the expected range."
Luckily, there's an easy fix. Just check to see if the collection has items before getting a file:

```
public async void OpenRandomExcuseAsync()
{
    IReadOnlyList<IStorageFile> files = await excuseFolder.GetFilesAsync();
    if (files.Count() == 0) {
        await new MessageDialog("The current excuse folder is empty.").ShowAsync();
        return;
    }
    excuseFile = files[random.Next(0, files.Count())];
    await ReadExcuseAsync();
}
```

By checking for excuse files in the folder **before** we create the Excuse object, we can prevent the exception from being thrown—and display a helpful dialog, too.

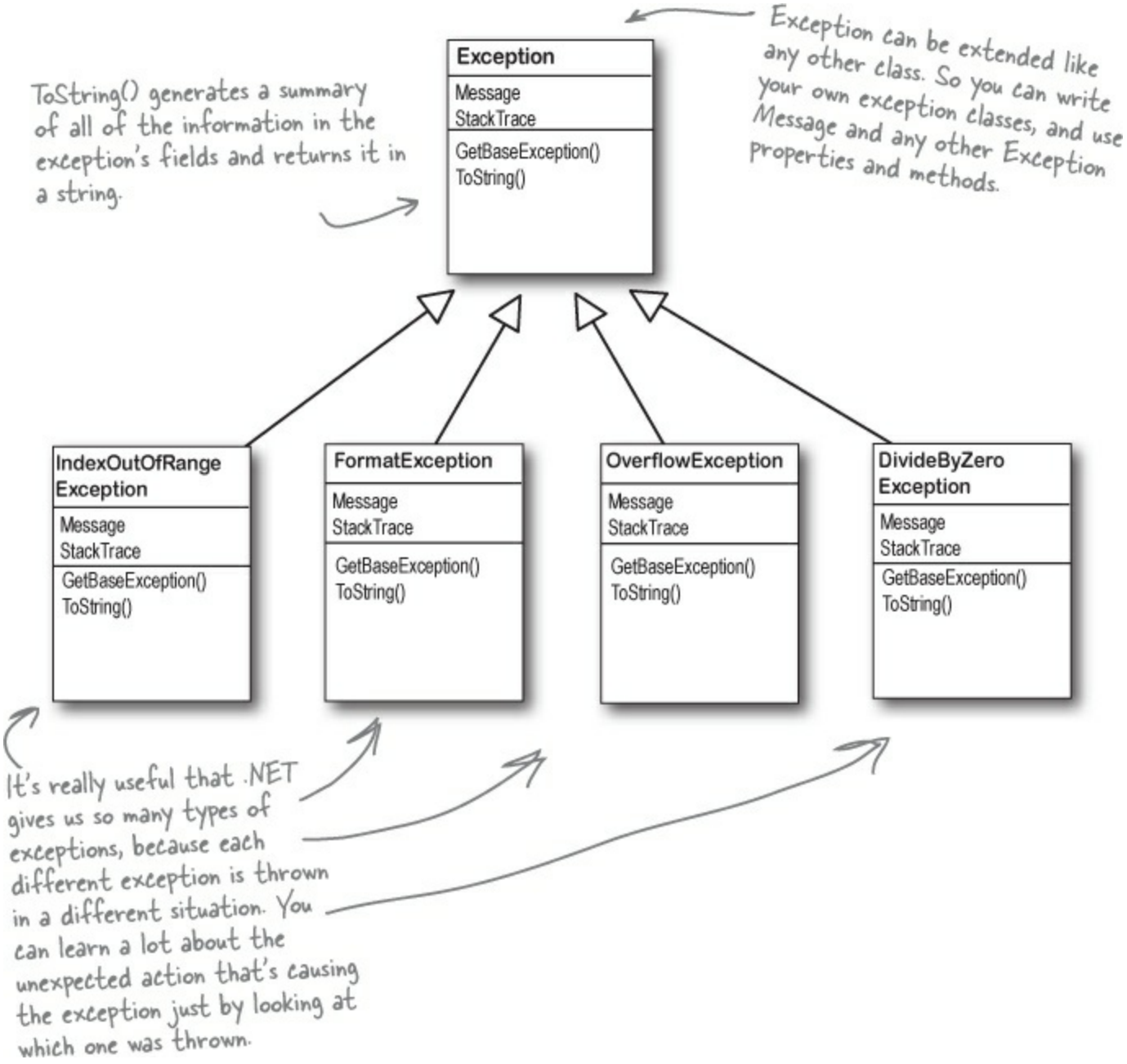
That's right. Exceptions are a really useful tool that you can use to find places where your code acts in ways you don't expect.

A lot of programmers get frustrated the first time they see an exception. But exceptions are really useful, and you can use them to your advantage. When you see an exception, it's giving you a lot of clues to help you figure out when your code is reacting to a situation that you didn't anticipate. And that's good for you: it lets you know about a new scenario that your program has to handle, and it gives you an opportunity to **do something about it**.

All exception objects inherit from Exception

.NET has lots of different exceptions it may need to report. Since many of these have a lot of similar features, inheritance comes into play. .NET defines a base class, called Exception, that all specific exceptions types inherit from.

The Exception class has a couple of useful members. The `Message` property stores an easy-to-read message about what went wrong. And `StackTrace` tells you what code was being executed when the exception occurred, and what led up to the exception. (There are others, too, but we'll use those first.)

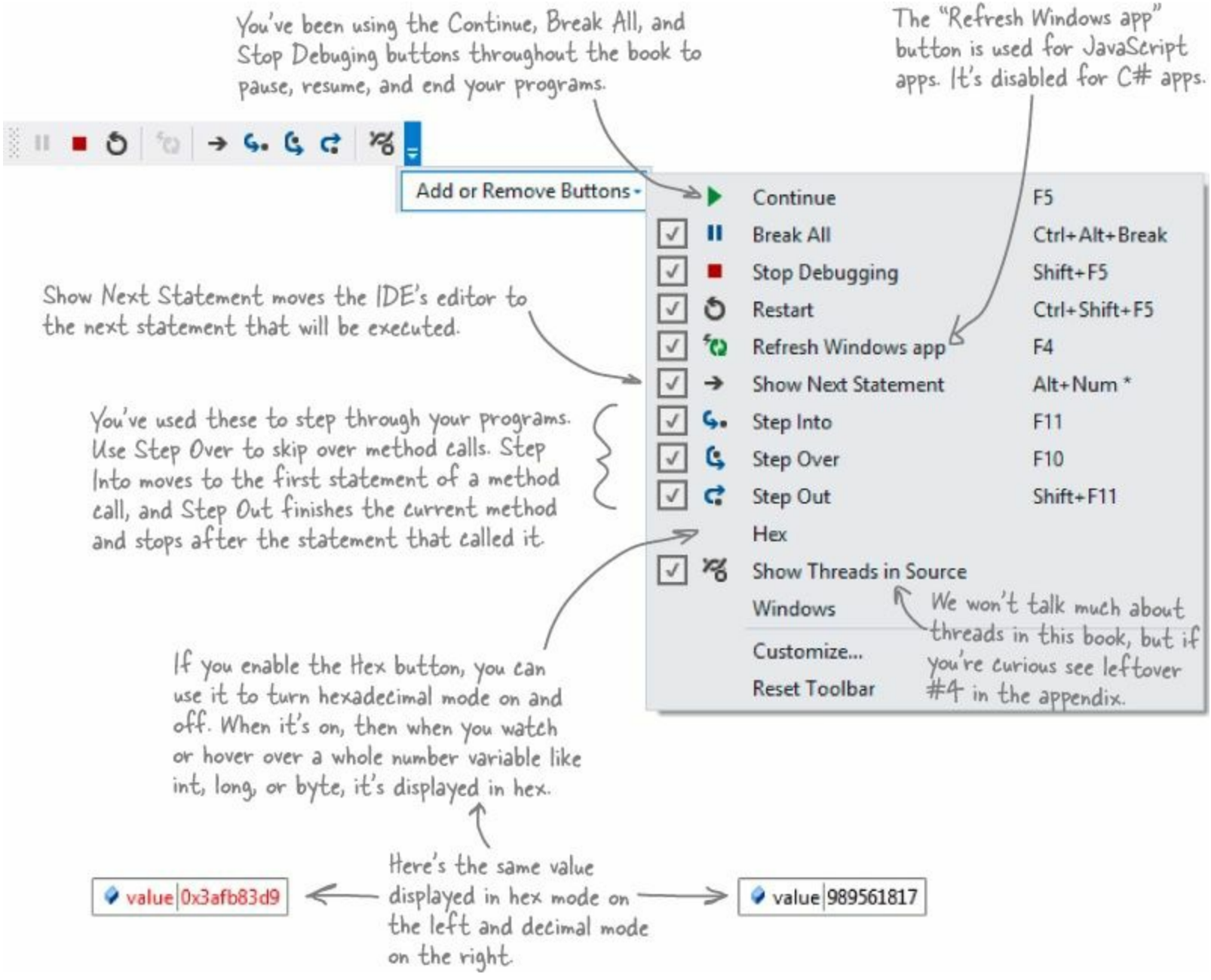


The debugger helps you track down and prevent exceptions in your code

Before you can add exception handling to your program, you need to know which statements in your program are throwing the exception. That's where the **debugger** that's built into the IDE can be really helpful. You've been using the debugger throughout the book, but now let's take a few minutes and really dig into it. When you run the debugger, the IDE pops up a toolbar with some really useful buttons.

> **NOTE**
>
> The Debug toolbar only shows up when you're debugging your program in the IDE. So you'll have to run a program in order to see this.

Click the icon on the Debug toolbar and choose "Add or Remove Buttons" to drill down into the various debugging commands that are available.



Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager

Let's use the debugger to take a closer look at the problem that we ran into in the Excuse Manager. You've probably been using the debugger a lot over the last few chapters, but we'll go through it step by step anyway to make sure we don't leave out any details.

DEBUG THIS

- **❶ ADD A BREAKPOINT TO THE RANDOM BUTTON'S EVENT HANDLER.**
  You've got a starting point — the exception happens when the Random Excuse button is clicked after an empty folder is selected. So open up the code-behind for the button and use Debug→Toggle Breakpoint (F9) to add a breakpoint to the method. Start debugging the app, **choose an empty folder**, then click the Random button to make your program break at the breakpoint:

```
private void RandomExcuseButtonClick(object sender, RoutedEventArgs e)
{
    excuseManager.OpenRandomExcuseAsync();
}
```

- **❷ STEP INTO THE OPENRANDOM EXCUSEASYNC() METHOD.**
  Use the **Step Into** command (using either the toolbar or the F11 key) to debug into the method. Then use **Step Over** (F10) to step through the method line by line. Since you selected an empty folder, you should see the program show the `MessageDialog()` and then exit the method.

Notice how the debugger waits for the MessageDialog even though it's called with the await keyword.

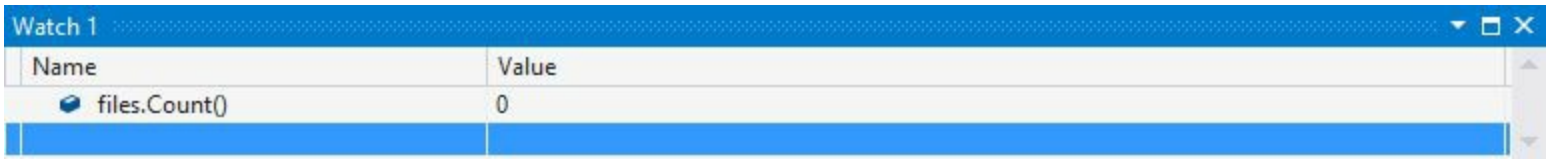
```
IReadOnlyList<IStorageFile> files = await excuseFolder.GetFilesAsync();
if (files.Count() == 0)
{
    await new MessageDialog("The current excuse folder is empty.").ShowAsync();
    return;
}
excuseFile = files[random.Next(0, files.Count())];
```

Now **select a folder with excuses in it**, click the Random button again, and step into the method again. This time, your code will skip past the `if` block and move on to the next line.

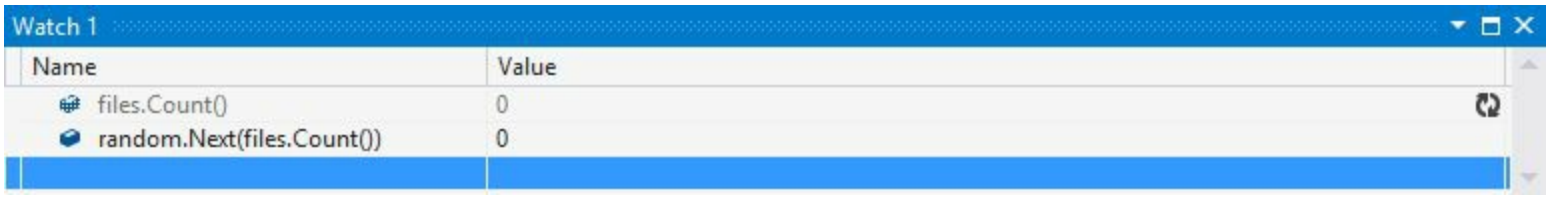
```
public async void OpenRandomExcuseAsync()
{
    IReadOnlyList<IStorageFile> files = await excuseFolder.GetFilesAsync();
    if (files.Count() == 0)
    {
        await new MessageDialog("The current excuse folder is empty.").ShowAsync();
        return;
    }
    excuseFile = files[random.Next(0, files.Count())];
    await ReadExcuseAsync();
}
```

- **❸ USE THE WATCH WINDOW TO START REPRODUCING THE PROBLEM.**
  You've already seen how handy the Watch window is. Now we'll use it to reproduce the exception. Stop the program, delete the old breakpoint, and **put a breakpoint on the second line of the `OpenRandomExcuseAsync()` method**. Start the program, choose an empty folder, then click the Random Excuse button. When the debugger breaks in the method, select `files.Count()`, right-click on it, and choose Add Watch to add a watch to the Watch window:

> **NOTE**
>
> You want to break on the second line because that's the line that accesses the files object.



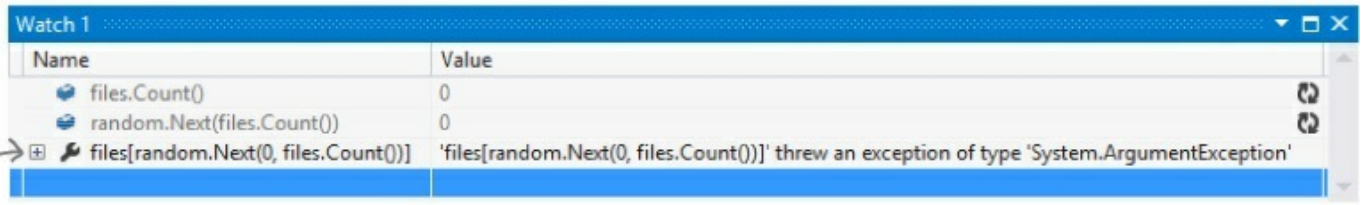
- **❹ ADD ANOTHER WATCH TO START TRACKING DOWN THE PROBLEM.**
  Debugging is a little like *performing a forensic crime scene investigation on your program.* You don't necessarily know what you're looking for until you find it, so you need to use your debugger "CSI kit" to follow clues and track down the culprit. Since `files.Count()` wasn't the guilty party, move on to the next suspect: select `random.Next(files.Count())` and add it to your Watch window:



  The Watch window has another very useful feature. It lets you **change the value** of variables and fields that it's displaying, and it even lets you **execute methods and create new objects**. When you do, it displays its reevaluate icon that you can click to tell it to execute that method again, because sometimes running the same method twice will generate different results (like with `Random`).

- **❺ CATCH THE CULPRIT THAT THREW BRIAN'S ORIGINAL EXCEPTION.**
  Here's where debugging gets really interesting. Add one more line to the debugger — the statement that actually threw the exception: `files[random.Next(0, files.Count())]`. As soon as you type it in, the Watch window evaluates it...and that throws the exception!

Even when you already fixed the problem by adding code to check that the folder has files, you can still use the Watch window to reproduce the exception.



  Click the + icon to expand the exception, and you'll see that its `Message` property contains "Value does not fall within the expected range." Now you know exactly what caused the problem, and why it happened. `GetFilesAsync()` returns a `IReadOnlyList<IStorageFile>` collection that has a count of 0 for an empty folder. If you try to use its indexer (`files[0]`), it will throw an `ArgumentException`.

When you get an exception, you can go back and reproduce it in the debugger and use the

`Exception` object to help you fix your code.

THERE ARE NO DUMB QUESTIONS

Q: **Q: When I run my app in the IDE, I can view the exception details using the Watch window. But what happens if I run the program outside of the IDE?**

A: **A:** There's an easy way to answer that question. Comment out the change you made the `OpenRandomExcuseAsync()` method to fix the problem, and then launch your app by choosing **Start Without Debugging** from the Debug menu. This will launch your app as if it were clicked from the Start screen. (You can also just go and click it from the Start screen.) Choose an empty folder, click the Random Excuse button, and...bam! Your app just disappears. That's what normally happens when an app has an **unhandled exception**. (You'll learn more about how to handle exceptions later in the chapter.) Most users don't want to see a cryptic window full of method names and exception details. But don't worry — your exception isn't lost. Open up the Windows Control Panel (you can search for "Control Panel" from the Start screen), search for "event," and **view the event logs**. Expand Windows Logs and click on Application. One of the events in the Application event log will contain your app's exception, including a **stack trace** that shows you the line that threw the exception, the line that called it, the one that called it, etc. (that's called the *call stack*). When you're debugging, the stack trace is in the `StackTrace` property of the `Exception` object.

Q: **Q: So that's it? When an exception happens outside the IDE, my program just stops and there's nothing I can do about it?**

A: **A:** Well, your program does stop when there's an ***unhandled*** exception. But that doesn't mean that all of your exceptions have to be unhandled! We'll talk a lot more about how you can handle exceptions in your code. There's no reason your users ever have to see an unhandled exception.

Q: **Q: How do I know where to put a breakpoint?**

A: **A:** That's a really good question, and there's no one right answer. When your code throws an exception, it's always a good idea to start with the statement that threw it. But usually, the problem actually happened earlier in the program, and the exception is just fallout from it. For example, the statement that throws a divide-by-zero error could be dividing values that were generated 10 statements earlier but just haven't been used yet. So there's no one good answer to where you should put a breakpoint, because every situation is different. But as long as you've got a good idea of how your code works, you should be able to figure out a good starting point.

Q: **Q: Can I run any method in the Watch window?**

A: **A:** Yes. Any statement that's valid in your program will work inside the Watch window, even things that make absolutely no sense to run inside a Watch window. Here's an example. Bring up a program, start it running, break it, and then **add this to the Watch window**: `System.Threading.Thread.Sleep(2000)`. That method causes your program to delay for two seconds. There's no reason you'd ever do that in real life, but it's interesting to see what happens: the IDE will block and you'll get a wait cursor for two seconds while the method evaluates. Then, since `Sleep()` has no return value, the Watch window will display the value `Expression has been evaluated and has no value` to let you know that it didn't return anything. But it did evaluate it. Not only that, but it displays IntelliSense pop ups to help you type code into the window. That's useful because it shows the available properties and methods for objects currently in memory.

Q: **Q: Wait, so isn't it possible for me to run something in the Watch window that'll change the way my program runs?**

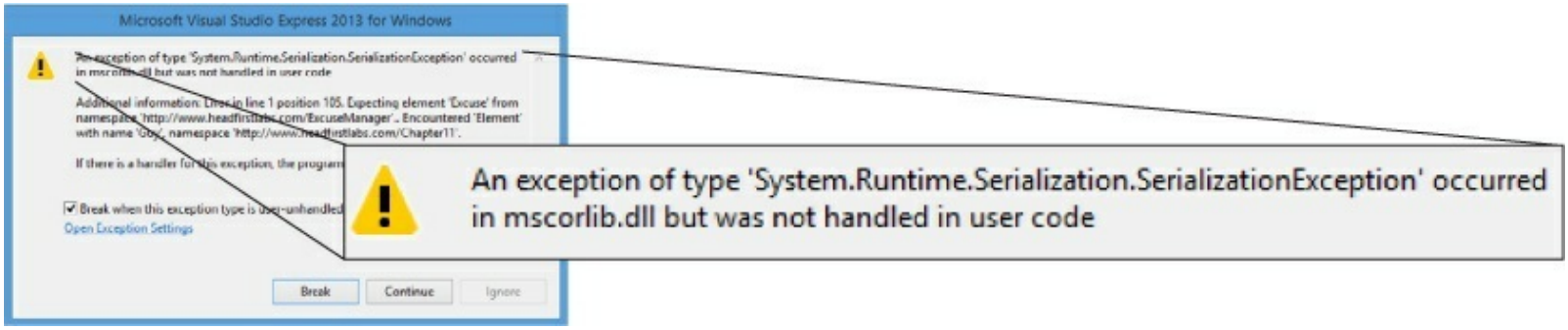
A: **A:** Yes! Not permanently, but it can definitely affect your program's output. But even better, just ***hovering*** over fields inside the debugger can cause your program to change its behavior, because hovering over a property ***executes its get accessor***. If you have a property that has a `get` accessor that executes a method, then hovering over that property will cause that method to execute. And if that method sets a value in your program, then that value will stay set if you run the program again. And that can cause some pretty unpredictable results inside the debugger. Programmers have a name for results that seem to be unpredictable and random: they're called **heisenbugs** (which is a joke that makes sense to physicists and cats trapped in boxes).

When you run your program inside the IDE, an unhandled exception will cause it to break as if it had run into a breakpoint.

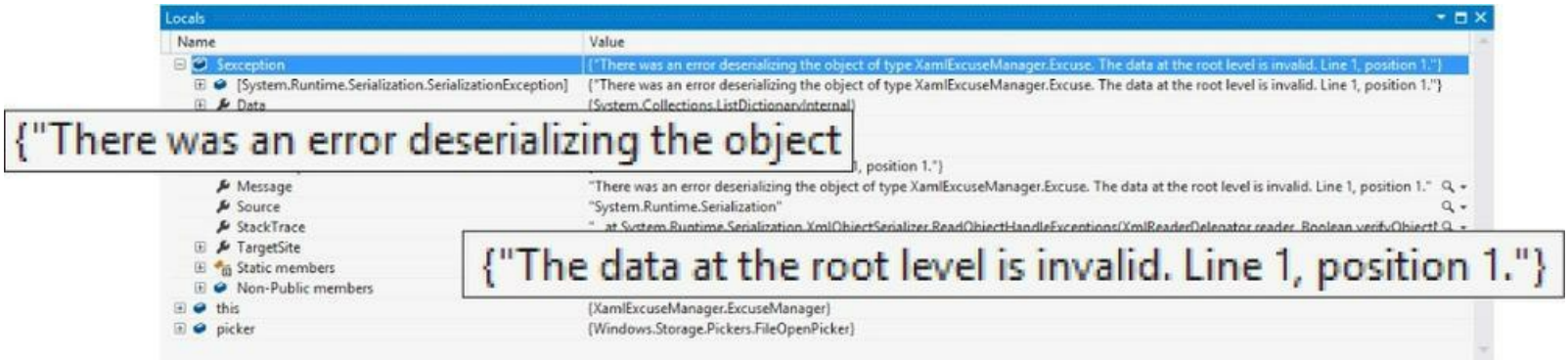
Uh oh — the code's still got problems...

Brian was happily using his Excuse Manager when he accidentally chose a folder full of XML files that weren't created by the Excuse Manager. Let's see what happens when he tries to load one of them....

- ❶ You can re-create Brian's problem. Find one of the XML files that contains a serialized `Excuse` object. Open it up in Notepad and and add some invalid, non-XML text to the very beginning, right before the opening < character.
- ❷ Pop open the Excuse Manager in the IDE and open up the excuse. It throws an exception! Look at the message, then click the Break button to start investigating.



- ❸ Open up the Locals window and expand `$exception` (you can also enter it into the Watch window). Take a close look at its members to see if you can figure out what went wrong.



DO YOU SEE WHY THE PROGRAM THREW THE EXCEPTION? DOES IT MAKE SENSE FOR THE PROGRAM TO CRASH IF IT ENCOUNTERS AN INVALID EXCUSE XML FILE? CAN YOU THINK OF ANYTHING YOU CAN DO ABOUT THIS?

Actually, there is something you can do about it.

Yes, it's true that users screw up all the time. That's a fact of life. But that doesn't mean you can't do anything about it. There's a name for programs that deal with bad data, malformed input, and other unexpected situations gracefully: they're called **robust** programs. And C# gives you some really powerful exception handling tools to help you make your programs more robust. Because while you *can't* control what your users do, you *can* make sure that your program doesn't crash when they do it.

NOTE

ro-bust, adj.

sturdy in construction; able to withstand or overcome adverse conditions. *After the Tacoma Narrows Bridge disaster, the civil engineering team looked for a more **robust** design for the bridge that would replace it.*

WATCH IT!

Serializers will throw an exception if there's anything at all wrong with a serialized file.

It's easy to get the Excuse Manager to throw a `SerializationException` — just feed it any file that's not a serialized Excuse object. When you try to deserialize an object from a file, `DataContractSerializer` expects the file to contain a serialized object that matches the contract of the class that it's trying to read. If the file contains anything else, almost anything at all, then the `ReadObject()` method will throw a `SerializationException`.

The BinaryFormatter class will also throw a SeralizationException if you give it a file that doesn't contain exactly the right serialized object. It's even more finicky than DataContractSerializer!

Handle exceptions with try and catch

In C#, you can basically say, "**Try** this code, and if an exception occurs, **catch** it with this *other* bit of code." The part of the code you're trying is the **`try` block**, and the part where you deal with exceptions is called the **`catch` block**. In the `catch` block, you can do things like print a friendly error message instead of letting your program come to a screeching halt:

Put the code that might throw an exception inside the try block. If no exception happens, it'll get run exactly as usual, and the statements in the catch block will be ignored. But if a statement in the try block throws an exception, the rest of the try block **won't get executed**.

```
public async Task ReadExcuseAsync() {
    try
    {
        using (IRandomAccessStream stream =
                await excuseFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead()) {
          DataContractSerializer serializer
                    = new DataContractSerializer(typeof(Excuse));
          CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
        }

        await new MessageDialog("Excuse read from "
                                + excuseFile.Name).ShowAsync();
        OnPropertyChanged("CurrentExcuse");
        await UpdateFileDateAsync();
    }
    catch (SerializationException)
    {
        new MessageDialog("Unable to read " + excuseFile.Name).ShowAsync();
    }
}
```

This is the **try block**. You start exception handling with try. In this case, we'll put the existing code in it.

You'll recognize the code here because we surrounded the entire method with this try block.

The catch keyword means that the block immediately following it contains an **exception handler**.

When an exception is thrown, the program immediately jumps to the **catch** statement and starts executing the **catch block**.

This is the simplest kind of exception handling: stop the program, write out the exception message, and keep running. Notice how there's no `await` keyword when showing the `MessageDialog`? That's because **you can't await in the body of a `catch` clause**. Luckily, you can still call the `ShowAsync()` method, but it will block until the user dismisses the dialog.

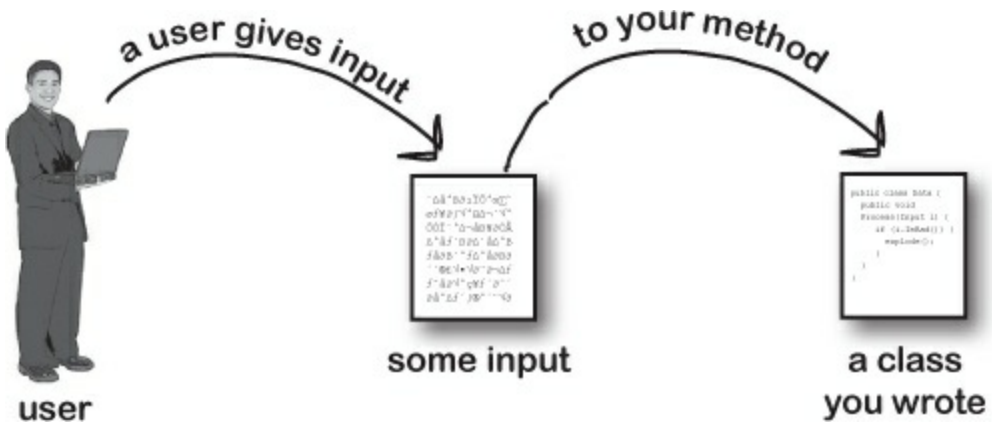
BRAIN POWER

If throwing an exception makes your code automatically jump to the `catch` block, what happens to the objects and data you were working with before the exception happened?

What happens when a method you want to call is risky?

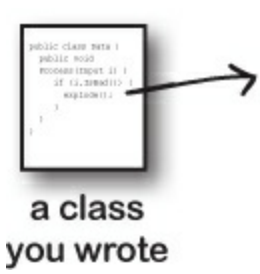
Users are unpredictable. They feed all sorts of weird data into your program, and click on things in ways you never expected. And that's just fine, because you can handle unexpected input with good exception handling.

- **❶ Let's say your user is using your code, and gives it some input that it didn't expect.**



- **❷ That method does something risky, something that might not work at runtime.**

> **NOTE**
>
> "Runtime" just means "while your program is running." Some people refer to exceptions as "runtime errors."



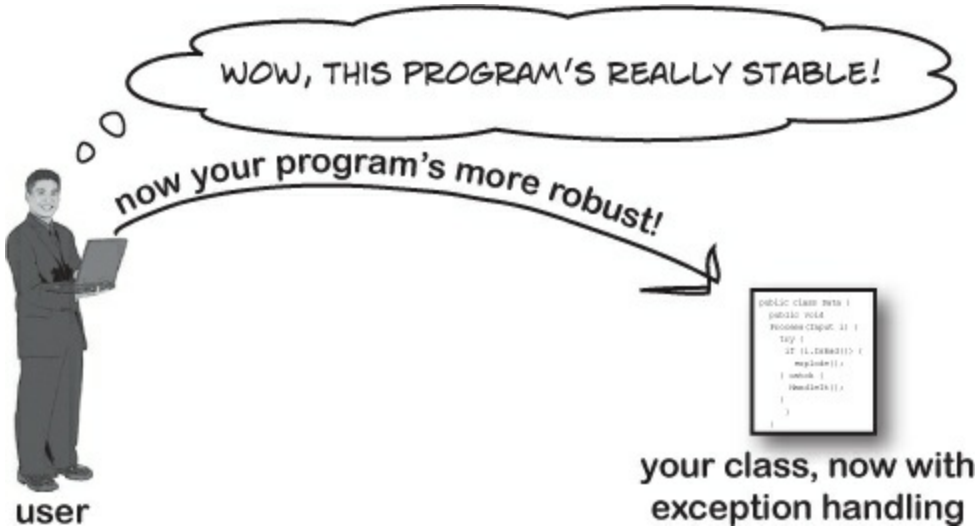

```
public void
    Process(Input i) {
    if (i.IsBad()) {
        Explode();
    }
}
```

- **❸ You need to *know* that the method you're calling is risky.**

- **❹ You then write code that can handle the failure if it *does* happen. You need to be prepared, just in case.**

NOTE

If you can come up with a way to do a less risky thing that avoids throwing the exception, that's the best possible outcome! But some risks just can't be avoided, and that's when you want to do this.



THERE ARE NO DUMB QUESTIONS

Q: **Q: So when do I use `try` and `catch`?**

A: **A:** Any time you're writing risky code, or code that could throw an exception. The trick is figuring out which code is risky, and which code is safer. You've already seen that code that uses input provided by a user can be risky. Users give you incorrect files, words instead of numbers, and names instead of dates, and they pretty much click everywhere you could possibly imagine. A good program will take all that input and work in a calm, predictable way. It might not give the users a result they can use, but it will let them know that it found the problem and hopefully suggest a solution.

Q: **Q: How can a program suggest a solution to a problem it doesn't even know about in advance?**

A: **A:** That's what the `catch` block is for. A `catch` block is only executed when code in the `try` block throws an exception. It's your chance to make sure the user knows that something went wrong, and to let the user know that it's a situation that might be corrected.
If the Excuse Manager simply crashes when there's bad input, that's not particularly useful. But if it tries to read the input and displays garbage in the form, that's also not useful — in fact, some people might say that it's worse. But if you have the program display an error message telling the user that it couldn't read the file, then the user has an idea of what went wrong, and information that he can use to fix the problem.

Q: **Q: So the debugger should really only be used to troubleshoot exceptions then?**

A: **A:** No. As you've already seen many times throughout the book, the debugger's a really useful tool that you can use to examine any code you've written. Sometimes it's useful to step through your code and check the values of certain fields and variables — like when you've got a really complex method, and you want to make sure it's working properly.
But as you may have guessed from the name "debugger," its most common use is to track down and remove bugs. Sometimes those bugs are exceptions that get thrown. But a lot of the time, you'll be using the debugger to try to find other kinds of problems, like code that gives a result that you don't expect.

Q: **Q: I'm not sure I totally got what you did with the Watch window.**

A: **A:** When you're debugging a program, you usually want to pay attention to how a few variables and fields change. That's where the Watch window comes in. If you add watches for a few variables, the Watch window updates their values every time you step into, out of, or over code. That lets you monitor exactly what happens to them after every statement, which can be really useful when you're trying to track down a problem.
The Watch window also lets you type in any statement you want, and even call methods, and the IDE will evaluate it and display the results. If the statement updates any of the fields and variables in your program, then it does that, too. That lets you change values while your program is running, which can be another really useful tool for reproducing exceptions and other bugs.

NOTE

Any changes you make in the Watch window just affect the data in memory, and only last as long as the program is running. Restart your program, and values that you changed will be undone.

The catch block is only executed when code in the try block throws an exception. It gives you a chance to make sure your user has the information to fix the problem.

Use the debugger to follow the try/catch flow

An important part of exception handling is that when a statement in your `try` block throws an exception, the rest of the code in the block gets **short-circuited**. The program's execution immediately jumps to the first line in the `catch` block. ***But don't take our word for it....***

DEBUG THIS

- ❶ Add the `try/catch` from a few pages ago to your Excuse Manager app's `ReadExcuseAsync()` method. Then place a breakpoint on the opening bracket `{` in the `try` block.

  > **`SerializationException` is in the `System.Runtime.Serialization` namespace. Luckily, you already have `using System.Runtime.Serialization` at the top of your *ExcuseManager.cs* file.**

- ❷ Start debugging your app and open up a file that's **not a valid excuse file** (but still has the *.xml* extension). When the debugger breaks on your breakpoint, click the Step Over button (or F10) five times to get to the statement that calls `ReadObject()` to deserialize the `Excuse` object. Here's what your debugger screen should look like:

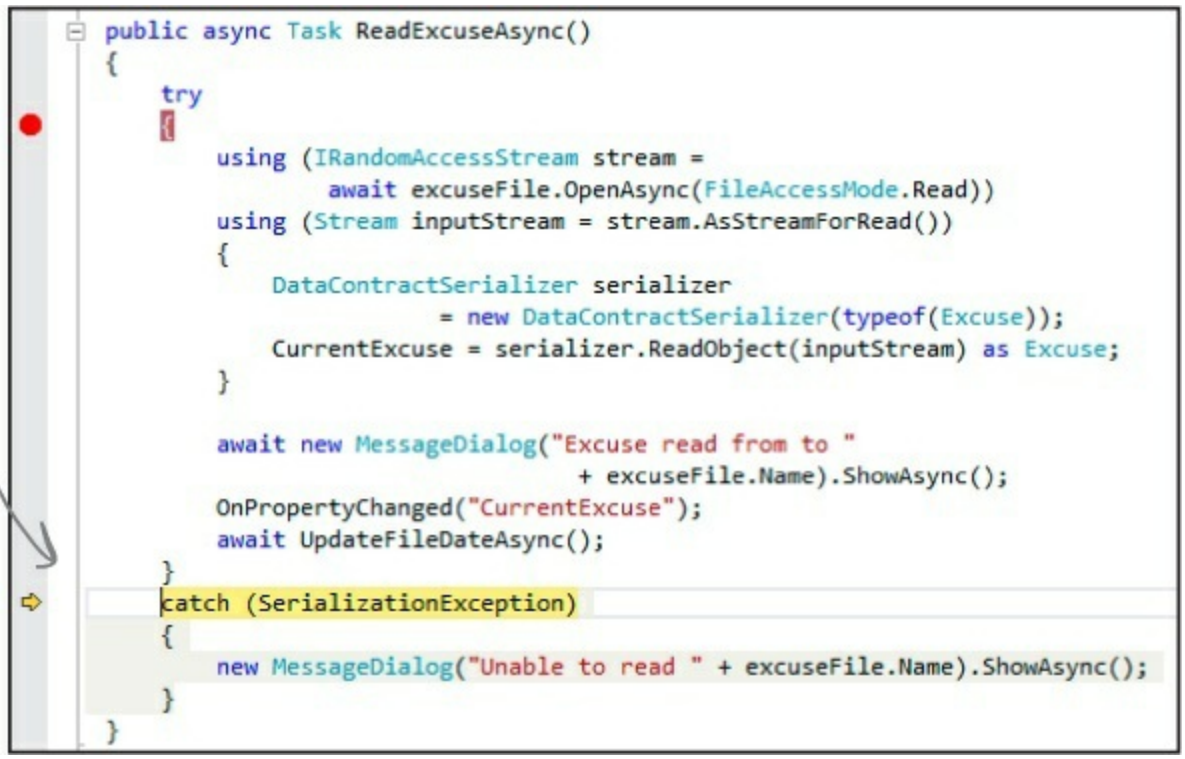
Put the breakpoint on the opening bracket of the try block.

Step over the statements until your yellow "next statement" bar shows that the next statement to get executed will read the Excuse object from the stream.

```
public async Task ReadExcuseAsync()
{
    try
    {
        using (IRandomAccessStream stream =
                await excuseFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead())
        {
            DataContractSerializer serializer
                        = new DataContractSerializer(typeof(Excuse));
            CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
        }

        await new MessageDialog("Excuse read from to "
                                + excuseFile.Name).ShowAsync();
        OnPropertyChanged("CurrentExcuse");
        await UpdateFileDateAsync();
    }
    catch (SerializationException)
    {
        new MessageDialog("Unable to read " + excuseFile.Name).ShowAsync();
    }
}
```

- ❸ Keep stepping through the code. As soon as the debugger executes the `ReadObject()` statement, the exception is thrown and the program **short-circuits** right past the rest of the method and **jumps straight to the `catch` block**.

The debugger will highlight the catch statement with its yellow "next statement" block, but it shows the rest of the block in gray to show you that it's about to execute the whole thing.



```
public async Task ReadExcuseAsync()
{
    try
    {
        using (IRandomAccessStream stream =
                await excuseFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead())
        {
            DataContractSerializer serializer
                        = new DataContractSerializer(typeof(Excuse));
            CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
        }

        await new MessageDialog("Excuse read from to "
                                    + excuseFile.Name).ShowAsync();
        OnPropertyChanged("CurrentExcuse");
        await UpdateFileDateAsync();
    }
    catch (SerializationException)
    {
        new MessageDialog("Unable to read " + excuseFile.Name).ShowAsync();
    }
}
```

- ❹ Start the program again by pressing the Continue button (or F5). It'll begin running the program again, starting with whatever's highlighted by the yellow "next statement" block — in this case, the `catch` block. It will just display the dialog, and then act as if nothing happened. The ugly crash has now been handled.

WATCH IT!

Keep risky code out of the constructor!

You've noticed by now that a constructor doesn't have a return value, not even `void`. That's because a constructor doesn't actually return anything. Its only purpose is to initialize an object — which is a problem for exception handling inside the constructor. When an exception is thrown inside the constructor, then the statement that tried to instantiate the class ***won't end up with an instance of the object****.*

Here's a career tip: a lot of C# programming job interviews include a question about how you deal with exceptions in a constructor.

If you have code that ALWAYS should run, use a finally block

When your program throws an exception, a couple of things can happen. If the exception ***isn't*** handled, your program will stop processing and crash. If the exception ***is*** handled, your code jumps to the `catch` block. But what about the rest of the code in your `try` block? What if you were closing a stream, or cleaning up important resources? That code needs to run, even if an exception occurs, or you're going to make a mess of your program's state. That's where the **`finally`** block comes in really handy. It comes after the `try` and `catch` blocks. The **`finally` block always runs**, whether or not an exception was thrown. Here's how you can use it to make sure the `ReadExcuseAsync()` method always fires the `PropertyChanged` event:

```
public async Task ReadExcuseAsync() {
    try
    {
        using (IRandomAccessStream stream =
                await excuseFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead()) {
            DataContractSerializer serializer
                        = new DataContractSerializer(typeof(Excuse));
            CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
        }

        await new MessageDialog("Excuse read from to "
                                    + excuseFile.Name).ShowAsync();
        await UpdateFileDateAsync();
    }
    catch (SerializationException)
    {
        new MessageDialog("Unable to read " + excuseFile.Name).ShowAsync();
        NewExcuseAsync();
    }
    finally
    {
        OnPropertyChanged("CurrentExcuse");
    }
}
```

NewExcuseAsync() resets the Excuse object, but the page won't read the CurrentExcuse property if the PropertyChanged event doesn't fire. The finally block makes sure that the PropertyChanged event gets fired whether or not an exception was thrown.

Adding NewExcuseAsync() to the catch block clears the form if the Excuse constructor throws an exception.

Always catch specific exceptions like `SerializationException`. You typically follow a `catch` statement with a specific kind of exception telling it what to catch. It's valid C# code to just have **`catch (Exception)`** and you can even leave the exception type out and just use **`catch`**. When you do that, it **catches all exceptions**, no matter what type of exception is thrown. But it's a ***really bad practice to have a catch-all exception handler*** like that. Your code should always catch as specific an exception as possible.

NOW DEBUG THIS

- ❶ Update the `ReadExcuseAsync()` method with the code on the facing page. Then place a breakpoint on the opening bracket in the `try` block and debug the program.
- ❷ Run the program normally, and make sure that the Open button works when you load a working excuse file. The debugger should break at the breakpoint you set:

When the "next statement" bar and the breakpoint are on the same line, the IDE shows you the yellow arrow placed over the big red dot in the margin.

```
public async Task ReadExcuseAsync()
{
    try
    {
        using (IRandomAccessStream stream =
                await excuseFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead())
        {
            DataContractSerializer serializer
                        = new DataContractSerializer(typeof(Excuse));
            CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
        }

        await new MessageDialog("Excuse read from to "
                                        + excuseFile.Name).ShowAsync();
        await UpdateFileDateAsync();
    }
    catch (SerializationException)
    {
        new MessageDialog("Unable to read " + excuseFile.Name).ShowAsync();
        NewExcuseAsync();
    }
    finally
    {
        OnPropertyChanged("CurrentExcuse");
    }
}
```

Pay special attention to what happens with the dialogs. Sometimes a dialog won't get displayed until after your method finishes. Welcome to the world of asynchronous methods!

- ❸ Step through the rest of the method and make sure it runs the way you expect it to. It should finish the `try` block, skip over the `catch` block (because no exceptions were thrown), and then execute the `finally` block.
- ❹ Now try opening a malformed excuse file. The method should start executing the `try` block, and then jump to the `catch` block when it throws the exception. After it finishes all of the statements in the `catch` block, it'll execute the `finally` block.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Back up a second. So every time my program runs into an exception, it's going to stop whatever it's doing unless I specifically write code to catch it. How is that a good thing?**

A: **A:** One of the best things about exceptions is that they make it really obvious when you run into problems. Imagine how easy it could be in a complex application for you to lose track of all of the objects your program was working with. Exceptions call attention to your problems and help you root out their causes so that you always know that your program is doing what it's supposed to do.

Any time an exception occurs in your program, something you expected to happen didn't. Maybe an object reference wasn't pointing where you thought it was, or it was possible for a user to supply a value you hadn't considered, or a file you thought you'd be working with suddenly isn't available. If something like that happened and you didn't know it, it's likely that the output of your program would be wrong, and the behavior from that point on would be pretty different from what you expected when you wrote the program.

Now imagine that you had no idea the error had occurred and your users started calling you up with incorrect data and telling you that your program was unstable. That's why it's a *good* thing that exceptions disrupt everything your program is doing. They force you to deal with the problem while it's still easy to find and fix.

Q: **Q: OK, so then what's the difference between a *handled* exception and an *unhandled* exception?**

A: **A:** Whenever your program throws an exception, the runtime environment will search through your code looking for a `catch` block that handles it. If you've written one, the `catch` block will execute and do whatever you specified for that particular exception. Since you wrote a `catch` block to deal with that error up front, that exception is considered handled. If the runtime can't find a `catch` block to match the exception, it stops everything your program is doing and raises an error. That's an *unhandled* exception.

Q: **Q: But isn't it easier to use a catch-all exception? Isn't it safer to write code that always catches every exception?**

A: **A:** You should **always do your best to avoid catching `Exception`,** and instead catch specific exceptions. You know that old saying about how an ounce of prevention is better than a pound of cure? That's especially true in exception handling. Depending on catch-all exceptions is usually just a way to make up for bad programming. For example, you're often better off using `File.Exists()` to check for a file before you try to open it than catching a `FileNotFoundException`. While some exceptions are unavoidable, you'll find that a surprising number of them never have to be thrown in the first place. It's sometimes really useful to leave exceptions unhandled. Real-life programs have complex logic, and it's often difficult to recover correctly when something goes wrong, especially when a problem occurs very far down in the program. By only handling specific exceptions, avoiding catch-all exception handlers, and letting those exceptions bubble up to get caught on a top level, you end up with a more robust app because it will be immediately obvious if there's a problem.

NOTE

A system that's designed to immediately report a failure (rather than slowly becoming unstable) is sometimes referred to as "fail-fast."

Q: **Q: What happens when you have a `catch` that doesn't specify a particular exception?**

A: **A:** A `catch` block like that will catch any kind of exception the `try` block can throw.

Q: **Q: If a `catch` block with no specified exception will catch anything, why would I ever want to specify an exception type?**

A: **A:** Certain exceptions might require different actions to keep your program moving. Maybe an exception caused by dividing by zero might have a `catch` block where it goes back and sets properties to save some of the data you've been working with, while a null reference exception in the same block of code might require it to create new instances of an object.

Q: **Q: Does all error handling happen in a `try`/`catch`/`finally` sequence?**

A: **A:** No. You can mix it up a bit. You could have **multiple `catch` blocks** if you wanted to deal with lots of different kinds of errors. You could also have no `catch` block at all. It's legal to have a `try`/`finally` block. That wouldn't handle any exceptions, but it would make sure that the code in the `finally` block ran even if you got stopped halfway through the `try` block. But we'll talk a lot more about that in a minute....

An unhandled exception means your program will run unpredictably. That's why the program stops whenever it runs into one.

POOL PUZZLE

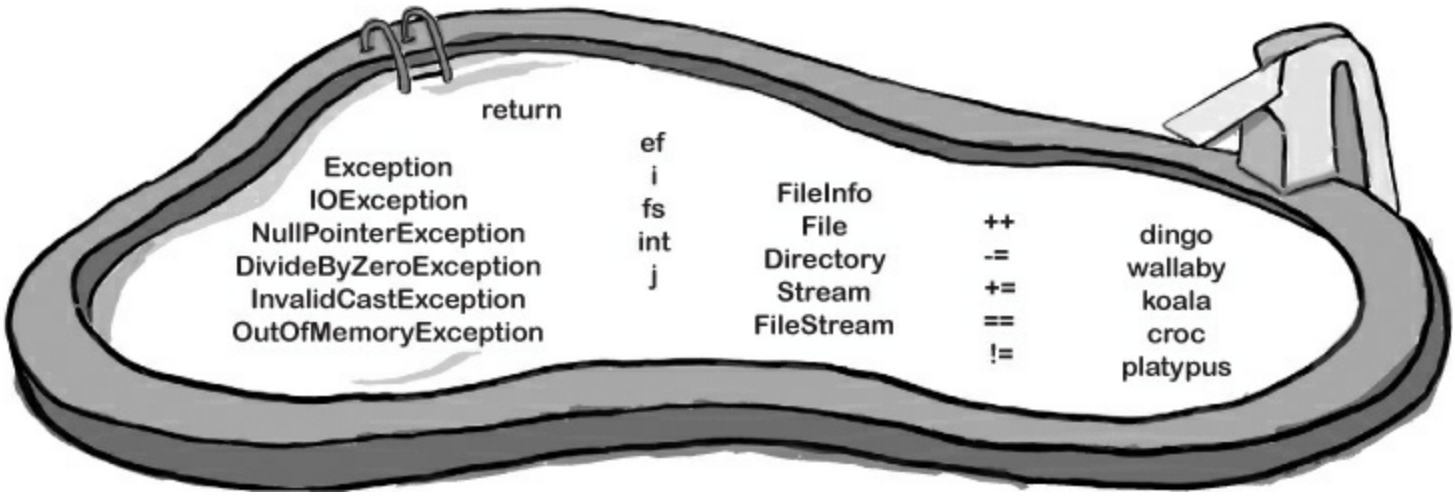
Your ***job*** is to take code snippets from the pool and place them into the blank lines in the program. You can use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make the program produce the output.

Output: `G'day Mate!`

```
using System.IO;
public static void Main() {
    Kangaroo joey = new Kangaroo();
```

```
    int koala = joey.Wombat(
           joey.Wombat(joey.Wombat(1)));
    try {
        Console.WriteLine((15 / koala)
                  + " eggs per pound");
    }
    catch (____________________) {
        Console.WriteLine("G'Day Mate!");
    }
}

class Kangaroo {
  ___________ fs;
  int croc;
  int dingo = 0;
  public int Wombat(int wallaby) {
     _______ __;
     try {
        if (________ > 0) {
           __ = _____.OpenWrite("wobbiegong");
           croc = 0;
        } else if (________ < 0) {
           croc = 3;
        } else {
           ___ = _____.OpenRead("wobbiegong");
           croc = 1;
        }
    }
    catch (IOException) {
         croc = -3;
    }
    catch {
         croc = 4;
    }
    finally {
       if (______ > 2) {
          croc ___ dingo;
       }
    }
    ________ ______;
    }
}
```

Note: Each snippet from the pool can be used more than once!



The pool puzzles are getting harder, and the names are getting more obscure to give you fewer hints. You'll really need to work through the problem! Remember, the puzzles are optional, so don't worry if you need to move on and come back to this one...but if you really want to get this stuff into your brain, these puzzles will do the trick!

POOL PUZZLE SOLUTION

```
using System.IO;

public static void Main() {
    Kangaroo joey = new Kangaroo();
    int koala = joey.Wombat(joey.Wombat(joey.Wombat(1)));
    try {
        Console.WriteLine((15 / koala) + " eggs per pound");
    }
    catch (DivideByZeroException) {
        Console.WriteLine("G'Day Mate!");
    }
}

class Kangaroo {
    FileStream fs;
    int croc;
    int dingo = 0;

    public int Wombat(int wallaby) {
        dingo ++;
        try {
            if (wallaby > 0) {
                fs = File.OpenWrite("wobbiegong");
                croc = 0;
            } else if (wallaby < 0) {
                croc = 3;
            } else {
                fs = File.OpenRead("wobbiegong");
                croc = 1;
            }
        }
        catch (IOException) {
            croc = -3;
        }
        catch {
            croc = 4;
        }
        finally {
            if (dingo > 2) {
                croc -= dingo;
            }
        }
        return croc;
    }
}
```

joey.Wombat() is called three times, and the third time it returns zero. That causes the WriteLine() to throw a DivideByZeroException.

This catch block only catches exceptions where the code divides by zero.

The clue that this is a FileStream is that it has an OpenRead() method and throws an IOException.

This code opens a file called "wobbiegong" and keeps it open the first time it's called. Later on, it opens the file again. But it never closed the file, which causes it to throw an IOException.

Remember, you should avoid catch-all exceptions in your code. But you should also avoid other things we do to make puzzles more interesting, like using obfuscated variable names.

You already know that you always have to close files when you're done with them. If you don't, the file will be locked open, and if you try to open it again it'll throw an IOException.

Use the Exception object to get information about the problem

We've been saying all along that .NET generates an `Exception` object when an exception is thrown. When you write your `catch` block, you have access to that object. Here's how it works:

- ❶ An object is humming along, doing its thing, when it encounters something unexpected and throws an exception.

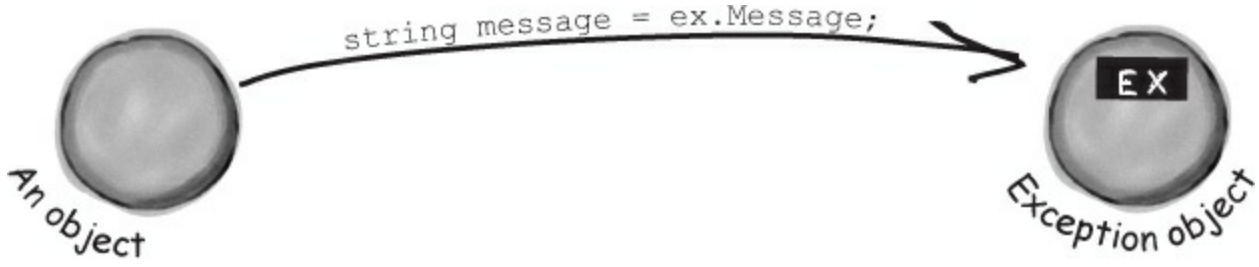
- ❷ Luckily, its `try`/`catch` block caught the exception. Inside the `catch` block, it gave the exception a name: **`ex`**.

If a statement inside the `DoSomethingRisky()` method throws an exception that isn't handled in the method, then it will be caught by the exception handler for the code that called it. If there's no exception handler there, the exception will keep moving up the call stack. If it gets to the top of the call stack without being handled, then you end up with an unhandled exception that causes your program to crash.

```
try {
      DoSomethingRisky();
}
catch (RiskyThingException ex) {
      string message = ex.Message;
      new MessageDialog("I took too many risks! "
                                    + message).ShowAsync();
}
```

When you specify a type of exception in the catch block, if you provide a variable name, then your code can use it to access the Exception object.

- ❸ The `Exception` object stays around until the `catch` block is done. Then the **`ex`** reference disappears, and it's eventually garbage-collected.


string message = ex.Message;
An object
EX
Exception object

Use more than one catch block to handle multiple types of exceptions

You can also call the exception's `ToString()` method to get a lot of the pertinent data into a dialog.

You know that you can catch a specific type of exception...but what if you write code where more than one problem can occur? In these cases, you may want to write code that handles each different type of exception. That's where using more than one `catch` block comes in. Here's an example from the code in the beehive nectar processing plant. You can see how it catches several kinds of exceptions. In some cases it uses properties in the `Exception` object. It's pretty common to use the `Message` property, which usually contains a description of the exception that was thrown. You can also call `throw;` to **rethrow** the exception, so it can be handled further up the call stack.

```
public void ProcessNectar(NectarVat vat, Bee worker, HiveLog log) {
    try {
        NectarUnit[] units = worker.EmptyVat(vat);
        for (int count = 0; count < worker.UnitsExpected, count++) {
            stream hiveLogFile = log.OpenLogFile();
            worker.AddLogEntry(hiveLogFile);
        }
    }
    catch (VatEmptyException) {
        vat.Emptied = true;
    }
    catch (HiveLogException ex) {
        throw;
    }
    catch (IOException ex) {
        worker.AlertQueen("An unspecified file error happened: "
            + "Message: " + ex.Message + "\r\n"
            + "Stack trace: " + ex.StackTrace + "\r\n"
            + "Data: " + ex.Data + "\r\n");
    }
    finally {
        vat.Seal();
        worker.FinishedJob();
    }
}
```

If you won't use the Exception object, there's no need to declare it.

When you have several catch blocks, they're examined in order. In this code, first it checks for a VatEmptyException and then a HiveLogException. The last catch block catches IOException. That's the base class for several different file exceptions, including FileNotFoundException and EndOfStreamException.

Sometimes you want to bubble an exception up to the method that called this one by using throw; to **rethrow** the exception.

This catch block assigns the exception to the variable ex, which it can use to get information from the Exception object.

It's fine for two blocks to use the same name ("ex") for the Exception.

This statement uses three properties in the Exception object: Message, which has the message you'd normally see in the exception window in the IDE ("Attempted to divide by zero"); StackTrace, which gives you a summary of the call stack; and Data, which sometimes contains pertinent data that's associated with the exception.

One class throws an exception that a method in another class can catch

NOTE

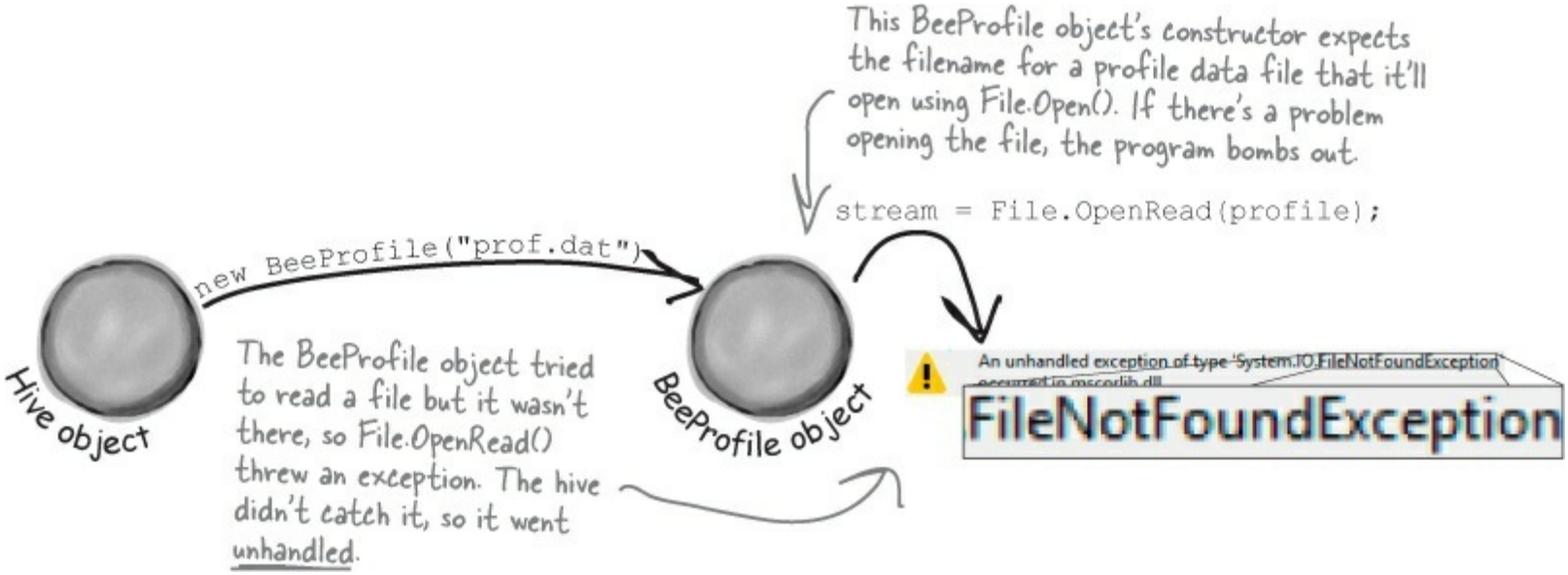
...or another method in the same class.

When you're building a class, you don't always know how it's going to be used. Sometimes other people will end up using your objects in a way that causes problems — and sometimes you do it yourself ! That's where exceptions come in.

The whole point behind throwing an exception is to see what might go wrong, so you can put in place some sort of contingency plan. You don't usually see a method that throws an exception and then catches it. An exception is usually thrown in one method and then caught in a totally different one — usually in a different object.

Instead of this...

Without good exception handling, one exception can halt the entire program. Here's how it would work in a program that manages bee profiles for a queen bee.



...we can do this.

The `BeeProfile` object can intercept the exception and add a log entry. Then it can turn around and throw the exception back to the hive, which catches it and recovers gracefully.

Notice how the BeeProfile object **intercepts** the exception, logs it using its WriteLogEntry() method, and then throws it again so it's passed along to the hive.

```
try {
   stream = File.OpenRead(profile);
} catch (FileNotFoundException ex) {
   WriteLogEntry("unable to find " +
      profile + ": " + ex.Message();
   throw;
}
```

```
try {
   prof = new BeeProfile("prof.dat");
} catch (FileNotFoundException) {
   Hive.RecreateBeeProfile("prof.dat");
}
```

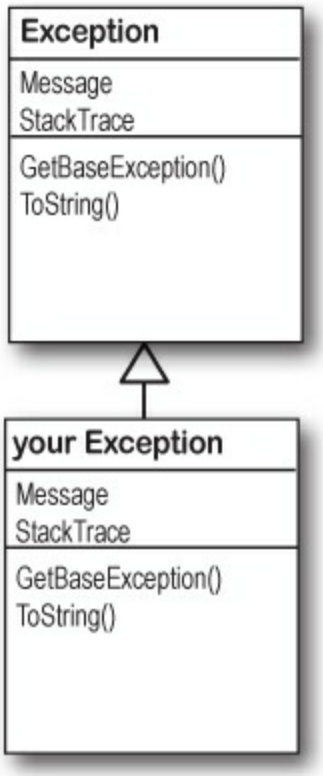
Now when the hive tries to create a new BeeProfile object by passing it an invalid filename, it can trust BeeProfile to log the error and then alert it to the problem by throwing an exception. The hive can catch the exception and take some corrective action—in this case, recreating the bee profile.

Bees need an OutOfHoney exception

Your classes can throw their own exceptions. For example, if you get a null parameter in a method that was expecting a value, it's pretty common to throw the same exception that a .NET method would:

```
throw new ArgumentException();
```

Your methods can throw this exception if they get invalid or unexpected values in their parameters.



But sometimes you want your program to throw an exception because of a special condition that could happen when it runs. The bees we created in the hive, for example, consume honey at a different rate depending on their weight. If there's no honey left to consume, it makes sense to have the hive throw an exception. You can create a custom exception to deal with that specific error condition just by creating your own class that inherits from `Exception` and then throwing the exception whenever you encounter a specific error.

```
class OutOfHoneyException : System.Exception {
    public OutOfHoneyException(string message) : base(message) { }
}
class HoneyDeliverySystem {
    ...
    public void FeedHoneyToEggs() {
        if (honeyLevel == 0) {
            throw new OutOfHoneyException("The hive is out of honey.");
        } else {
            foreach (Egg egg in Eggs) {
                ...
}
public partial class Form1 : Form {
...
   private void consumeHoney_Click(object sender, EventArgs e) {
      HoneyDeliverySystem delivery = new HoneyDeliverySystem();
      try {
          delivery.FeedHoneyToEggs()
      }
      catch (OutOfHoneyException ex){
          MessageBox.Show(ex.Message, "Warning: Resetting Hive");
          Hive.Reset();
      }
   }
}
```

You need to create a class for your exception and make sure that it inherits from System.Exception. Notice how we're overloading the constructor so we can pass an exception message.

This throws a new instance of the exception object.

If there's honey in the hive, the exception will never get thrown and this code will run.

You can catch a custom exception by name just like any other exception, and do whatever you need to do to handle it.

In this case, if the hive is out of honey none of the bees can work, so the simulator can't continue. The only way to keep the program working once the hive runs out of honey is to reset it, and we can do that by putting the code to reset it in the catch block.

EXCEPTION MAGNETS

Arrange the magnets so the application writes the output to the console.

```
public static void Main() {
    Console.Write("when it ");
    ExTestDrive.Zero("yes");
    Console.Write(" it ");
    ExTestDrive.Zero("no");
    Console.WriteLine(".");
}

class MyException : Exception { }
```

output:

when it thaws it throws.

```
}
}
}
```

```
if (t == "yes") {
```

```
Console.Write("a");
Console.Write("o");
Console.Write("t");
```

```
Console.Write("w");
Console.Write("s");
```

```
try {
} catch (MyException) {
throw new MyException();
} finally {
```

```
DoRisky(test);
```

```
        Console.Write("r");
    }
}
class ExTestDrive {
    public static void Zero(string test) {
static void DoRisky(String t) {
    Console.Write("h");
```

EXCEPTION MAGNETS SOLUTION

Arrange the magnets so the application writes the output to the console.

```
public static void Main() {
    Console.Write("when it ");
    ExTestDrive.Zero("yes");
    Console.Write(" it ");
    ExTestDrive.Zero("no");
    Console.WriteLine(".");
}

class MyException : Exception { }
```

output:

when it thaws it throws.

This line defines a custom exception called MyException, which gets caught in a catch block in the code.

```
class ExTestDrive {
   public static void Zero(string test) {
      try {
         Console.Write("t");
         DoRisky(test);
         Console.Write("o");
      } catch (MyException) {
         Console.Write("a");
      } finally {
         Console.Write("w");
      }
      Console.Write("s");
   }

   static void DoRisky(String t) {
      Console.Write("h");
      if (t == "yes") {
         throw new MyException();
      }
      Console.Write("r");
   }
}
```

The Zero() method either prints "thaws" or "throws", depending on whether it was passed "yes" or something else as its test parameter.

The finally block makes sure that the method always prints "w". And the "s" is printed outside the exception handler, so it always prints, too.

This line only gets executed if DoRisky() doesn't throw the exception.

The DoRisky() method only throws an exception if it's passed the string "yes".

BULLET POINTS

- Any statement can throw an exception if something fails at runtime.
- Use a `try`/`catch` block to handle exceptions. Unhandled exceptions will cause your program to stop execution and pop up an error window.
- Any exception in the block of code after the `try` statement will cause the program's execution to immediately jump to the first statement in the block of code after `catch`.
- The `Exception` object gives you information about the exception that was caught. If you specify an `Exception` variable in your `catch` statement, that variable will contain information about any exception thrown in the `try` block:

```
try {
  // statements that might
  // throw exceptions
} catch (IOException ex) {
  // if an exception is thrown,
  // ex has information about it
}
```

- There are many different kinds of exceptions that you can catch. Each has its own object that inherits from `Exception`. Really try to avoid just catching `Exception` — catch specific exceptions instead.
- Each `try` can have more than one `catch`:

```
try { ... }
catch (NullReferenceException ex) {
   // these statements will run if a
   // NullReferenceException is thrown
}
catch (OverflowException ex) { ... }
catch (FileNotFoundException) { ... }
catch (ArgumentException) { ... }
```

- Your code can throw an exception using `throw`:

```
throw new Exception("Exception message");
```

- Your code can also **rethrow** an exception using `throw;` but this only works inside of a `catch` block. Rethrowing an exception preserves the call stack.
- You can create a custom exception by inheriting from the `Exception` base class.

```
class CustomException : Exception { }
```

- Most of the time, you only need to throw exceptions that are built into .NET, like `ArgumentException`. The reason you use different kinds of exceptions is so that you can **give more information to your users**. Popping up a window with the text "An unknown error has occurred" is not nearly as useful as an error message that says "The excuse folder is empty. Please select a different folder if you want to read excuses."

An easy way to avoid a lot of problems: using gives you try and finally for free

> **NOTE**
>
> Remember, when you declare a reference in a "using" statement, its Dispose() method is automatically called at the end of the block.

You already know that `using` is an easy way to make sure that your files always get closed. But what you didn't know is that it's really **just a C# shortcut** for `try` and `finally`!

```
using (YourClass c
          = new YourClass() ) {

    // code

}
```

is like this

```
YourClass c = new YourClass();

try {

    // code

} finally {

    c.Dispose();
}
```

When you use a using statement, you're taking advantage of finally to make sure its Dispose() method is always called.

Exception avoidance: implement IDisposable to do your own cleanup

> **NOTE**
>
> IDisposable is a really effective way to avoid common exceptions and problems. Make sure you use using statements any time you're working with any class that implements it.

Streams are great, because they already have code written to close themselves when the object is disposed. But what if you have your own custom object, and it always needs to do something when it's disposed of ? Wouldn't it be great if you could write your own code that got run if your object was used in a `using` statement?

C# lets you do just that with the `IDisposable` interface. Implement `IDisposable`, and write your cleanup code in the `Dispose()` method, like this:

> **NOTE**
>
> You can only use a class in a using statement if it implements IDisposable; otherwise, your program won't compile.

Your object must implement IDisposable if you want to use your object within a using statement.

```
class Nectar : IDisposable {
  private double amount;
  private BeeHive hive;
  private Stream hiveLog;
  public Nectar(double amount, BeeHive hive, Stream hiveLog) {
    this.amount = amount;
    this.hive = hive;
    this.hiveLog = hiveLog;
  }
  public void Dispose() {
    if (amount > 0) {
      hive.Add(amount);
      hive.WriteLog(hiveLog, amount + " mg nectar added to the hive");
      amount = 0;
    }
  }
}
```

The IDisposable interface only has one member: the Dispose() method. Whatever you put in this method will get executed at the end of the using statement...or whenever Dispose() is called manually.

This Dispose() method was written so it could be called many times, not just once.

This particular code empties any remaining nectar into the hive and logs a message. It's important, and must happen, so we put it in the Dispose() method.

One of the guidelines for implementing `IDispose` is that your `Dispose()` method can be called multiple times without side effects. Can you think of why that's an important guideline?

We can use multiple `using` statements now. First, let's use a built-in object, `Stream`, which implements `IDisposable`. Then, we'll work with our updated `Nectar` object, which also implements `IDisposable`:

You'll see nested using statements like this when you need to declare two IDisposable references in the same block of code.

```
using (Stream log = new File.OpenWrite("log.txt"))
using (Nectar nectar = new Nectar(16.3, hive, log)) {
      Bee.FlyTo(flower);
      Bee.Harvest(nectar);
      Bee.FlyTo(hive);
}
```

The Nectar object uses the log stream, which will close automatically at the end of the outer using statement.

Then the Bee object uses the Nectar object, which will add its nectar to the hive automatically at the end of the inner using statement.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Is it possible to use an object with a `using` statement if it doesn't implement `IDisposable`?**

A: **A:** No, you can only create objects that implement `IDisposable` with `using` statements, because they're tailor-made for each other. Adding a `using` statement is just like creating a new instance of a class, except that it always calls its `Dispose()` method at the end of the block. That's why the class **must implement** the `IDisposable` interface.

Q: **Q: Can you put any statement inside a `using` block?**

A: **A:** Definitely. The whole idea with `using` is that it helps you make sure that every object you create with it is disposed. But what you do with those objects is entirely up to you. In fact, you can create an object with a `using` statement and never even use it inside the block. But that would be pretty useless, so we don't recommend doing that.

Q: **Q: Can you call `Dispose()` outside of a `using` statement?**

A: **A:** Yes. You don't ever actually ***need*** to use a `using` statement. You can call `Dispose()` yourself when you're done with the object. Or you can do whatever cleanup is necessary — like calling a stream's `Close()` method manually. But if you use a `using` statement, it'll make your code easier to understand and prevent problems that happen if you don't dispose of your objects.

Q: **Q: You mentioned a "`try`/`finally`" block. Does that mean it's OK to have a `try` and `finally` without a `catch`?**

A: **A:** Yes! You can definitely have a `try` block without a `catch`, and just a `finally`. It looks like this:

```
try {
     DoSomethingRisky();
     SomethingElseRisky();
}
finally {
     AlwaysExecuteThis();
}
```

If `DoSomethingRisky()` throws an exception, then the `finally` block will immediately run.

Q: **Q: Does `Dispose()` only work with files and streams?**

A: **A:** No, there are a lot of classes that implement `IDisposable`, and when you're using one you should always use a `using` statement. (You'll see some of them in the next few chapters.) And if you write a class that has to be disposed of in a certain way, then you can implement `IDisposable`, too.

You want to know what type of exception is thrown, so you can handle that exception.

There's more to exception handling than just printing out a generic error message. For instance, in the excuse finder, if we know we've got a `FileNotFoundException`, we might print an error that suggested where the right files should be located. If we have an exception related to databases, we might send an email to the database administrator. All that depends on you catching *specific* exception types.

NOTE

This is why there are so many classes that inherit from Exception, and why you may even want to write your own classes to inherit from Exception.

The worst catch block EVER: catch-all plus comments

A `catch` block will let your program keep running if you want. An exception gets thrown, you catch the exception, and instead of shutting down and giving an error message, you keep going. But sometimes, that's not such a good thing.

Take a look at this `Calculator` class, which seems to be acting funny all the time. What's going on?

```
class Calculator {

...

  public void Divide(int dividend, int divisor) {

    try {

      this.quotient = dividend / divisor;

    } catch {

      // Note from Jim: we need to figure out a way to prevent

      // people from entering in zero in a division problem.

    }

  }

}
```

Here's the problem. If divisor is zero, this will create a DivdeByZeroException.

But there's a catch block. So why are we still getting errors?

The programmer thought that he could **bury** his exceptions by using an empty catch block, but he just caused a headache for whoever had to track down problems with it later.

You should handle your exceptions, not bury them

Just because you can keep your program running doesn't mean you've *handled* your exceptions. In the code above, the calculator won't crash...at least, not in the `Divide()` method. But what if some other code calls that method, and tries to print the results? If the divisor was zero, then the method probably returned an incorrect (and unexpected) value.

Instead of just adding a comment and burying the exception, you need to **handle the exception**. And if you're not able to handle the problem, ***don't leave empty or commented catch blocks!*** That just makes it harder for someone else to track down what's going on. It's better to let the program continue to throw exceptions, because then it's easy to figure out what's going wrong.

NOTE

Remember, when your code doesn't handle an exception, the exception bubbles up the call stack. Letting an exception bubble up is a perfectly valid way of dealing with an exception, and in some cases it makes more sense to do that than to use a try/catch block to handle the exception.

Temporary solutions are OK (temporarily)

NOTE

...but in real life, "temporary" solutions have a nasty habit of becoming permanent.

Sometimes you find a problem, and know it's a problem, but aren't sure what to do about it. In these cases, you might want to log the problem and note what's going on. That's not as good as handling the exception, but it's better than doing nothing.

Here's a temporary solution to the calculator:

Take a minute and think about this `catch` block. What happens if the `StreamWriter` can't write to the *C:\Logs* folder? You can nest another `try/catch` block inside it to make it less risky. Can you think of a better way to do this?

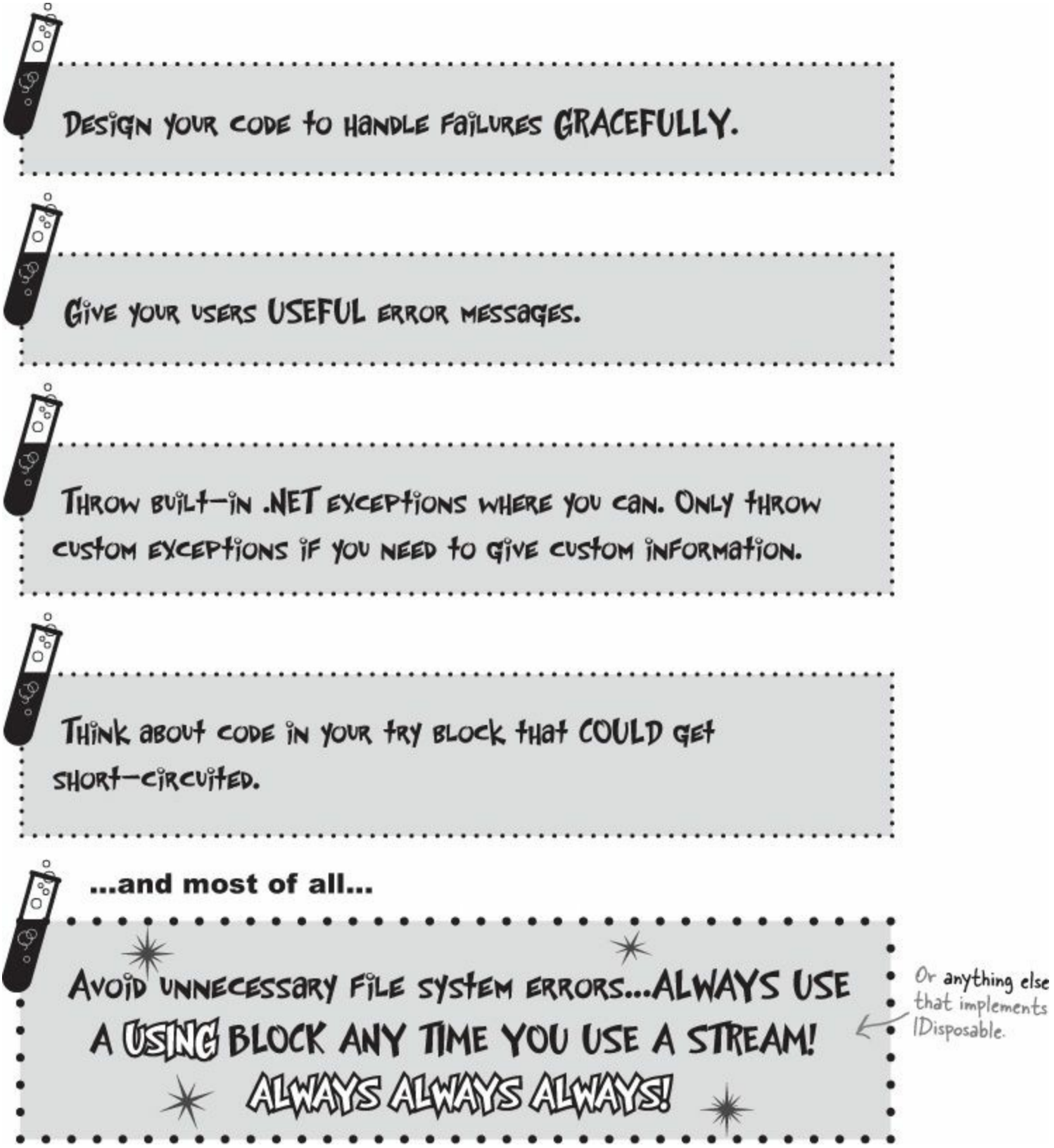
```
class Calculator {
...
  public void Divide(int dividend, int divisor) {
    try {
      this.quotient = dividend / divisor;
    } catch (Exception ex) {
      using (StreamWriter sw = new StreamWriter(@"C:\Logs\errors.txt");
         sw.WriteLine(ex.getMessage());
      };
    }
  }
}
```

This still needs to be fixed, but short-term, this makes it clear where the problem occurred. Still, wouldn't it be better to figure out why your Divide method is being called with a zero divisor in the first place?

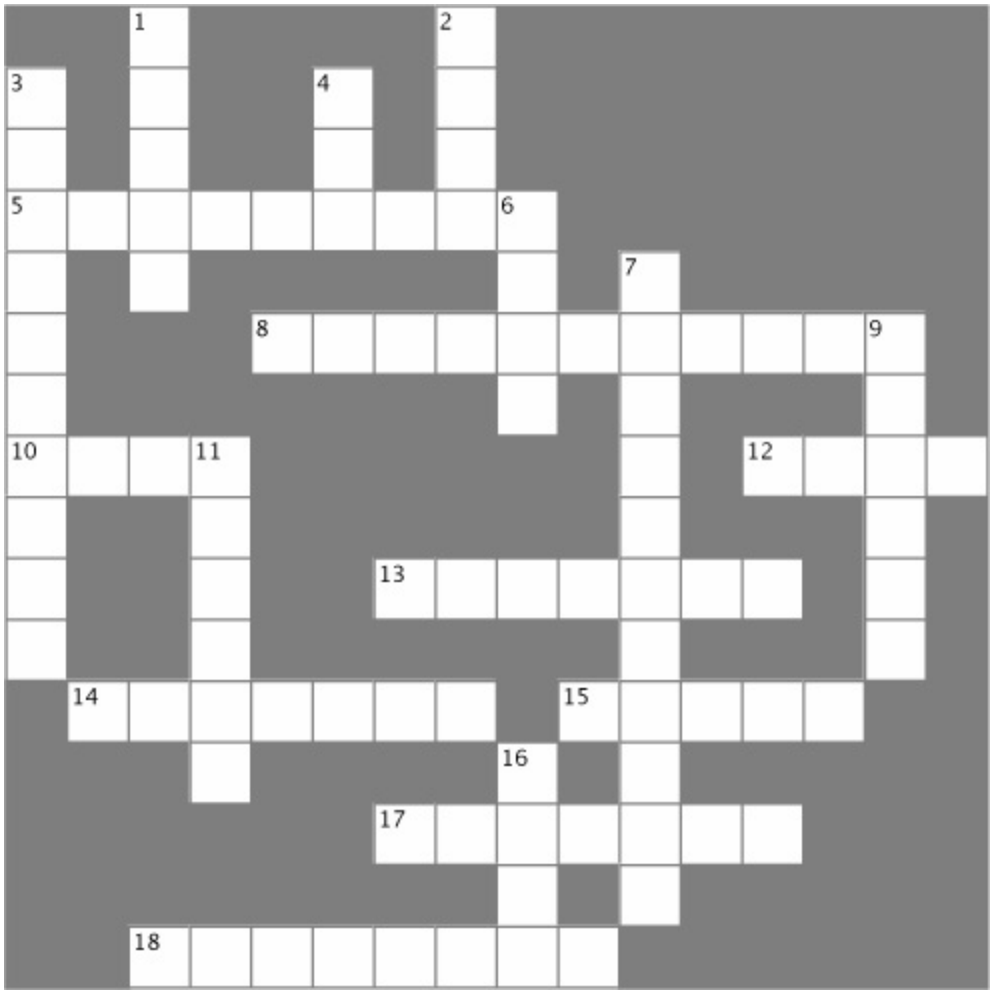
Handling exceptions doesn't always mean the same thing as FIXING exceptions.

It’s never good to have your program bomb out. But it’s way worse to have no idea why it’s crashing or what it’s doing to users’ data. That’s why you need to be sure that you’re always dealing with the errors you can predict and logging the ones you can’t. But while logs can be useful for tracking down problems, preventing those problems in the first place is a better, more permanent solution.

A few simple ideas for exception handling

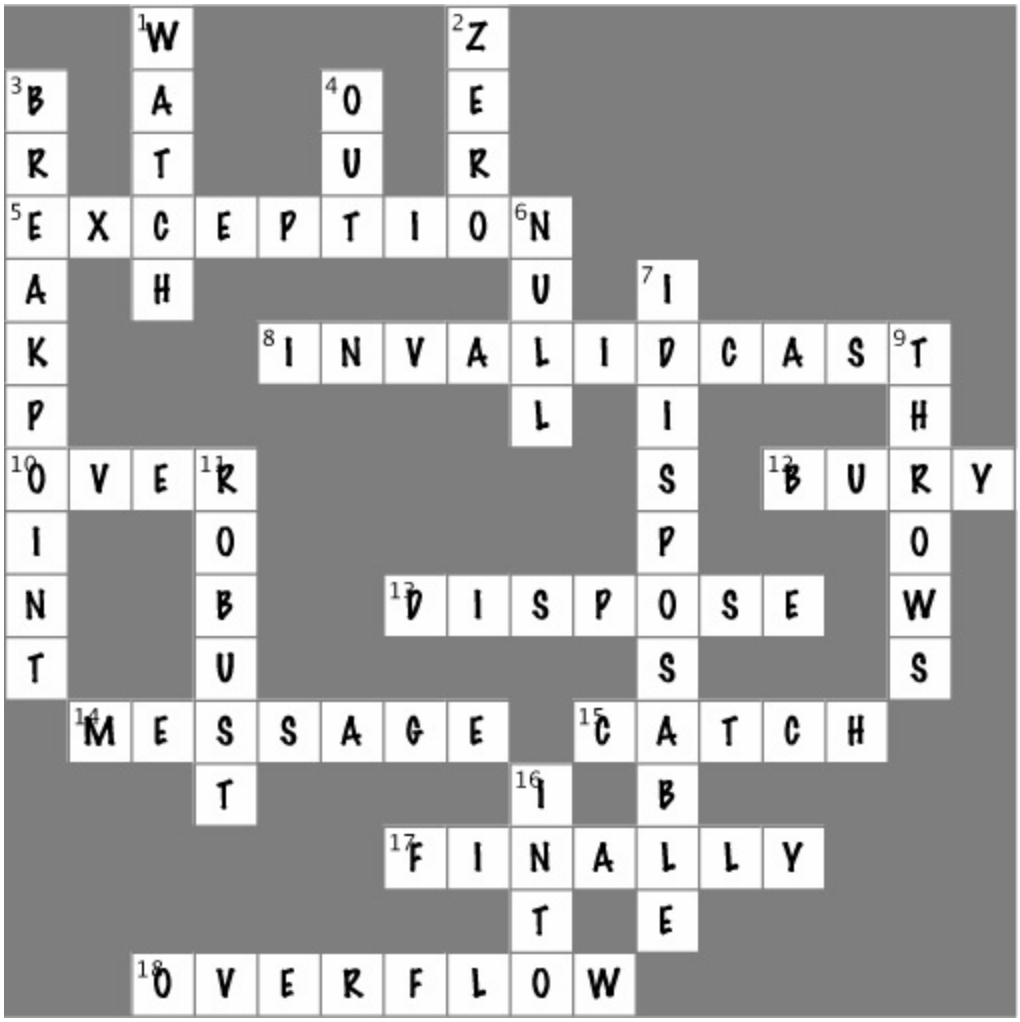


EXCEPTIONCROSS



Across	Down
5. The base class that `DivideByZeroException` and `FormatException` inherit from 8. An _____exception happens when you try to cast a value to a variable that can't hold it 10. If the next statement is a method, "Step _____" tells the debugger to execute all the statements in the method and break immediately afterward 12. If you ____ your exceptions, it can make them hard to track down 13. This method is always called at the end of a `using` block 14. The field in the `Exception` object that contains a string with a description 15. One `try` block can have multiple _____ blocks 17. The _____ block contains any statements that absolutely must be run after an exception is handled 18. An _____exception means you tried to cram a number that was too big into a variable that couldn't hold it	1. The window in the IDE that you can use to check your variables' values 2. You'll get an exception if you try to divide by this 3. Toggle this if you want the debugger to stop execution when it hits a specific line of code 4. "Step ____" tells the debugger to execute the rest of the statements in the current method and then break 6. What a reference contains if it doesn't point to anything 7. You can only declare a variable with a `using` statement if it implements this interface 9. When a statement has a problem, it _____ an exception 11. A program that handles errors well 16. If the next statement is a method, "Step _____" tells the debugger to execute the first statement in that method

EXCEPTIONCROSS SOLUTION



Brian finally gets his vacation...

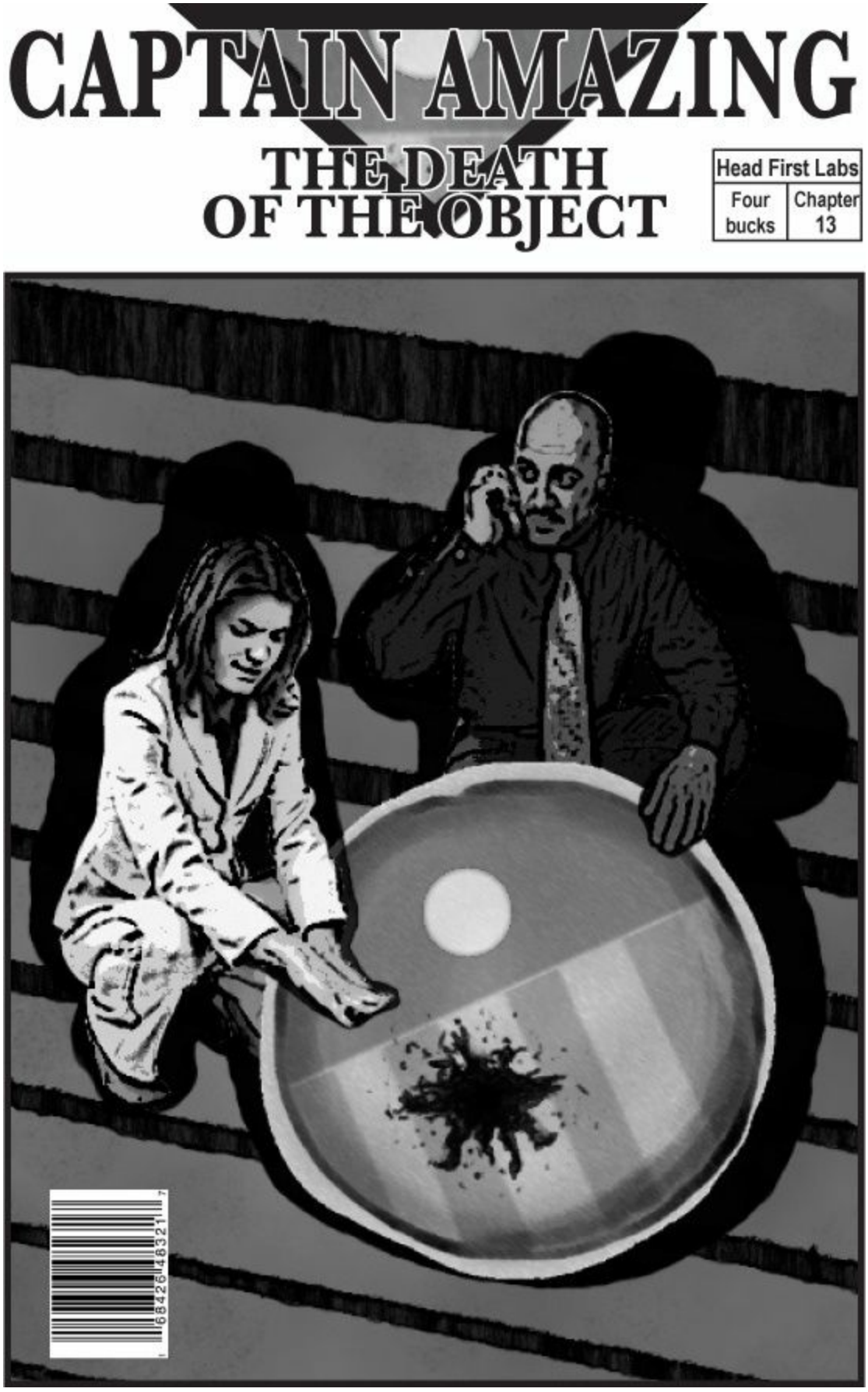
Now that Brian's got a handle on his exceptions, his job's going smoothly and he can take that well-deserved (and boss-approved!) vacation day.

...and things are looking up back home!

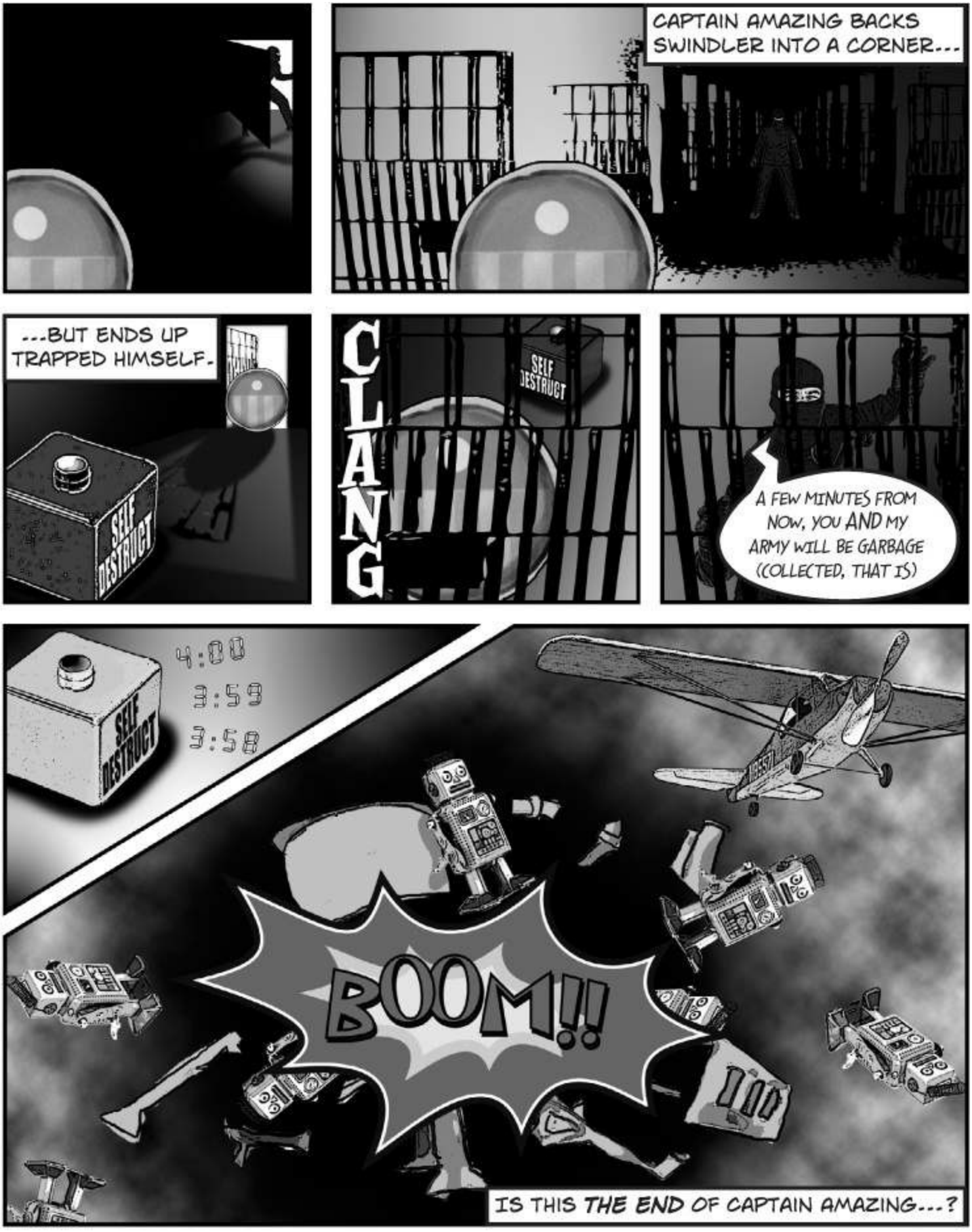
Your exception handling skills did more than just prevent problems. They ensured that Brian's boss has no idea anything went wrong in the first place!

Good exception handling is invisible to your users. The program never crashes, and if there are problems, they are handled gracefully, without confusing error messages.

Chapter 13. Captain Amazing: The Death of the Object



CAPTAIN AMAZING, OBJECTVILLE'S MOST AMAZING OBJECT, PURSUES HIS ARCH-NEMESIS...
I'VE GOT YOU NOW, SWINDLER.
YOU'RE TOO LATE! AS WE SPEAK MY CLONE ARMY IS GATHERING IN THE FACTORY BENEATH US...
...READY TO WREAK HAVOC ON THE STREETS OF OBJECTVILLE!
Welcome to
Objectville
Home of Polymorphism
POW!!
I'LL TAKE DOWN EACH CLONE'S REFERENCES, ONE BY ONE.



SHARPEN YOUR PENCIL

Below is the code detailing the fight between Captain Amazing and Swindler (not to mention his clone army). Your job is to draw out what's going on in memory when the `FinalBattle` class is instantiated.

```
class FinalBattle {
  public CloneFactory Factory = new CloneFactory();
  public List<Clone> Clones = new List<Clone>() { ... };
  public SwindlersEscapePlane escapePlane;

  public FinalBattle() {
    Villain swindler = new Villain(this);
    using (Superhero captainAmazing = new Superhero()) {
      Factory.PeopleInFactory.Add(captainAmazing);
      Factory.PeopleInFactory.Add(swindler);
      captainAmazing.Think("I'll take down each clone's reference,
                            one by one");
      captainAmazing.IdentifyTheClones(Clones);
      captainAmazing.RemoveTheClones(Clones);
      swindler.Think("A few minutes from now, you AND my army will be garbage");
      swindler.Think("(collected, that is!)");
      escapePlane = new SwindlersEscapePlane(swindler);
      swindler.TrapCaptainAmazing(Factory);
      new MessageDialog("The Swindler escaped.").ShowAsync();
    }
  }
}
[Serializable]
class Superhero : IDisposable {
      private List<Clone> clonesToRemove = new List<Clone>();
      public void IdentifyTheClones(List<Clone> clones) {
          foreach (Clone clone in clones)
              clonesToRemove.Add(clone);
      }
      public void RemoveTheClones(List<Clone> clones) {
          foreach (Clone clone in clonesToRemove)
              Clones.Remove(clone);
       ...
      }
      ...
}
class Villain {
  private FinalBattle finalBattle;
  public Villain(FinalBattle finalBattle) {
    this.finalBattle = finalBattle;
  }
  public void TrapCaptainAmazing(CloneFactory factory) {
    factory.SelfDestruct.Tick += new EventHandler(SelfDestruct_Tick);
    factory.SelfDestruct.Interval = TimeSpan.FromSeconds(60);
    factory.SelfDestruct.Start();
  }
  private void SelfDestruct_Tick(object sender, EventArgs e) {
    finalBattle.Factory = null;
  }
}
```

You can assume that Clones was set using a collection initializer.

We've gotten you started here, with what's going on in the factory object.

2 Draw what's going on right here, when the SwindlersEscapePlane object is instantiated.

3 Draw a picture of what the heap will look like exactly one second after the FinalBattle constructor runs.

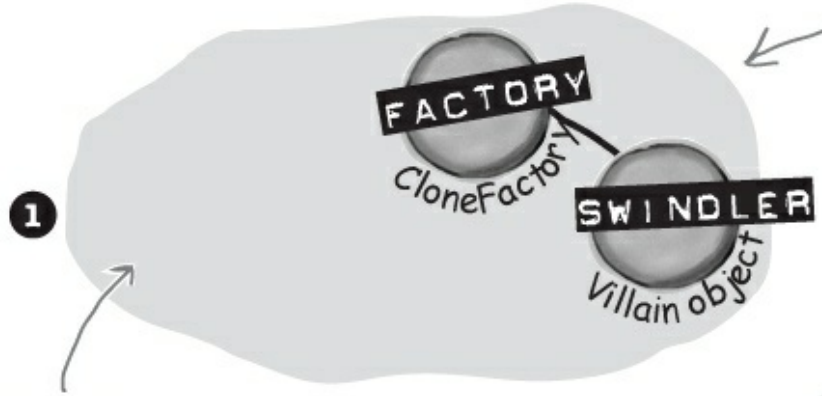
There's a Clone class that we're not showing you in this code, too. You don't need it to answer the questions.

There's more code here (including the Dispose() method to implement IDisposable) that we aren't showing you, but you don't need it to answer this.

```
class SwindlersEscapePlane {
  public Villain PilotsSeat;
  public SwindlersEscapePlane(Villain escapee) {
    PilotsSeat = escapee;
  }
}

class CloneFactory {
  public DispatcherTimer SelfDestruct = new DispatcherTimer();
  public List<object> PeopleInFactory = new List<object>();
  ...
}
```

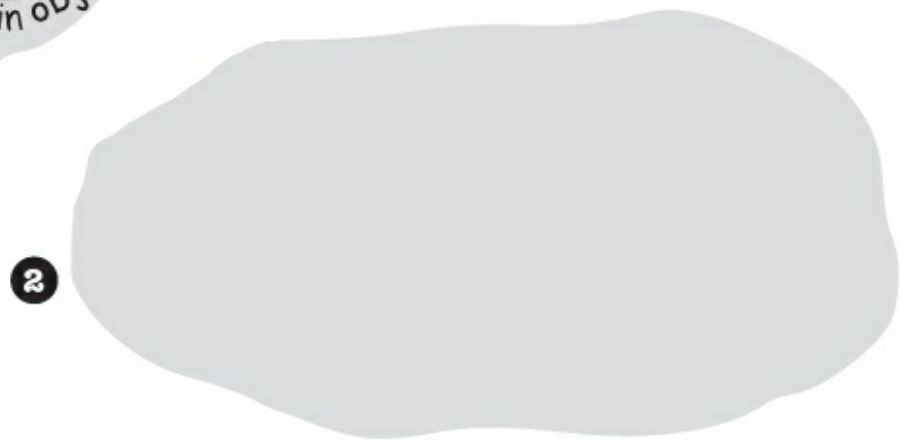
Make sure you add labels to your objects to show the reference variables that are pointing to them.

We started the first one for you. Make sure you draw in lines showing the architecture—we drew a line from the clone factory to the Villain object, because the factory has references to it (via its PeopleInFactory field).

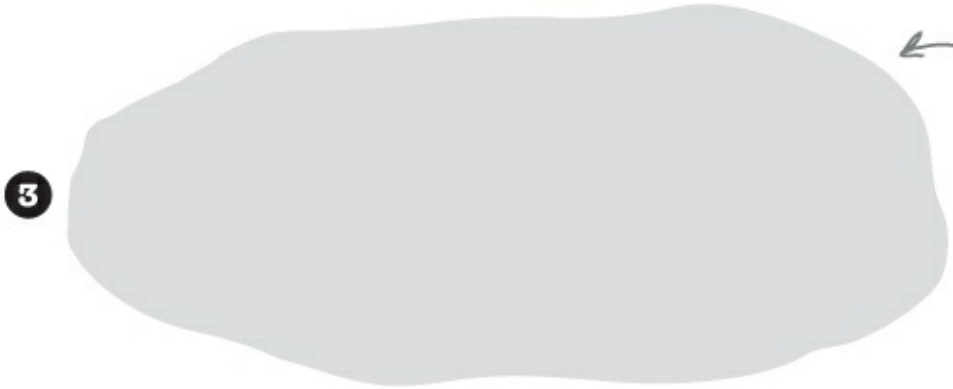


We've left space, as there is more to be drawn at this stage.

Don't worry about drawing the Clone and List objects—just add the objects for the Captain, the Swindler, the clone factory, and Swindler's escape plane.



Your job is to draw what's going on in these two bits of memory, too.



Based on your diagrams, where in the code did Captain Amazing die?

_____

_____

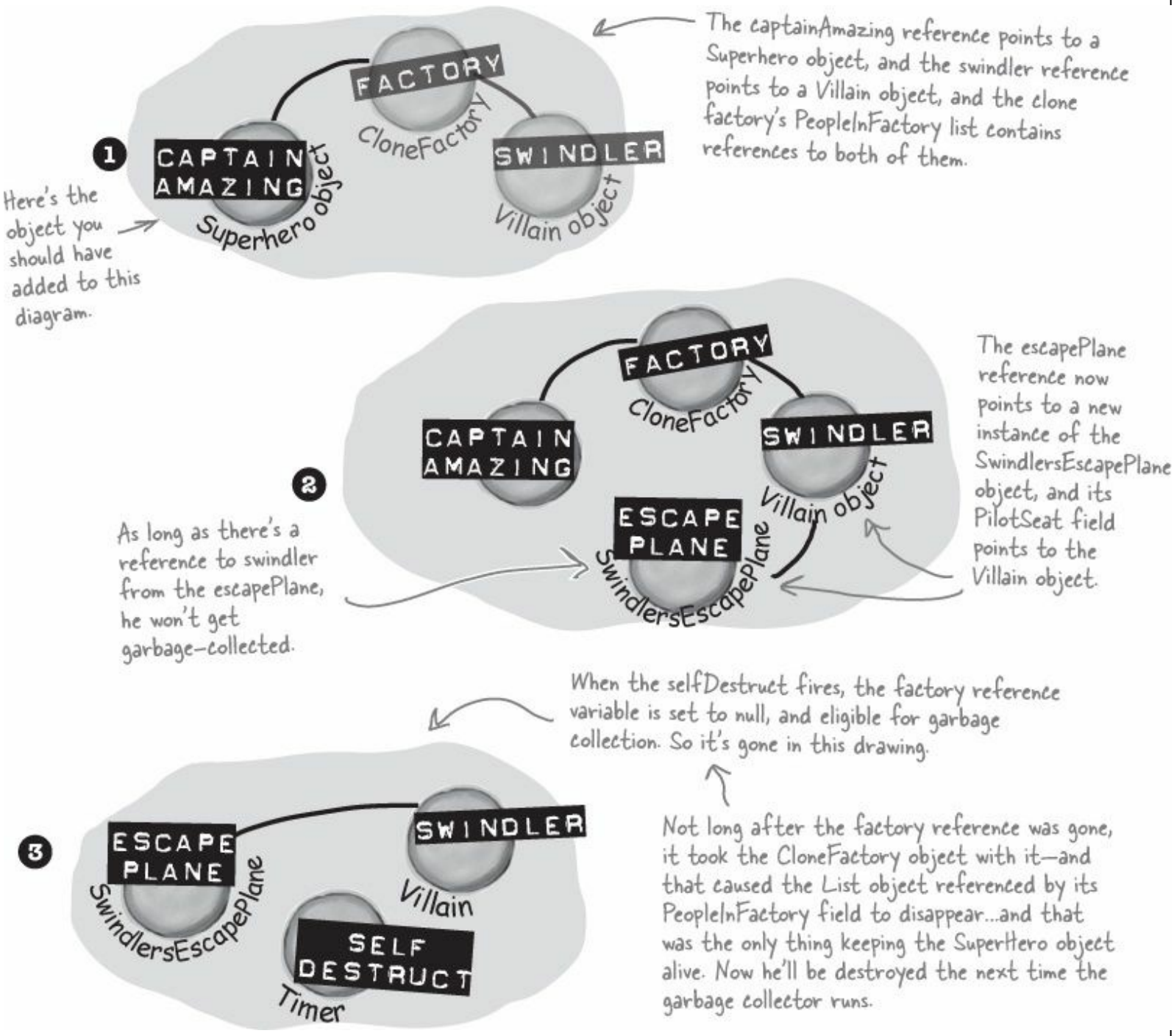
_____

_____

Be sure to annotate that on your diagram, too.

SHARPEN YOUR PENCIL SOLUTION

Draw what's happening in memory with the `FinalBattle` program.



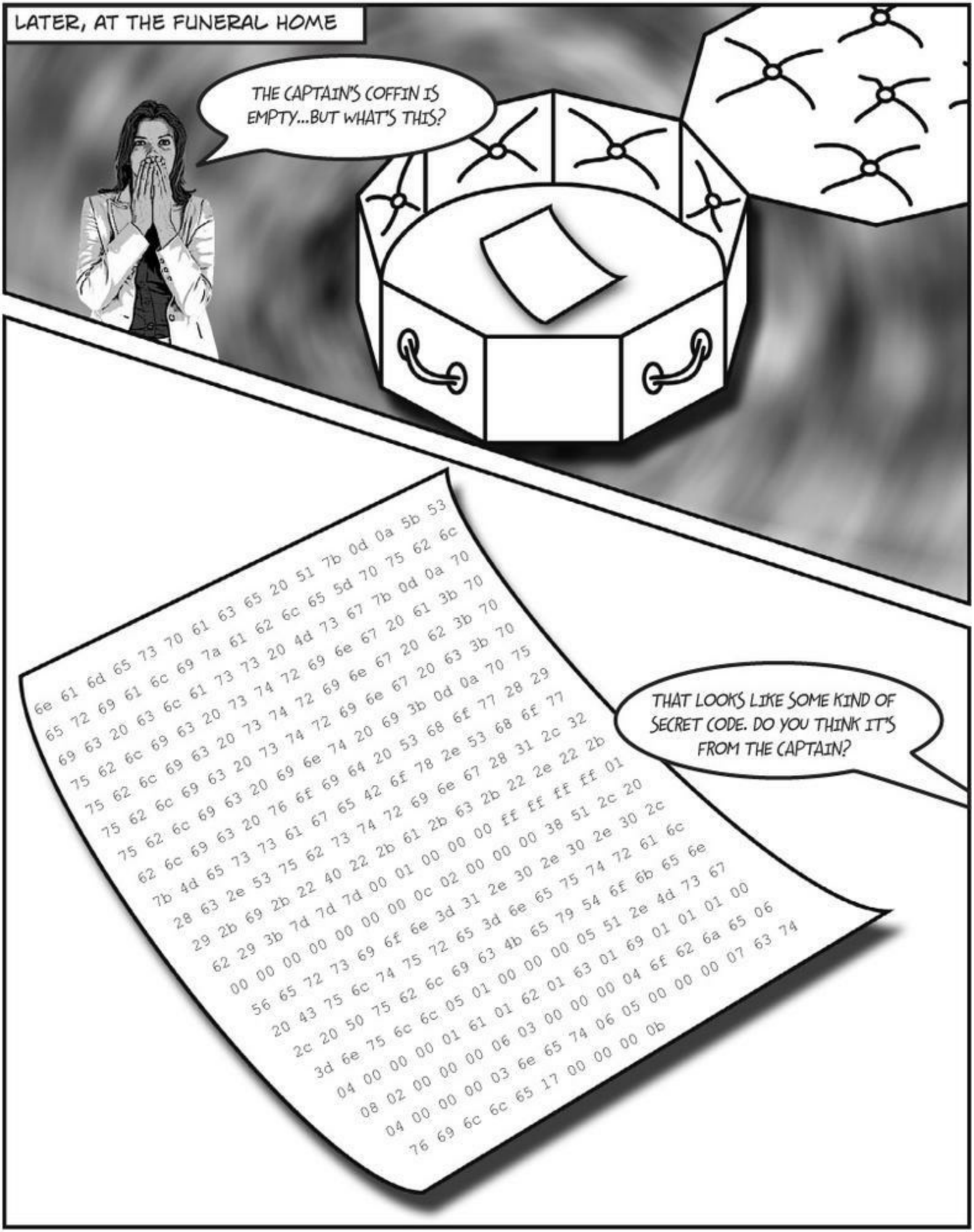
Based on your diagrams, where in the code did Captain Amazing die?

```
void SelfDestruct_Tick(object sender, EventArgs e) {
  finalBattle.factory = null;
}
```

Not long after the FinalBattle constructor ran, the hero was gone.

Once `finalBattleFactory` was set to null, it was ready for garbage collection. And it took the last reference to the Captain with it!

> **NOTE**
>
> Once the Superhero instance had no clone factory referencing it, it was marked for garbage collection too.


LATER, AT THE FUNERAL HOME
THE CAPTAIN'S COFFIN IS EMPTY...BUT WHAT'S THIS?
6e 61 6d 65 73 70 61 63 65 20 51 7b 0d 0a 5b 53
65 72 69 61 6c 69 7a 61 62 6c 65 5d 70 75 62 6c
69 63 20 63 6c 61 73 73 20 4d 73 67 7b 0d 0a 70
75 62 6c 69 63 20 73 74 72 69 6e 67 20 61 3b 70
75 62 6c 69 63 20 73 74 72 69 6e 67 20 62 3b 70
75 62 6c 69 63 20 73 74 72 69 6e 67 20 63 3b 70
75 62 6c 69 63 20 73 74 72 69 6e 67 20 69 3b 0d 0a 70 75
75 62 6c 69 63 20 69 6e 74 20 69 3b 0d 0a 70 75
62 6c 69 63 20 76 6f 69 64 20 53 68 6f 77 28 29
7b 4d 65 73 73 61 67 65 42 6f 78 2e 53 68 6f 77
28 63 2e 53 75 62 73 74 72 69 6e 67 28 31 2c 32
29 2b 69 2b 22 40 22 2b 61 2b 63 2b 22 2e 22 2b
62 29 3b 7d 7d 00 01 00 00 00 ff ff ff ff 01
00 00 00 00 00 00 0c 02 00 00 00 38 51 2c 20
56 65 72 73 69 6f 6e 3d 31 2e 30 2e 30 2e 30 2c
20 43 75 6c 74 75 72 65 3d 6e 65 75 74 72 61 6c
2c 20 50 75 62 6c 69 63 4b 65 79 54 6f 6b 65 6e
3d 6e 75 6c 6c 05 01 00 00 00 05 51 2e 4d 73 67
04 00 00 00 01 61 01 62 01 63 01 69 01 01 01 00
08 02 00 00 00 06 03 00 00 00 04 6f 62 6a 65 06
04 00 00 00 03 6e 65 74 06 05 00 00 00 07 63 74
76 69 6c 6c 65 17 00 00 00 0b
THAT LOOKS LIKE SOME KIND OF SECRET CODE. DO YOU THINK IT'S FROM THE CAPTAIN?

Your last chance to DO something... your object's finalizer

Sometimes you need to be sure something happens ***before*** your object gets garbage-collected, like **releasing unmanaged resources**.

In general, you'll never write a finalizer for an object that only owns managed resources. Everything you've encountered so far in this book has been managed — meaning managed by the CLR (including any object that ends up on the heap). But occasionally programmers need to access an underlying Windows resource that isn't part of the .NET Framework. If you find code on the Internet that uses the `[DllImport]` attribute, you might be using an unmanaged resource. And some of those non-.NET resources might leave your system unstable if they're not "cleaned up" somehow (maybe by calling a method). And that's what finalizers are for.

A special method in your object called the **finalizer** allows you to write code that will always execute when your object is destroyed. Think of it as your object's personal `finally` block: it gets executed last, no matter what.

Here's an example of a finalizer in the `Clone` class:

```
class Clone {
  string Location;
  int CloneID;

  public Clone (int cloneID, string location){
    this.CloneID = cloneID;
    this.Location = location;
  }

  public void TellLocation(string location, int cloneID){
     Console.WriteLine("My Identification number is {0} and " +
                       "you can find me here: {1}.", cloneID, location);
  }

  public void WreakHavoc(){...}

  ~Clone() {
    TellLocation(this.Location, this.CloneID);
    Console.WriteLine ("{0} has been destroyed", CloneID);
  }
}
```

Here's the constructor. It looks like the CloneID and Location fields are populated any time a Clone gets created.

This ~ (or "tilde") character says that the code in this block gets run when the object is garbage-collected.

This is the finalizer. It sends a message to the villain telling the ill-fated clone's location and ID. But it will only run when the object is garbage-collected.

You write a finalizer method just like a constructor, but instead of an access modifier, you put a ~ in front of the class name. That tells .NET that the code in the finalizer block should be run right before it garbage-collects the object.

Also, finalizers can't have parameters, because .NET doesn't need to tell it anything other than "you're done!"

WATCH IT!

Some of this code is for learning purposes only, not for your real programs.

Throughout the book we've made reference to how objects "eventually" get garbage-collected, but we never really specified exactly when that happens...just that it happens sometime after the reference to the object disappears. We're about to show you some code that automatically triggers garbage collection ***using*** `GC.Collect()` *and* ***pops up a*** `MessageBox` ***in a finalizer****. These things mess with the "guts" of the CLR. We're doing this to teach you about garbage collection.* ***Never do this outside of toy programs.***

When EXACTLY does a finalizer run?

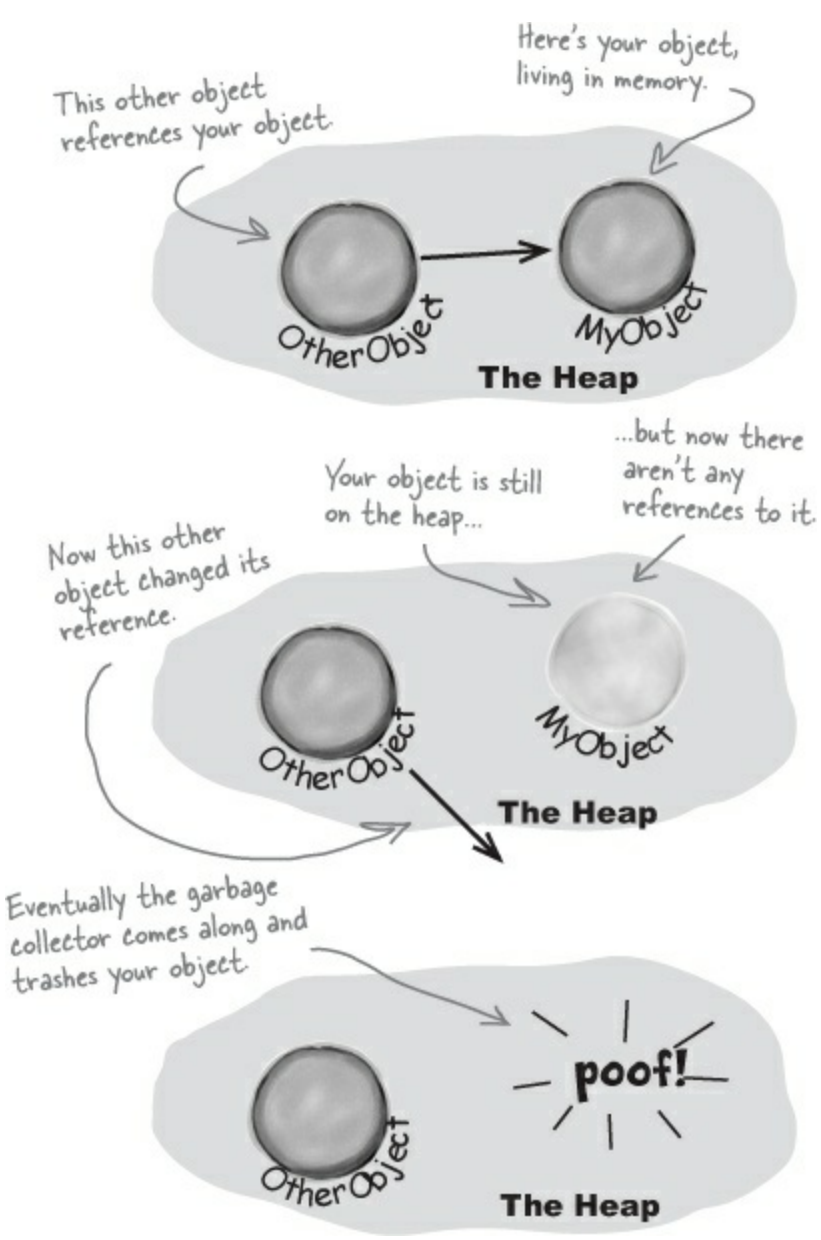
The finalizer for your object runs **after** all references are gone, but **before** that object gets garbage-collected. And garbage collection happens after ***all*** references to your object go away. But garbage collection doesn't always happen *right after* the references are gone.

Suppose you have an object with a reference to it. .NET sends the garbage collector to work, and it checks out your object. But since there are references to your object, the garbage collector ignores it and moves along. Your object keeps living on in memory.

Then, something happens. That last object holding a reference to *your* object decides to move on. Now, your object is sitting in memory, with no references. It can't be accessed. It's basically a **dead object**.

But here's the thing. ***Garbage collection is something that .NET controls,*** not your objects. So if the garbage collector isn't sent out again for, say, a few seconds, or maybe even a few minutes, your object still lives on in memory. It's unusable, but it hasn't been garbage-collected. **And any finalizer your object has does not (yet) get run.**

Finally, .NET sends the garbage collector out again. Your finalizer runs...possibly several minutes after the last reference to the object was removed or changed. Now that it's been finalized, your object is dead, and the collector tosses it away.


Here's your object, living in memory.
This other object references your object.
OtherObject
MyObject
The Heap
...but now there aren't any references to it.
Your object is still on the heap...
Now this other object changed its reference.
OtherObject
MyObject
The Heap
Eventually the garbage collector comes along and trashes your object.
poof!
OtherObject
The Heap

You can SUGGEST to .NET that it's time to collect the garbage

.NET does let you ***suggest*** that garbage collection would be a good idea. **Most times, you'll never use this method, because garbage collection is tuned to respond to a lot of conditions in the CLR and calling it *isn't really a good idea***. But just to see how a finalizer works, you could call for garbage collection on your own. If that's what you want to do, just call `GC.Collect()`.

Be careful, though. That method doesn't ***force*** .NET to garbage-collect things immediately. It just says, "Do garbage collection as soon as possible."

```
public void RemoveTheClones(
                  List<Clone> clones) {
  foreach (Clone clone in clonesToRemove)
    Clones.Remove(clone);
  GC.Collect();
}
```

We can't emphasize enough just how bad an idea it is to use GC.Collect() in a program that's not just a toy, because it can really confuse the CLR's garbage collector. It's an excellent tool for **learning** about garbage collection and finalizers, so we'll build a toy to play with it.

Dispose() works with using; finalizers work with garbage collection

NOTE

Like you saw earlier, Dispose() works without a using statement. When you build a Dispose() method, it shouldn't have any side effects that cause problems if it's run many times.

When an object implements `IDisposable`, its `Dispose()` is called at the end of the block after a `using` statement. If you don't use a `using` statement, then just setting the reference to `null` won't cause `Dispose()` to be called — you'll need to call it directly. An object's finalizer runs at garbage collection for that particular object. Let's explore how these two patterns differ. Start up **Visual Studio for Windows Desktop** and create a **Windows Forms Application project**.

DO THIS!

- **❶ Create a `Clone` class that implements `IDisposable` and has a finalizer.**
  The class should have one `int` automatic property called `Id`. It has a constructor, a `Dispose()` method, and a finalizer:

```
using System.Windows.Forms;

class Clone : IDisposable {
    public int Id { get; private set; }

    public Clone(int Id) {
        this.Id = Id;
    }

    public void Dispose() {
        MessageBox.Show("I've been disposed!",
                        "Clone #" + Id + " says...");
    }

    ~Clone() {
        MessageBox.Show("Aaargh! You got me!",
                        "Clone #" + Id + " says...");
    }
}
```

Since the class implements IDisposable, it has to have a Dispose() method.

Here's the finalizer. It will run when the object gets garbage-collected.

Popping up a MessageBox in a finalizer can mess with the "guts" of the CLR. Don't do it outside of a toy program for learning about garbage collection.

Here's a good example of how desktop apps can make good tools for exploring C# and .NET. In this project, you'll go back to creating a Windows Forms project to take advantage of the way `MessageBox` windows block as a tool to explore how garbage collection works.

Quick reminder: if you're using Visual Studio Professional, Premium, or Ultimate, you can create both Windows Store and Windows Desktop apps.

- **❷ Create a `Form` with three buttons.**
  Create one instance of `Clone` inside the `Click` handler for the first button with a `using` statement. Here's the first part of the code for the button:

Here's the form you should create.

The method creates a new Clone and then immediately kills it by taking away its reference.

```
private void clone1_Click(object sender, EventArgs e) {
    using (Clone clone1 = new Clone(1)) {
        // Do nothing!
    }
}
```

Since we declared clone1 with a using statement, its Dispose() method gets run.

As soon as the using block is done and the Clone object's Dispose() method is called, there's no more reference to it and it gets marked for garbage collection.

NOTE

You can make IDs for many objects — it'll add 2#, 3#, etc.

Add a watch for one of the `Clone` references, right-click on it, and choose Make Object ID to add the name `1#` to your Watch window. This lets you keep watching the object even after the reference goes out of scope. The Watch window will let you know when the object is collected (you may need to Click 🔄 to refresh it).

- **❸ Add the other two buttons.**
  Create another instance of `Clone` in the second button's `Click` handler, and set it to `null` manually:

```
private void clone2_Click(object sender, EventArgs e) {
    Clone clone2 = new Clone(2);
    clone2 = null;
}
```

Since this doesn't use a using statement, Dispose() won't ever get run, but the finalizer will.

For the third button, add a call to `GC.Collect()` to suggest that garbage collection occur.

```
private void gc_Click(object sender, EventArgs e) {
    GC.Collect();
}
```

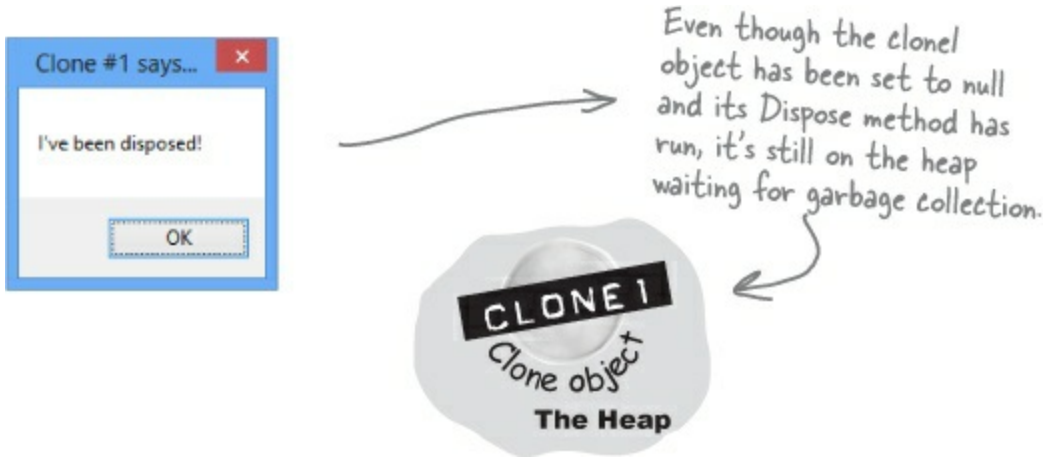
This suggests that garbage collection run.

Remember, normally it's not a great idea to do this. But it's fine here, because it's a good way to learn about garbage collection.

- **❹ Run the program and play with `Dispose()` and finalizers.**
  Click on the first button and check out the message box: `Dispose()` runs first.

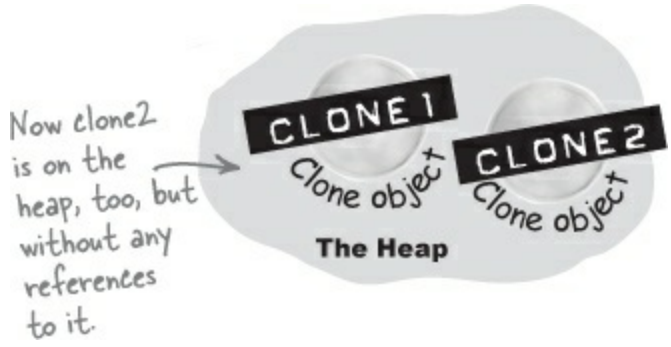
NOTE

Don't forget to add "using System.Windows.Forms;" to the top of your Clone class.



Garbage is collected...***eventually***. In most cases, you ***won't*** see the garbage collection message box, because your object is set to `null`, but garbage collection hasn't run yet.

Now click on the second button...nothing happens, right? That's because we didn't use a `using` statement, so there's no `Dispose()` method. And until the garbage collector runs, you won't see the message boxes from the finalizer.

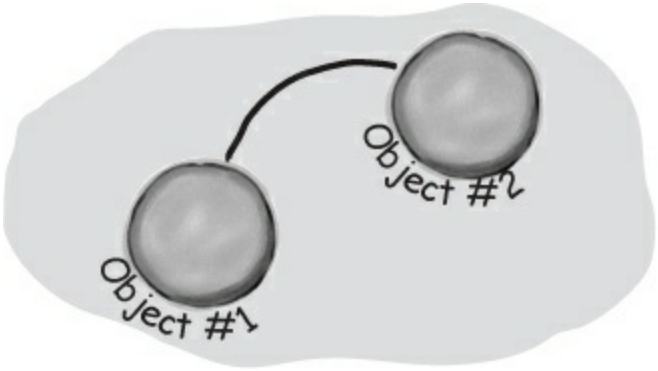


Now click the third button, to suggest garbage collection. You should see the finalizer from both `clone1` and `clone2` fire up and display message boxes.

Play around with the program. Click the Clone #1 button, then the Clone #2 button, then the GC button. Do it a few times. Sometimes Clone #1 is collected first, and sometimes Clone #2 is. And once in a while, the garbage collector runs even though you didn't ask it to using `GC.Collect()`.

Finalizers can't depend on stability

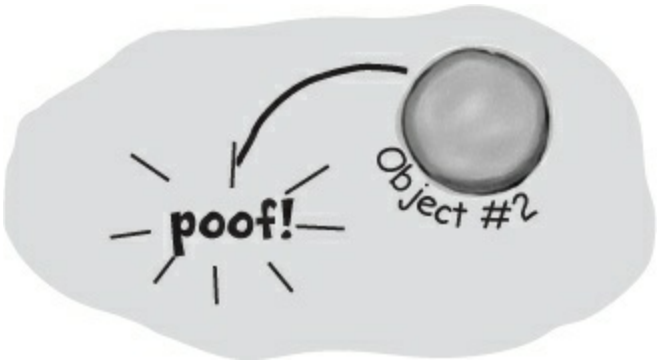
Let's say you've got two objects that have references to each other...



When you write a finalizer, you can't depend on it running at any one time. Even if you call `GC.Collect()` — which you should avoid, unless you have a really good reason to do it — you're only ***suggesting*** that the garbage collector is run. It's not a guarantee that it'll happen right away. And when it does, you have no way of knowing what order the objects will be collected in.

So what does that mean, in practical terms? Well, think about what happens if you've got two objects that have references to each other. If object #1 is collected first, then object #2's reference to it is pointing to an object that's no longer there. But if object #2 is collected first, then object #1's reference is invalid. So what that means is that ***you can't depend on references in your object's finalizer***. Which means that it's a really bad idea to try to do something inside a finalizer that depends on references being valid.

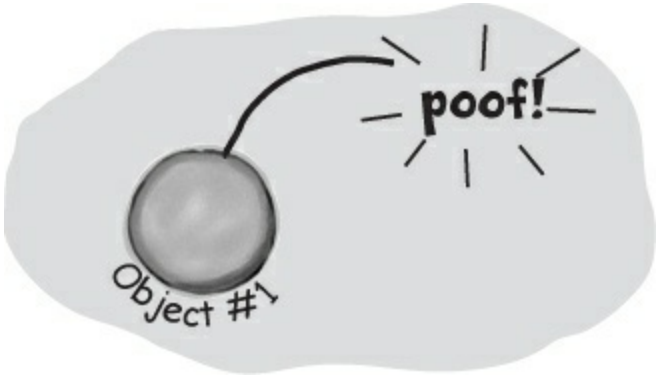
...if they're both marked for garbage collection at the same time, then object #1 could disappear first...



Serialization is a really good example of something that you **shouldn't do inside a finalizer**. If your object's got a bunch of references to other objects, serialization depends on ***all*** of those objects still being in memory... and all of the objects they reference, and the ones those objects reference, and so on. So if you try to serialize when garbage collection is happening, you could end up **missing** vital parts of your program because some objects might've been collected ***before*** the finalizer ran.

Luckily, C# gives us a really good solution to this: `IDisposable`. Anything that could modify your core data or that depends on other objects being in memory needs to happen as part of a `Dispose()` method, not a finalizer.

...on the other hand, object #2 could disappear before object #1. You've got no way of knowing the order...



Some people like to think of a finalizers as a kind of fail-safe for the `Dispose()` method. And that makes sense — you saw with your `Clone` object that just because you implement `IDisposable`, that doesn't mean the object's `Dispose()` method will get called. But you need to be careful — if your `Dispose()` method depends on other objects that are on the heap, then calling `Dispose()` from your finalizer can cause trouble. The best way around this is to make sure you **always use a `using` statement** any time you're creating an `IDisposable` object.

...and that's why one object's finalizer can't rely on any other object still being on the heap.

Make an object serialize itself in its Dispose()

Once you understand the difference between `Dispose()` and a finalizer, it's pretty easy to write objects that serialize themselves out automatically when they're disposed of.

DO THIS!

- **❶ MAKE THE CLONE CLASS (FROM Dispose() works with using; finalizers work with garbage collection) SERIALIZABLE.**
  Just add the `Serializeable` attribute on top of the class so that we can save the file out.

```
[Serializable]
class Clone : IDisposable
```

- **❷ MODIFY CLONE'S DISPOSE() METHOD TO SERIALIZE ITSELF OUT TO A FILE.**
  Let's use a `BinaryFormatter` to write `Clone` out to a file in `Dispose()`:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

// existing code

public void Dispose() {
  string filename = @"C:\Temp\Clone.dat";
  string dirname = @"C:\Temp\";
  if (File.Exists(filename) == false) {
    Directory.CreateDirectory(dirname);
  }
  BinaryFormatter bf = new BinaryFormatter();
  using (Stream output = File.OpenWrite(filename)) {
    bf.Serialize(output, this);
  }
  MessageBox.Show("Must...serialize...object!",
                  "Clone #" + Id + " says...");
}
```

We've gone back to using binary serialization and hardcoded directories as teaching tools because they're simple—and because we don't want you doing this in live code! This is only for toy programs.

You'll need a few more using directives to access the I/O classes we'll use.

The Clone will create the C:\Temp directory and serialize itself out to a file called Clone.dat.

We hardcoded the filename—we included them as string literals in the code. That's fine for a small toy program like this, but it's not problem-free. Can you think of problems this might cause, and how you could avoid them?

And is this Dispose() method really side-effect free? What happens if it's called more than once? These are all things you need to think about when you implement IDisposable.

- **❸ RUN THE APPLICATION.**
  You'll see the same behavior you saw on the last few pages...but before the `clone1` object is garbage-collected, it's serialized to a file. Look inside the file and you'll see the binary representation of the object.

BRAIN POWER

There's a lot to think about in this project! What do you think the rest of the `SuperHero` object's code looked like? We showed you part of it in Make an object serialize itself in its Dispose(). Could you write the rest now? More importantly, **should** you?

It's clearly ***possible*** to have your object serialize itself when it's disposed. But is that a good idea? Does it violate separation of concerns? Could it lead to code that's hard to maintain? What other problems could occur?

FIRESIDE CHATS

Tonight's talk: **the Dispose() method and a finalizer spar over who's more valuable.**

Dispose()	**Finalizer**
To be honest, I'm a little surprised I was invited here. I thought the programming world had come to a consensus. I mean, I'm way more valuable than you are. Really, you're pretty feeble. You can't even serialize yourself out, alter core data, anything. Pretty unstable, aren't you?	
	Excuse me? That's rich. I'm feeble...OK. Well, I didn't want to get into this, but since we're already stooping this low...at least I don't need an interface to get started. Without `IDisposable`, you're just another useless method.
There's an interface specifically **because** I'm so important. In fact, I'm the only method in it!	
	Right, right...keep telling yourself that. And what happens when someone forgets to use a `using` statement when they instantiate their object? Then you're nowhere to be found.
OK, you're right, programmers need to know they're going to need me and either call me directly or use a `using` statement to call me. But they always know when I'm gonna run, and they can use me to do whatever they need to do to clean up after their object. I'm powerful, reliable, and easy to use. I'm a triple threat. And you? Nobody knows exactly when you'll run or what the state of the application will be when you finally do decide to show up.	
	OK, but if you need to do something at the very last moment when an object is garbage-collected, there's no way to do it without me. I can free up network resources and Windows handles and streams and anything else that might cause a problem for the rest of the program if you don't clean it up. I can make sure that your objects deal with being trashed more gracefully, and that's nothing to sneeze at. **NOTE** Handles are what your programs use when they go around .NET and the CLR and interact directly with Windows. Since .NET doesn't know about them, it can't clean them up for you.
So there's basically nothing you can do that I can't do. You think you're a big shot because you always run with GC, but at least I can depend on other objects.	
	That's right, pal — I always run; you need someone else to run you. I don't need anyone or anything!

THERE ARE NO DUMB QUESTIONS

Q: **Q: Can a finalizer use all of an object's fields and methods?**

A: **A:** Sure. While you can't pass parameters to a finalizer method, you can use any of the fields in an object, either directly or using `this` — but be careful, because if those fields reference other objects, then the other objects may have already been garbage-collected. But you can definitely call other methods and properties in the object being finalized (as long as those methods and properties don't depend on other objects).

Q: **Q: What happens to exceptions that get thrown in a finalizer?**

A: **A:** Good question. It's totally legal to put a `try`/`catch` block inside a finalizer method. Give it a try yourself. Create a divide-by-zero exception inside a `try` block in the `Clone` program we just wrote. Catch it and throw up a message box that says "I just caught an exception." right before the "...I've been destroyed." box we'd already written. Now run the program and click on the first button and then the GC button. You'll see both the exception box and the destroyed box pop up. (Of course, it's generally a **really bad idea** to pop up message boxes in finalizers for objects that are more than just toys...and those message boxes may never actually pop up.)

Q: **Q: How often does the garbage collector run automatically?**

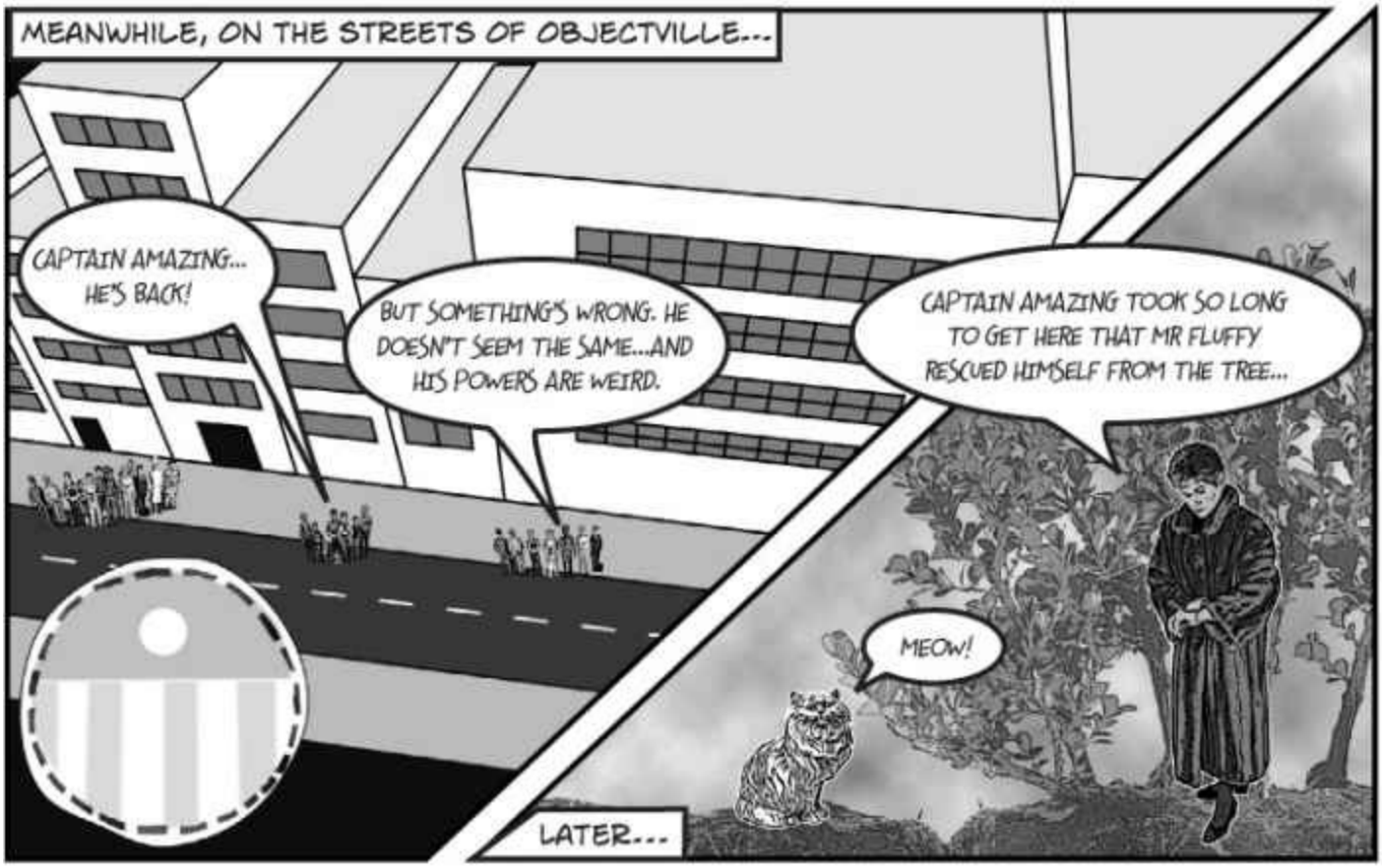
A: **A:** There's no good answer to that one. It doesn't run on an easily predictable cycle, and you don't have any firm control over it. You can be sure it will be run when your program exits. But if you want to be sure it'll run, you have to use `GC.Collect()` to set it off...and even then, you're only *suggesting* that the CLR should collect now.

Q: **Q: How soon after I call `GC.Collect()` will .NET start garbage collection?**

A: **A:** When you run `GC.Collect()`, you're telling .NET to garbage-collect as soon as possible. That's ***usually*** as soon as .NET finishes whatever it's doing. That means it'll happen pretty soon, but you can't actually control when.

Q: **Q: So if something absolutely must run, I put it in a finalizer?**

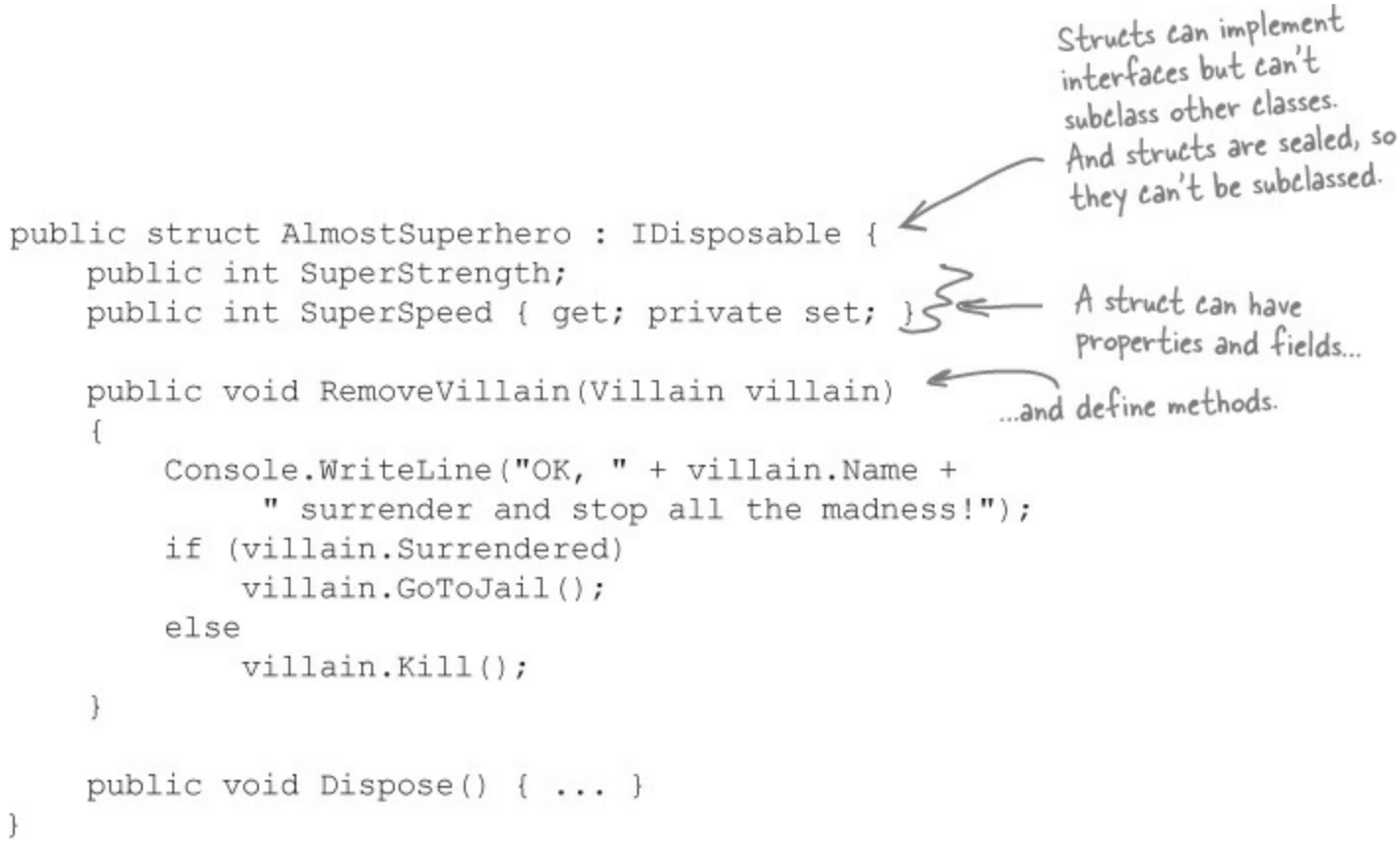
A: **A:** It's possible that your finalizer won't run. It's possible to suppress finalizers when garbage collection happens. Or the process could end entirely. If you aren't freeing unmanaged resources, you're almost always better off using `IDisposable` and `using` statements.


MEANWHILE, ON THE STREETS OF OBJECTVILLE...
CAPTAIN AMAZING... HE'S BACK!
BUT SOMETHING'S WRONG. HE DOESN'T SEEM THE SAME...AND HIS POWERS ARE WEIRD.
CAPTAIN AMAZING TOOK SO LONG TO GET HERE THAT MR FLUFFY RESCUED HIMSELF FROM THE TREE...
MEOW!
LATER...

EVEN LATER...
Captain Amazing's Hideout Collection
TOP SECRET
PUFF...PANT...UGH! I'M EXHAUSTED.
WHAT'S WRONG? WHY ARE THE CAPTAIN'S POWERS BEHAVING DIFFERENTLY? IS THIS THE END?

A struct looks like an object...

One of the types in .NET we haven't talked about much is the *struct*. Struct is short for **structure**, and structs look a lot like objects. They have fields and properties, just like objects. And you can even pass them into a method that takes an `object` type parameter:



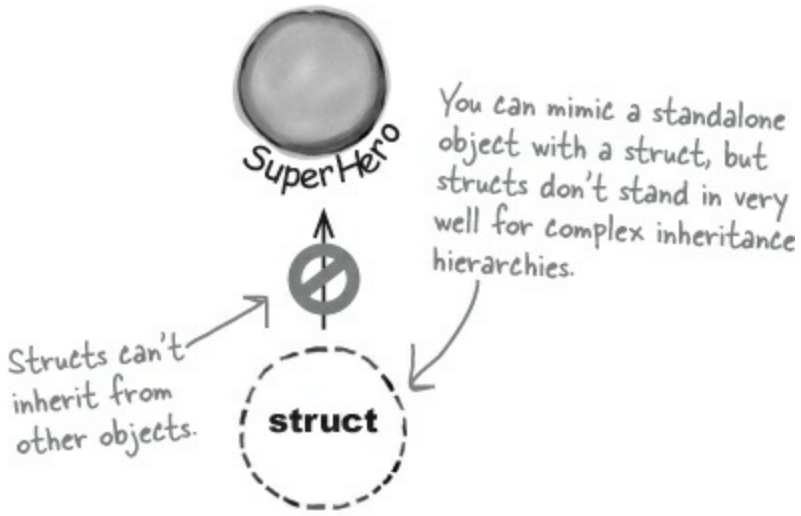

```
public struct AlmostSuperhero : IDisposable {
    public int SuperStrength;
    public int SuperSpeed { get; private set; }

    public void RemoveVillain(Villain villain)
    {
        Console.WriteLine("OK, " + villain.Name +
             " surrender and stop all the madness!");
        if (villain.Surrendered)
            villain.GoToJail();
        else
            villain.Kill();
    }

    public void Dispose() { ... }
}
```

The power of objects lies in their ability to mimic real-world behavior, through inheritance and polymorphism.

...but isn't an object

But structs ***aren't*** objects. They can have methods and fields, but they can't have finalizers. They also can't inherit from other classes or structs, or have classes or structs inherit from them.

> **NOTE**
>
> All structs inherit from System.ValueType, which in turn inherits from System. Object. That's why every struct has a ToString() method — it gets it from Object. But that's all the inheriting that structs are allowed to do.



Structs are best used for storing data, but the lack of inheritance and references can be a serious limitation.

> **NOTE**
>
> That's why you use classes a lot more than structs. But that doesn't mean they don't have their uses!

But the thing that sets structs apart from objects more than almost anything else is that you ***copy them by value, not by reference***. Flip the page to see what this means....

Values get copied; references get assigned

You already have a sense of how some types are different than others. On one hand you've got **value types** like `int`, `bool`, and `decimal`. On the other hand, you've got **objects** like `List`, `Stream`, and `Exception`. And they don't quite work exactly the same way, do they?

When you use the equals sign to set one value type variable to another, it **makes a copy of the value**, and afterward the two variables aren't connected to each other. On the other hand, when you use the equals sign with references, what you're doing is **pointing both references at the same object**.

> **NOTE**
>
> Here's a quick refresher on value types vs. objects.

- Variable declaration and assignment works the same with value types or object types:

> **NOTE**
>
> Remember when we said that methods and statements ALWAYS live in classes? Well, it turns out that's not 100% accurate — they can also live in structs.

int and bool are value types, List and Exception are object types.

```
int howMany = 25;
bool Scary = true;
List<double> temperatures = new List<double>();
Exception ex = new Exception("Does not compute");
```

These are all initialized in the same basic way.

- Differences creep in when you start to assign values, though. Value types all are handled with copying. Here's an example:

This line copies the value that's stored in the fifteenMore variable into the howMany variable and adds 15 to it.

Changing the fifteenMore variable has no effect on howMany, and vice versa.

```
int fifteenMore = howMany;
fifteenMore += 15;
Console.WriteLine("howMany has {0}, fifteenMore has {1}",
                  howMany, fifteenMore);
```

```
howMany has 25, fifteenMore has 40
```

The output here shows that `fifteenMore` and `howMany` are ***not*** connected:

- With object assignments, though, you're assigning references, not actual values:

This line sets the differentList reference to point to the same object as the temperatures reference.

```
temperatures.Add(56.5D);
temperatures.Add(27.4D);
List<double> differentList = temperatures;
differentList.Add(62.9D);
```

Both references point at the same actual object.

So changing the `List` means both references see the update...since they both point to a single `List` object.

```
Console.WriteLine("temperatures has {0}, differentlist has {1}",
                  temperatures.Count(), differentList.Count());
```

The output here demonstrates that `differentList` and `temperatures` are actually pointing to the ***same*** object:

```
temperatures has 3, differentList has 3
```

When you called differentList.Add(), it added a new temperature to the object that both differentList and temperatures point to.

Structs are value types; objects are reference types

When you create a struct, you're creating a **value type**. What that means is when you use equals to set one struct variable equal to another, you're creating a fresh *copy* of the struct in the new variable. So even though a struct *looks* like an object, it doesn't act like one.

SHARPEN YOUR PENCIL

Do this

- **❶ Create a struct called `Dog`.**
  Here's a simple struct to keep track of a dog. It looks just like an object, but it's not. Add it to a **new console application**.

```
public struct Dog {
  public string Name;
  public string Breed;

  public Dog(string name, string breed) {
    this.Name = name;
    this.Breed = breed;
  }

  public void Speak() {
      Console.WriteLine("My name is {0} and I'm a {1}.", Name, Breed);
  }
}
```

Yes, this is not good encapsulation. Bear with us—we're making a point.

- **❷ Create a class called `Canine`.**
  Make an exact copy of the `Dog` struct, except **replace `struct` with `class`** and then **replace `Dog` with `Canine`**. (Don't forget to rename `Dog`'s constructor.) Now you'll have a `Canine` class that you can play with, which is almost exactly equivalent to the `Dog` struct.
- **❸ Add a `Main()` method that makes some copies of `Dogs` and `Canines`.**
  Here's the code for the `Main()` method:

```
Canine spot = new Canine("Spot", "pug");
Canine bob = spot;
bob.Name = "Spike";
bob.Breed = "beagle";
spot.Speak();

Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
betty.Name = "Betty";
betty.Breed = "pit bull";
jake.Speak();

Console.ReadKey();
```

You've already used structs in your programs. Remember `DateTime` from previous chapters? You were working with a struct the whole time!

- **❹ Before you run the program...**
  Write down what you think will be written to the console when you run this code:

_____

_____

SHARPEN YOUR PENCIL SOLUTION

What did you think would get written to the console?

My name is Spike and I'm a beagle.

My name is Jake and I'm a poodle.

Here's what happened...

The `bob` and `spot` references both point to the same object, so both changed the same fields and accessed the same `Speak()` method. But structs don't work that way. When you created `betty`, you made a fresh copy of the data in `jake`. The two structs are completely independent of each other.

A new Canine object was created and the spot reference points to it.

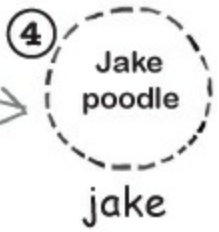
```
Canine spot = new Canine("Spot", "pug");  1
Canine bob = spot;  2
bob.Name = "Spike";
bob.Breed = "beagle";
spot.Speak();  3
```

The new reference variable bob was created, but no new object was added to the heap—the bob variable points to the same object as spot.

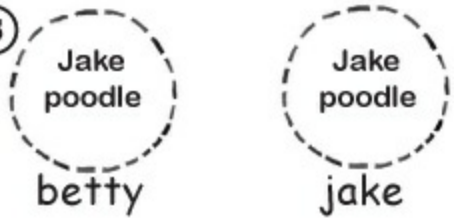
Since spot and bob both point to the same object, spot.Speak() and bob.Speak() both call the same method, and both of them produce the same output with "Spike" and "beagle".

```
Dog jake = new Dog("Jake", "poodle");  4
Dog betty = jake;  5
betty.Name = "Betty";
betty.Breed = "pit bull";
jake.Speak();  6
```

When you create a new struct, it looks really similar to creating an object—you've got a variable that you can use to access its fields and methods.



Here's the big difference. When you added the betty variable, you created a whole new value.



Since you created a fresh copy of the data, jake was unaffected when you changed betty's fields.

When you set one struct equal to another, you're creating a fresh COPY of the data inside the struct. That's because struct is a VALUE TYPE.

The stack vs. the heap: more on memory

Behind the Scenes

It’s not hard to understand how a struct differs from an object — you can make a fresh copy of a struct just using equals, which you can’t do with an object. But what’s really going on behind the scenes?

The .NET CLR divides your data into two places in memory. You already know that objects live on the **heap**. It also keeps another part of memory called the **stack** to store all of the local variables you declare in your methods, and the parameters that you pass into those methods. You can think of the stack as a bunch of slots that you can stick values in. When a method gets called, the CLR adds more slots to the top of the stack. When it returns, its slots are removed.

NOTE

Remember, when your program’s running, the CLR is actively managing memory, dealing with the heap, and collecting garbage.

Even though you can assign a struct to an object variable, structs and objects are different.

The Code

Here's code that you might see in a program.

```
Canine spot = new Canine("Spot", "pug");

Dog jake = new Dog("Jake", "poodle");
```

Here's what the stack looks like after these two lines of code run.

The Stack

This is where structs and local variables hang out.

```
Canine spot = new Canine("Spot", "pug");

Dog jake = new Dog("Jake", "poodle");

Dog betty = jake;
```

When you create a new struct—or any other value type variable—a new "slot" gets added onto the stack. That slot is a copy of the value in your type.

```
Canine spot = new Canine("Spot", "pug");

Dog jake = new Dog("Jake", "poodle");

Dog betty = jake;

SpeakThreeTimes(jake);

public SpeakThreeTimes(Dog dog) {
    int i;
    for (i = 0; i < 5; i++)
        dog.Speak();
}
```

When you call a method, the CLR puts its local variables on the top of the stack. It takes them off when it's done.

BEHIND THE SCENES

You definitely want to understand how a struct you copy by value is different from an object you copy by reference.

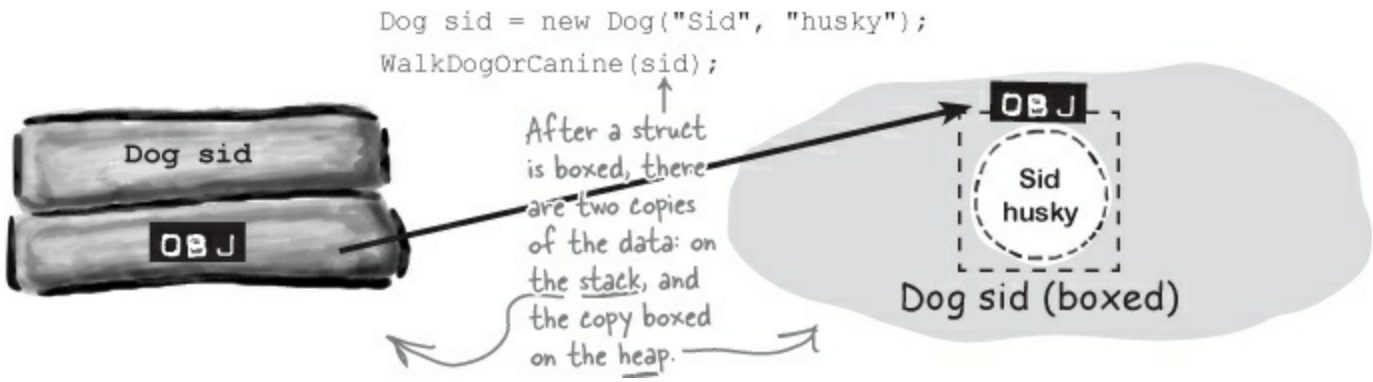
There are times when you need to be able to write a method that can take either a value type ***or*** a reference type — perhaps a method that can work with either a `Dog` struct or a `Canine` object. If you find yourself in that situation, you can use the `object` keyword:

```
public void WalkDogOrCanine(object getsWalked) { ... }
```

If you send this method a struct, the struct gets **boxed** into a special object "wrapper" that allows it to live on the heap. While the wrapper's on the heap, you can't do much with the struct. You have to "unwrap" the struct to work with it. Luckily, all of this happens *automatically* when you set an object equal to a value type, or pass a value type into a method that expects an object.

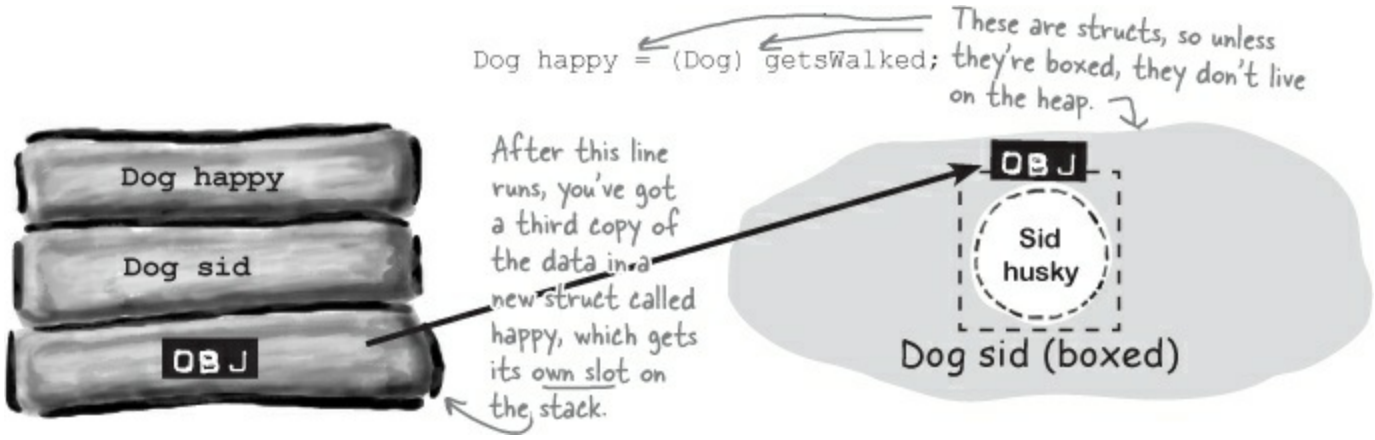
> **NOTE**
>
> You can also use the "is" keyword to see if an object is a struct, or any other value type, that's been boxed and put on the heap.

- ❶ Here's what the stack and heap look like after you create an object variable and set it equal to a `Dog` struct.



- ❷ If you want to unbox the object, all you need to do is cast it to the right type, and it gets unboxed automatically. You **can't use the `as` keyword with value types**, so you'll need to cast to `Dog`.

> **NOTE**
>
> The WalkDogOrCanine() method takes an object reference, so the Dog struct was boxed before it was passed in. Casting it back to a Dog unboxes it.



When a method is called, it looks for its arguments on the stack.

The stack plays an important part in how the CLR runs your programs. One thing we take for granted is the fact that you can write a method that calls another method, which in turn calls another method. In fact, a method can call itself (which is called *recursion*). The stack is what gives your programs the ability to do that.

Here are a couple of methods from a dog simulator program. They're pretty simple: `FeedDog()` calls `Eat()`, which calls `CheckBowl()`.

> **NOTE**
>
> Remember the terminology here: a parameter is what you call the part of the method declaration that specifies the values it needs; an argument is the actual value or reference that you pass into a method when you call it.

Here's what the stack looks like as the `FeedDog()` method calls `Eat()`, which calls `CheckBowl()`, which calls `Console.WriteLine()`:

```
public double FeedDog(Canine dogToFeed, Bowl dogBowl) {
    double eaten = Eat(dogToFeed.MealSize, dogBowl);
    return eaten + .05D; // A little is always spilled
}

public void Eat(double mealSize, Bowl dogBowl) {
    dogBowl.Capacity -= mealSize;
    CheckBowl(dogBowl.Capacity);
}

public void CheckBowl(double capacity) {
    if (capacity < 12.5D) {
        string message = "My bowl's almost empty!";
        Console.WriteLine(message);
    }
}
```

1. The `FeedDog()` method takes two parameters, a `Canine` reference and a `Bowl` reference. So when it's called, the two arguments passed to it are on the stack.

2. `FeedDog()` needs to pass two arguments to the `Eat()` method, so they're pushed onto the stack as well.

3. As the method calls pile up and the program goes deeper into methods that call methods that call other methods, the stack gets bigger and bigger.

4. When `Console.WriteLine()` exits, its arguments will be popped off of the stack. That way, `Eat()` can keep going as if nothing had happened. That's why the stack is so useful!

Use out parameters to make a method return more than one value

DO THIS!

Speaking of parameters and arguments, there are a few more ways that you can get values in and out of your programs, and they all involve adding **modifiers** to your method declarations. One of the most common ways of doing this is by using the **`out` modifier** to specify an output parameter. Here's how it works. Create a new Windows Forms application and add this empty method declaration to the form. Note the `out` modifiers on both parameters:

```
public int ReturnThreeValues(out double half, out int twice)
{
        return 1;
}
```

A method can return more than one value by using out parameters.

When you try to build your code, you'll see two errors: **the out parameter `half` must be assigned a value before control leaves the current method** (and you'll get an identical message for the `twice` parameter). Any time you use an `out` parameter, you ***always*** need to set it before the method returns — just like you always need to use a return statement if your method is declared with a return value. Here's the whole method:

```
Random random = new Random();
public int ReturnThreeValues(out double half, out int twice) {
    int value = random.Next(1000);
    half = ((double)value) / 2;
    twice = value * 2;
    return value;
}
```

This method needs to set all of its out parameters before it returns; otherwise, it won't compile.

Quick reminder: when Windows Forms programs call `Console.WriteLine()` it updates the IDE's Output window (View→Output).

Now that you've set the two `out` parameters, it compiles. So let's use them. Add a button with this event handler:

```
private void button1_Click(object sender, EventArgs e) {
    int a;
    double b;
    int c;
    a = ReturnThreeValues(b, c);
    Console.WriteLine("value = {0}, half = {1}, double = {2}", a, b, c);
}
```

Did you notice how you didn't need to initialize b and c? You don't need to initialize a variable before you use it as an argument to an out parameter.

Uh oh! There are more build errors: **Argument 1 must be passed with the `out` keyword**. Every time you call a method with an `out` parameter, you need to use the `out` keyword when you pass the argument to it. Here's what that line should look like:

```
a = ReturnThreeValues(out b, out c);
```

Now your program will build. When you run it, the `ReturnThreeValues()` methods sets the three values and returns all three of them: `a` gets the method's return value, `b` gets the value returned by the `half` parameter, and `c` gets the value returned by `twice`.

We're using a Windows Forms application for this project because it's easy for you to repeatedly click the buttons and see the console output in the Output window.

Pass by reference using the ref modifier

One thing you've seen over and over again is that every time you pass an `int`, `double`, `struct`, or any other value type into a method, you're passing a copy of that value to that method. There's a name for that: **pass by value**, which means that the entire value of the argument is copied.

But there's another way to pass arguments into methods, and it's called **pass by reference**. You can use the **`ref`** keyword to allow a method to work directly with the argument that's passed to it. Just like the `out` modifier, you need to use **`ref`** when you declare the method and also when you call it. It doesn't matter if it's a value type or a reference type, either — any variable that you pass to a method's `ref` parameter will be directly altered by that method.

> **NOTE**
>
> Under the hood, an out argument is just like a ref argument, except that it doesn't need to be assigned before going into the method, and must be assigned before the method returns.

You can see how it works — add this method to your program:

```
public void ModifyAnIntAndButton(ref int value, ref Button button) {
    int i = value;
    i *= 5;
    value = i - 3;
    button = button1;
}
```

When this method sets value and button parameters, what it's really doing is changing the values of the q and b variables in the button2_Click() method that called it.

And **add a button** with this event handler to call the method:

```
private void button2_Click(object sender, EventArgs e) {
    int q = 100;
    Button b = button1;
    ModifyAnIntAndButton(ref q, ref b);
    Console.WriteLine("q = {0}, b.Text = {1}", q, b.Text);
}
```

This prints "q = 497, b.Text = button1" because the method actually altered the q and b variables.

When `button2_Click()` calls the `ModifyAnIntAndButton()` method, it passes its `q` and `b` variables by reference. The `ModifyAnIntAndButton()` method works them just like any other variable. But since they were passed by reference, the method was actually updating the `q` and `b` variables all along, and not just a copy of them. So when the method exits, the `q` and `b` variables are updated with the modified value.

Run the program and debug through it, adding a watch for the `q` and `b` variables to see how this works.

> **BUILT-IN VALUE TYPES' TRYPARSE() METHOD USES OUT PARAMETERS**
>
> There's a great example of out parameters built right into some of the built-in value types. There are a lot of times that you'll want to convert a string like "35.67" into a double. And there's a method to do exactly that: double.Parse("35.67") will return the double value 35.67. But double.Parse("xyz") will throw a FormatException. Sometimes that's exactly what you want, but other times you want to check if a string can be parsed into a value. That's where the TryParse() method comes in: double.TryParse("xyz", out d) will return false and set d to 0, but double.TryParse("35.67", out d) will return true and set d to 35.67.
>
> Also, remember back in Chapter 9 when we used a switch statement to convert Spades into Suits.Spades? Well, there are static methods Enum.Parse() and Enum.TryParse() that do the same thing, except for enums!

Use optional parameters to set default values

A lot of times, your methods will be called with the same arguments over and over again, but the method still needs the parameter because occasionally it changes. It would be useful if you could set a default value, so you only needed to specify the argument when calling the method if it was different.

That's exactly what optional parameters do. You can specify an optional parameter in a method declaration by using an equals sign followed by the default value for that parameter. You can have as many optional parameters as you want, but all of the optional parameters have to come after the required parameters.

Here's an example of a method that uses optional parameters to check if someone has a fever:

```
void CheckTemperature(double temperature, double tooHigh = 99.5, double tooLow = 96.5)
{
    if (temperature < tooHigh && temperature > tooLow)
        Console.WriteLine("Feeling good!");
    else
        Console.WriteLine("Uh-oh -- better see a doctor!");
}
```

Optional parameters have default values specified in the declaration.

This method has two optional parameters: `tooHigh` has a default value of 99.5, and `tooLow` has a default value of 96.5. Calling `CheckTemperature()` with one argument uses default values for both `tooHigh` and `tooLow`. If you call it with two arguments, it will use the second argument for the value of `tooHigh`, but still use the default value for `tooLow`. You can specify all three arguments to pass values for all three parameters.

There's another option as well. If you want to use some (but not all) of the default values, you can use **named arguments** to pass values for just those parameters that you want to pass. All you need to do is give the name of each parameter followed by a colon and its values. If you use more than one named argument, make sure you separate them with commas, just like any other argument.

Add the `CheckTemperature()` method to your form, and then **add a button** with the following event handler. Debug through it to make sure you understand exactly how this works:

```
private void button3_Click(object sender, EventArgs e)
{
    // Those values are fine for your average person
    CheckTemperature(101.3);

    // A dog's temperature should be between 100.5 and 102.5 Fahrenheit
    CheckTemperature(101.3, 102.5, 100.5);

    // Bob's temperature is always a little low, so set tooLow to 95.5
    CheckTemperature(96.2, tooLow: 95.5);
}
```

Use optional parameters and named arguments when you want your methods to have default values.

Use nullable types when you need nonexistent values

In a lot of projects earlier in the book, you used `null` to indicate that there is no value. That's very typical: you can use `null` to indicate that a variable, field, or property is empty, and you can check to see if it's equal to `null`, which **means that it doesn't have a value**. But for structs (and ints, booleans, enums, and other value types), you can't set them to `null`. That means these statements:

```
bool myBool = null;
DateTime myDate = null;
```

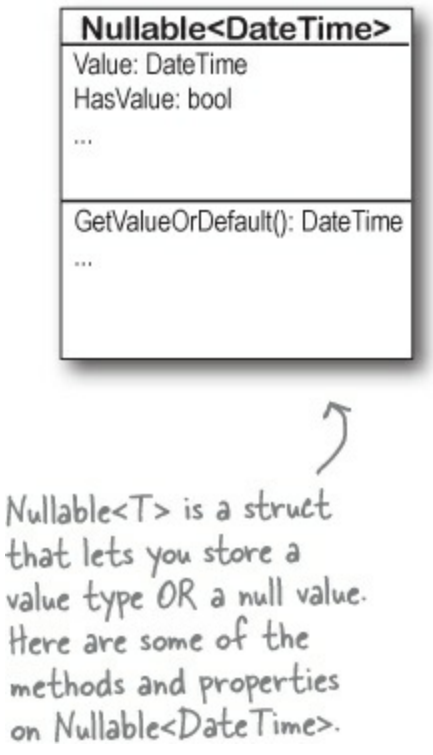
NOTE

Back in Chapter 11, you used DateTime.MinValue to mean "date not set" in the Excuse Manager app. Nullable<DateTime> would make both your code and the serialized XML files easier to read.

will cause errors when you try to compile! So how do you indicate an empty value for these types?

Let's say your program needs to work with a date and time value. Normally you'd use a `DateTime` variable. But what if that variable doesn't always have a value? That's where nullable types comes in really handy. All you need to do is add a question mark (?) to the end of any value type, and it becomes a **nullable type** that you can set to `null`.

```
bool? myNulableBool = null;
DateTime? myNullableDate = null;
```



Every nullable type has a property called `Value` that gets or sets the value. A `DateTime`? will have a `Value` of type `DateTime`, an `int`? will have one of type `int`, etc. They'll also have a property called `HasValue` that returns `true` if it's not `null`.

You can always convert a value type to a nullable type:

```
DateTime myDate = DateTime.Now;
DateTime? myNullableDate = myDate;
```

But you need to cast the nullable type in order to assign it back to a value type:

```
myDate = (DateTime) myNullableDate;
```

But you also get this handy `Value` property — it also returns the value:

```
myDate = myNullableDate.Value;
```

If `HasValue` is `false`, the `Value` property will throw an `InvalidOperationException`, and so will the cast (because that cast is equivalent to using the `Value` property).

THE QUESTION MARK T? IS AN ALIAS FOR NULLABLE<T>

When you add a question mark to any value type (like int? or decimal?), the compiler translates that to the Nullable<T> struct (Nullable<int> or Nullable<decimal>). You can see this for yourself: add a Nullable<DateTime> variable to a program, put a breakpoint on it, and add a watch for it in the debugger. You'll see System.DateTime? displayed in the Watch window in the IDE. This is an example of an alias, and it's not the first one you've encountered. Hover your cursor over any int. You'll see that it translates to a struct called System.Int32:

Take a minute and do that for each of the types at the beginning of Chapter 4. Notice how all of them are aliases for structs — except for string, which is a class called System.String (it's a reference type, not a value type).

Nullable types help you make your programs more robust

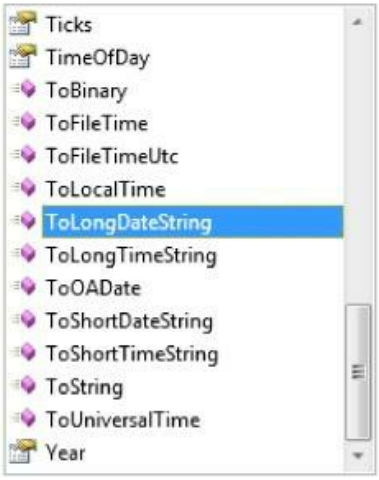
Do this!

Users do all sorts of crazy things. You think you know how people will use a program you're writing, but then someone clicks buttons in an unexpected order, or enters 256 spaces in a textbox, or uses the Windows Task Manager to quit your program halfway through writing data to a file, and suddenly it's popping up all manner of errors. Remember in Chapter 12 when we talked about how a program that can gracefully handle badly formatted, unexpected, or just plain bizarre input is called **robust**? When you're processing raw input from your users, nullable types can be very useful in making your programs more robust. Now see for yourself.**Create a new console application** and add this `RobustGuy` class to it:

When you add RobustGuy's ToString() method, take a look at the IntelliSense window when you enter Birthday.Value. Since the Value property is a DateTime, you'll see all the usual DateTime members.

Use the ToLongDateString() method to convert it to a human-readable string.

```
class RobustGuy {
    public DateTime? Birthday { get; private set; }
    public int? Height { get; private set; }

    public RobustGuy(string birthday, string height) {
        DateTime tempDate;
        if (DateTime.TryParse(birthday, out tempDate))
            Birthday = tempDate;
        else
            Birthday = null;

        int tempInt;
        if (int.TryParse(height, out tempInt))
            Height = tempInt;
        else
            Height = null;
    }

    public override string ToString() {
        string description;
        if (Birthday.HasValue)
            description = "I was born on " + Birthday.Value.ToLongDateString();
        else
            description = "I don't know my birthday";
        if (Height.HasValue)
            description += ", and I'm " + Height + " inches tall";
        else
            description += ", and I don't know my height";
        return description;
    }
}
```



Use the DateTime and int TryParse() methods to attempt to convert the user input into values.

If the user entered garbage, the Nullable types won't have values, so their HasValue() methods will return false.

Try experimenting with the other DateTime methods that start with "To" to see how they affect your program's output.

And here's the `Main()` method for the program. It uses **`Console.ReadLine()`** to get input from the user:

```
static void Main(string[] args) {
    Console.Write("Enter birthday: ");
    string birthday = Console.ReadLine();
    Console.Write("Enter height in inches: ");
    string height = Console.ReadLine();
    RobustGuy guy = new RobustGuy(birthday, height);
    Console.WriteLine(guy.ToString());
    Console.ReadKey();
}
```

When you run the program, see what happens when you enter different values for dates. `DateTime.TryParse()` can figure out a lot of them. When you enter a date it can't parse, the `RobustGuy`'s `Birthday` property will have no value.

NOTE

Console.ReadLine() lets the user enter text into the console window. When the user hits enter, it returns the input as a string.

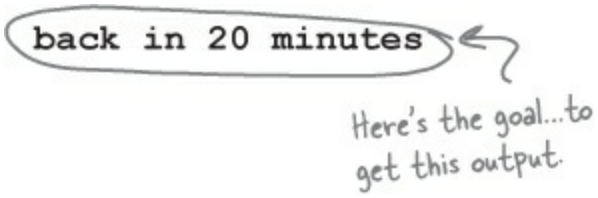
POOL PUZZLE

Your **job** is to take snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make the code write this output to the console when a **new instance**

of the `Faucet` class is created:

```
public class Faucet {
    public Faucet() {
        Table wine = new Table();
        Hinge book = new Hinge();
        wine.Set(book);
        book.Set(wine);
        wine.Lamp(10);
        book.garden.Lamp("back in");
        book.bulb *= 2;
        wine.Lamp("minutes");
        wine.Lamp(book);
    }
}
```

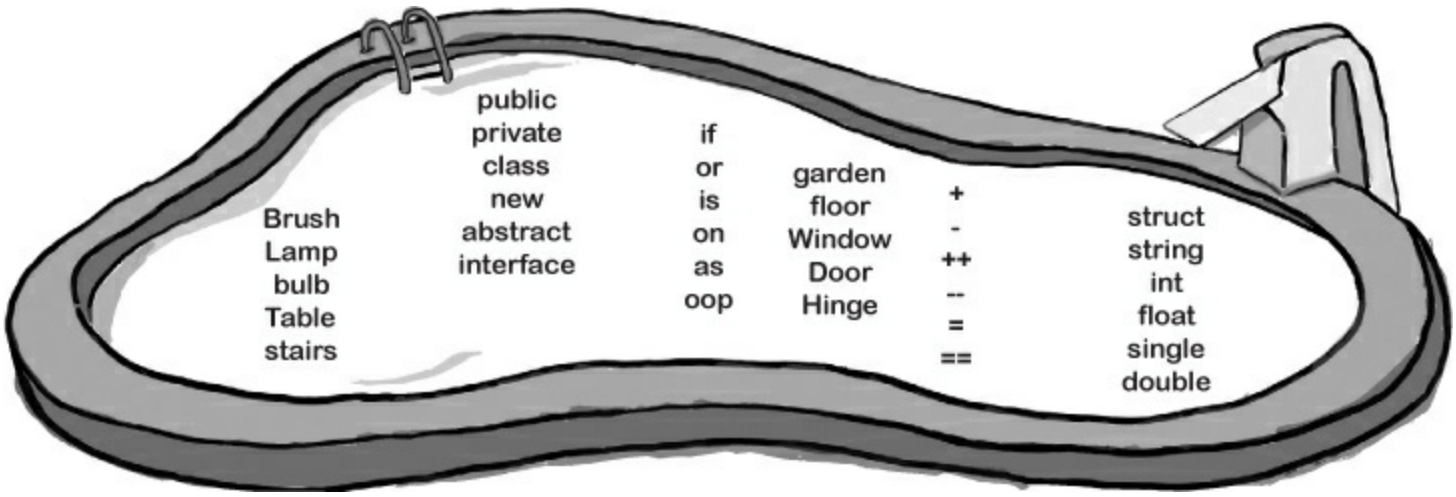
Output when you create a new `Faucet` object:




```
public _______ Table {
    public string stairs;
    public Hinge floor;
    public void Set(Hinge b) {
        floor = b;
    }
    public void Lamp(object oil) {
        if (oil ____ int)
            _______.bulb = (int)oil;
        else if (oil ____ string)
            stairs = (string)oil;
        else if (oil ____ Hinge) {
            _______ vine = oil ____ _______;
            Console.WriteLine(vine.Table()
            + " " + ______.bulb + " " + stairs);
        }
    }
}

public _______ Hinge {
    public int bulb;
    public Table garden;
    public void Set(Table a) {
        garden = a;
    }
    public string Table() {
        return _______.stairs;
    }
}
```

Bonus points: Circle the lines where boxing happens.

Note: Each thing from the pool can be used more than once.



Answers in Pool Puzzle Solution.

THERE ARE NO DUMB QUESTIONS

Q: **Q: OK, back up a minute. Why do I care about the stack?**

A: **A:** Because understanding the difference between the stack and the heap helps you keep your reference types and value types straight. It's easy to forget that structs and objects work very differently — when you use the equals sign with both of them, they look really similar. Having some idea of how .NET and the CLR handle things under the hood helps you understand ***why*** reference and value types are different.

Q: **Q: And boxing? Why is that important to me?**

A: **A:** Because you need to know when things end up on the stack, and you need to know when data's being copied back and forth. Boxing takes extra memory and more time. When you're only doing it a few times (or a few hundred times) in your program, then you won't notice the difference. But let's say you're writing a program that does the same thing over and over again, millions of times a second. That's not too far-fetched: you'll build an arcade game at the end of the book that could do many calculations per second. If you find that your program's taking up more and more memory, or going slower and slower, then it's possible that you can make it more efficient by avoiding boxing in the part of the program that repeats.

Q: **Q: I get how you get a fresh copy of a struct when you set one struct variable equal to another one. But why is that useful to me?**

A: **A:** One place that's really helpful is with **encapsulation**. Take a look at this familiar code from a class that knows its location:

```
protected Point location;
public Point Location {
  get { return location; }
}
```

If `Point` were a class, then this would be terrible encapsulation. It wouldn't matter that `location` is private, because you made a public read-only property that returns a reference to it, so any other object would be able to access it.
Lucky for us, `Point` is actually a struct. And that means that the public `Location` property returns a fresh copy of the point. The object that uses it can do whatever it wants to that copy — none of those changes will make it to the private `location` field.

NOTE

Go back to the label bouncer project from Chapter 4. Under the hood, you were indirectly using points and locations, which means your code was setting struct values (even if you didn't declare them directly).

Q: **Q: How do I know whether to use a struct or a class?**

A: **A:** Most of the time, programmers use classes. Structs have a lot of limitations that can really make it hard to work with them for large jobs. They don't support inheritance or abstraction, and only limited polymorphism, and you already know how important those things are for writing code.
Where structs come in really handy is if you have a small, limited type of data that you need to work with repeatedly. Rectangles and points are good examples — there's not much you'll do with them, but you'll use them over and over again. Structs tend to be relatively small and limited in scope. If you find that you have a small chunk of a few different kinds of data that you want to store in a field in a class or pass to a method as a parameter, that's probably a good candidate for a struct. But if the way you use the struct will cause it to be boxed most of the time, so you may be better off with a class.

A struct can be very valuable when you want to add good encapsulation to your class, because a read-only property that returns a struct always makes a fresh copy of it.

SHARPEN YOUR PENCIL

This method is supposed to kill a `Clone` object, but it doesn't work. Why not?

```
private void SetCloneToNull(Clone clone) {
    clone = null;
}
```

_____

_____

NOTE

Pop quiz, hotshot! Answer's in Extension methods add new behavior to EXISTING classes.

“Captain” Amazing...not so much

With all this talk of boxing, you should have a pretty good idea of what was going on with the less-powerful, more-tired Captain Amazing. In fact, it wasn’t Captain Amazing at all, but a boxed struct:

- ❶ **Structs can’t inherit from classes.**
  No wonder the Captain’s superpowers seemed a little weak! He didn’t get any inherited behavior.
- ❷ **Structs are copied by value.**
  This is one of the most useful things about them. It’s especially useful for encapsulation.

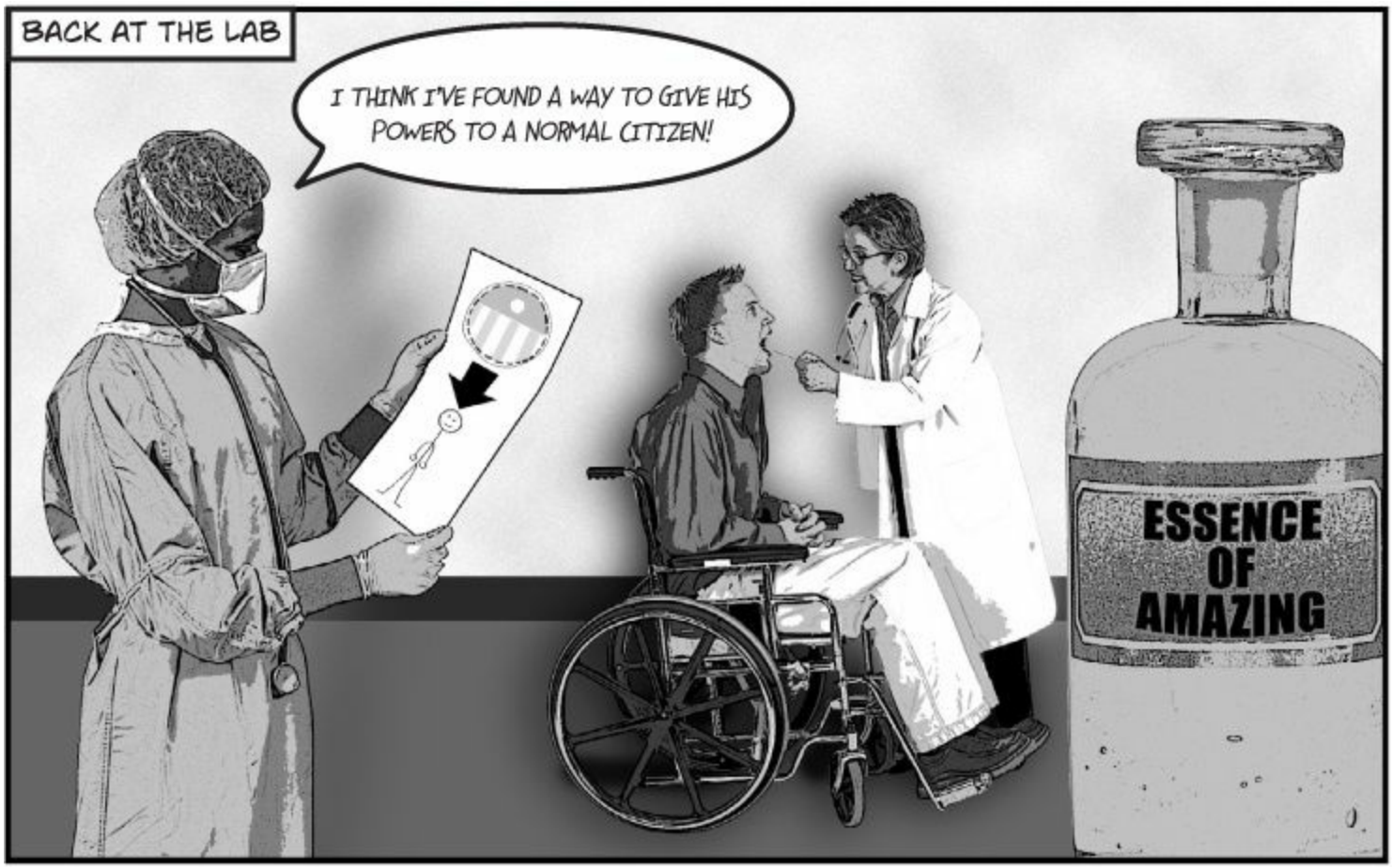
- ❶ **You can’t create a fresh copy of an object.**

> **NOTE**
>
> That’s one big advantage of structs (and other value types) — you can easily make copies of them.

When you set one object variable equal to another, you’re copying a ***reference*** to the ***same*** variable.

- ❷ **You can use the `as` keyword with an object.**
  Objects allow for polymorphism by allowing an object to function as any of the objects it inherits from.

> **NOTE**
>
> One important point: you can use the “is” keyword to check if a struct implements an interface, which is one aspect of polymorphism that structs do support.



Extension methods add new behavior to EXISTING classes

> **NOTE**
>
> Remember the sealed modifier from Chapter 7? It's how you set up a class that can't be extended.

Sometimes you need to extend a class that you can't inherit from, like a `sealed` class (a lot of the .NET classes are sealed, so you can't inherit from them). And C# gives you a flexible tool for that: **extension methods**. When you add a class with extension methods to your project, it **adds new methods that appear on classes** that already exist. All you have to do is create a static class, and add a **`static`** method that accepts an instance of the class as its first parameter using the `this` keyword.

So let's say you've got a `sealed OrdinaryHuman` class (remember, that means you can't extend it):

```
sealed class OrdinaryHuman {
    private int age;
    int weight;

    public OrdinaryHuman(int weight){
        this.weight = weight;
    }

    public void GoToWork() { /* code to go to work */ }
    public void PayBills() { /* code to pay bills */ }
}
```

The OrdinaryHuman class is sealed, so it can't be subclassed. But what if we want to add a method to it?

The `SuperSoldierSerum` method adds an extension method to `OrdinaryHuman`:

```
static class SuperSoldierSerum {
    public static string BreakWalls(this OrdinaryHuman h, double wallDensity) {
        return ("I broke through a wall of " + wallDensity + " density.");
    }
}
```

You use an extension method by specifying the first parameter using the "this" keyword.

Since we want to extend the OrdinaryHuman class, we make the first parameter this OrdinaryHuman.

Extension methods are always static methods, and they have to live in static classes.

As soon as the `SuperSoldierSerum` class is added to the project, `OrdinaryHuman` gets a `BreakWalls` method. So now a form can use it:

```
static void Main(string[] args){
    OrdinaryHuman steve = new OrdinaryHuman(185);
    Console.WriteLine(steve.BreakWalls(89.2));
}
```

When the program creates an instance of the OrdinaryHuman class, it can access the BreakWalls() method directly—as long as it has access to the SuperSoldierSerum class.

Go ahead, try it out! **Create a new console application** and add the two classes and the Main() method to it. Debug into the BreakWalls() method and see what's going on.

SHARPEN YOUR PENCIL SOLUTION

This method is supposed to kill a `Clone` object, but it doesn't work. Why not?

```
private void SetCloneToNull(Clone clone) {
    clone = null;
}
```

All this method does is set its own parameter to null, but that parameter's just a reference to a Clone. It's like sticking a label on an object and peeling it off again.

NOTE

So the clone parameter is just on the stack, so setting it to null doesn't do anything to the heap.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Tell me again why I wouldn't add the new methods I need directly to my class code, instead of using extensions?**

A: **A:** You could do that, and you probably should if you're just talking about adding a method to one class. Extension methods should be used pretty sparingly, and only in cases where you absolutely can't change the class you're working with for some reason (like it's part of the .NET Framework or another third party). Where extension methods really become powerful is when you need to extend the behavior of something you ***wouldn't normally have access to***, like a type or an object that comes for free with the .NET Framework or another library.

Q: **Q: Why use extension methods at all? Why not just extend the class with inheritance?**

A: **A:** If you can extend the class, then you'll usually end up doing that — extension methods aren't meant to be a replacement for inheritance. But they come in really handy when you've got classes that you can't extend. With extension methods, you can change the behavior of whole groups of objects, and even add functionality to some of the most basic classes in the .NET Framework.
Extending a class gives you new behavior, but requires that you use the new subclass if you want to use that new behavior.

Q: **Q: Does my extension method affect all instances of a class, or just a certain instance of the class?**

A: **A:** It will affect all instances of a class that you extend. In fact, once you've created an extension method, the new method will show up in your IDE alongside the extended class's normal methods.

NOTE

One more point to remember about extension methods: you don't gain access to any of the class's internals by doing an extension method, so it's still acting as an outsider!

Exactly! There are some classes that you can't inherit from.

Pop open any project, add a class, and try typing this:

```
class x : string { }
```

Try to compile your code — the IDE will give you an error. The reason is that some .NET classes are **sealed**, which means that you can't inherit from them. (You can do this with your own classes, too! Just add the `sealed` keyword to your class after the `public` access modifier, and no other class will be allowed to inherit from it.) Extension methods give you a way to extend it, even if you can't inherit from it.

But that's not all you can do with extension methods. In addition to extending classes, you can also extend **interfaces**. All you have to do is use an interface name in place of the class, after the `this` keyword in the extension method's first parameter. When you do, the extension method is added to **every class that implements that interface**. You'll learn all about LINQ in the next chapter — while you're learning, one thing to keep in mind is that it was built *entirely with extension methods*, extending the `IEnumerable<T>` interface.

NOTE

The combination of an interface plus extension methods can be very useful, because it lets you add behavior to any class that implements the interface.

Extending a fundamental type: string

You don’t often get to change the behavior of a language’s most fundamental types, like strings. But with extension methods, you can do just that! Create a new project, and add a file called *HumanExtensions.cs*. It doesn’t matter what kind of project you create — you’ll be using the IDE to explore how extension methods work.

DO THIS!

- **❶ PUT ALL OF YOUR EXTENSION METHODS IN A SEPARATE NAMESPACE.**
  It's a good idea to keep all of your extensions in a different namespace than the rest of your code. That way, you won't have trouble finding them for use in other programs. Set up a static class for your method to live in, too.

```
namespace MyExtensions {
  public static class HumanExtensions {
```

Using a separate namespace is a good organizational tool.

The class your extension method is defined in must be static.

- **❷ CREATE THE STATIC EXTENSION METHOD, AND DEFINE ITS FIRST PARAMETER AS THIS AND THEN THE TYPE YOU'RE EXTENDING.**
  The two main things you need to know when you declare an extension method are that the method needs to be static and it takes the class it's extending as its first parameter.

The extension method must be static, too.

```
public static bool IsDistressCall (this string s){
```

"this string" says we're extending the string class.

- **❸ PUT THE CODE TO EVALUATE THE STRING IN THE METHOD.**

```
public static class HumanExtensions {
  public static bool IsDistressCall(this string s){
    if (s.Contains("Help!"))
      return true;
    else
      return false;
  }
}
```

You want this class to be accessed by code in the other namespace, so **make sure you make it public!**

This checks the string for a certain value...something definitely not in the default string class.

- **❹ USE YOUR NEW IsDistressCall() EXTENSION METHOD.**
  Go to any other class and add `using MyExtensions;` to the top. Now, when you use a string, you get the extension methods for free. You can see this for yourself by typing the name of a string variable and a period:

```
static void Main(string[] args)
{
    string message = "Clones are wreaking havoc at the factory. Help!";
    message.
}
```

As soon as you type the dot, the IDE pops up a helper window with all of string's methods... including your extension method.

Comment out the using line, and the extension method will disappear from the IntelliSense window.

(extension) bool string.IsDistressCall()

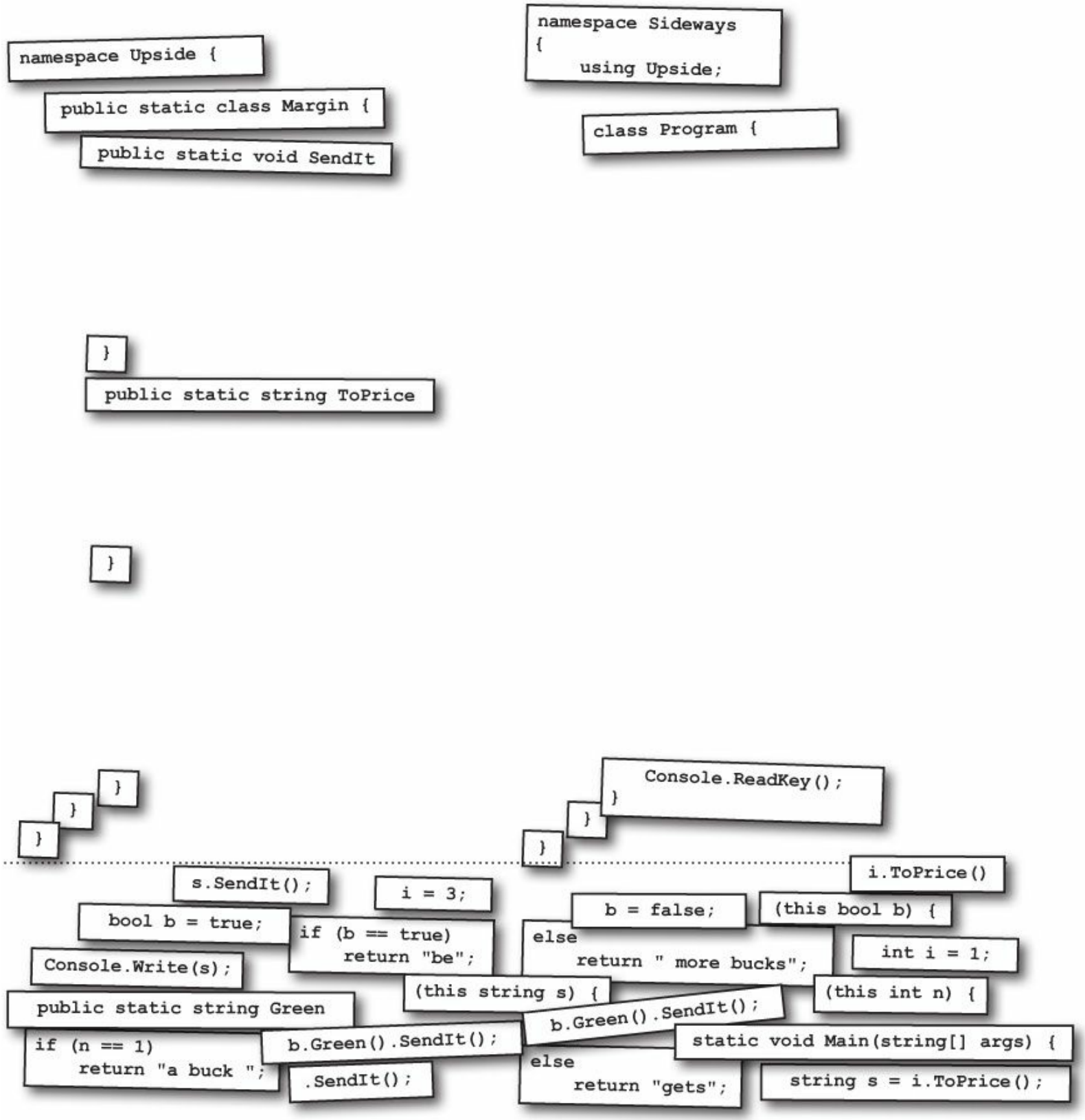
The IntelliSense window tells you that it's an extension.

This toy example just shows you the syntax of extension methods. To get a real sense of how useful they are, just wait until the next chapter. It's all about LINQ, which is implemented entirely with extension methods.

EXTENSION MAGNETS

Arrange the magnets to produce this output:

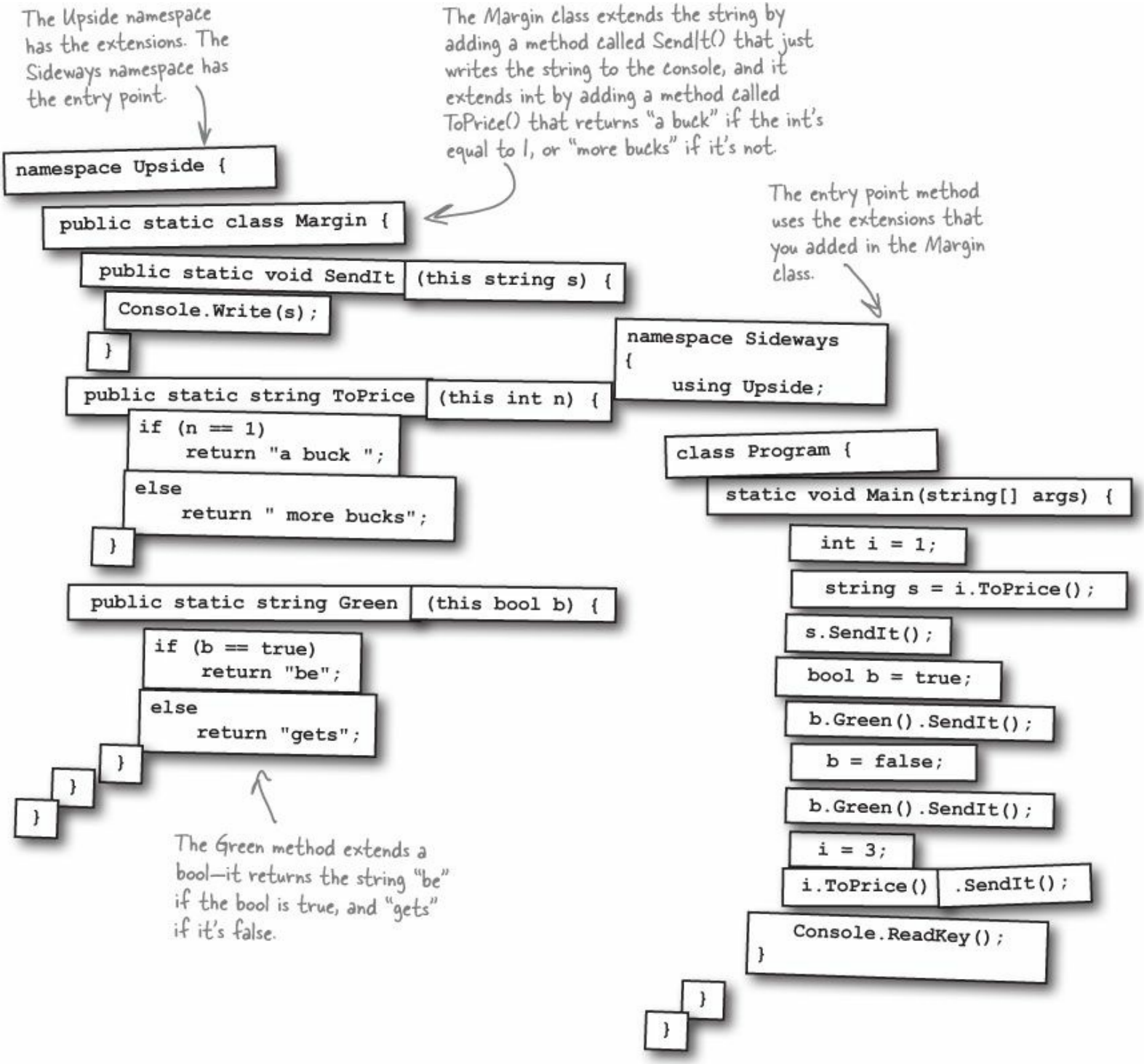
```
a buck begets more bucks
```



EXTENSION MAGNETS

Your job was to arrange the magnets to produce this output:

```
a buck begets more bucks
```



The **UNIVERSE**

CAPTAIN AMAZING REBORN

Death was not the end!

By Bucky Barnes
UNIVERSE STAFF WRITER

OBJECTVILLE

Captain Amazing deserializes himself, makes stunning comeback

In a stunning turn of events, Captain Amazing has returned to Objectville. Last month, Captain Amazing's coffin was found empty, and only a strange note left where his body should have been. Analysis of the note revealed Captain Amazing's object DNA—all his last fields and values, captured faithfully in binary format.

Today, that data has sprung to life. The Captain is back, deserialized from his own brilliant note. When asked how he conceived of such a plan, the Captain merely shrugged and mumbled, "Chapter 10." Sources close to the Captain refused to comment on the meaning of his cryptic reply, but did admit that prior to his failed assault on Swindler, the Captain had spent a lot of time reading books, studying `Dispose` methods and persistence. We expect Captain Amazing…

Captain Amazing is back!

…see AMAZING on A-5

POOL PUZZLE SOLUTION

Output when you create a new `Faucet` object:

```
back in 20 minutes
```

```
public class Faucet {
    public Faucet() {
        Table wine = new Table();
        Hinge book = new Hinge();
        wine.Set(book);
        book.Set(wine);
        wine.Lamp(10);
        book.garden.Lamp("back in");
        book.bulb *= 2;
        wine.Lamp("minutes");
        wine.Lamp(book);
    }
}
```

Here's why Table has to be a struct. If it were a class, then wine would point to the same object as book.Garden, which would cause this to overwrite the "back in" string.

```
public struct Table {
    public string stairs;
    public Hinge floor;
    public void Set(Hinge b) {
        floor = b;
    }
    public void Lamp(object oil) {
        if (oil is int)
            floor.bulb = (int)oil;
        else if (oil is string)
            stairs = (string)oil;
        else if (oil is Hinge) {
            Hinge vine = oil as Hinge;
            Console.WriteLine(vine.Table()
             + " " + floor.bulb + " " + stairs);
        }
    }
}
```

The Lamp() method sets the various strings and ints. If you call it with an int, then it sets the Bulb field in whatever object Hinge points to.

If you pass a string to Lamp, it sets the Stairs field to whatever is in that string.

Remember, the as keyword only works with classes, not structs.

```
public class Hinge {
    public int bulb;
    public Table garden;
    public void Set(Table a) {
        garden = a;
    }
    public string Table() {
        return garden.stairs;
    }
}
```

Both Hinge and Table have a Set() method. Hinge's Set() sets its Table field called Garden, and Table's Set() method sets its Hinge field called Floor.

Bonus question: Circle the lines where boxing happens.

NOTE

Since the Lamp() method takes an object parameter, boxing automatically happens when it's passed an int or a string.

Chapter 14. Querying Data and Building Apps With Linq: 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 wrangled your data into manageable chunks, your Windows Store apps **have controls for navigating data** that let your users navigate, explore, and even zoom into the details.

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.

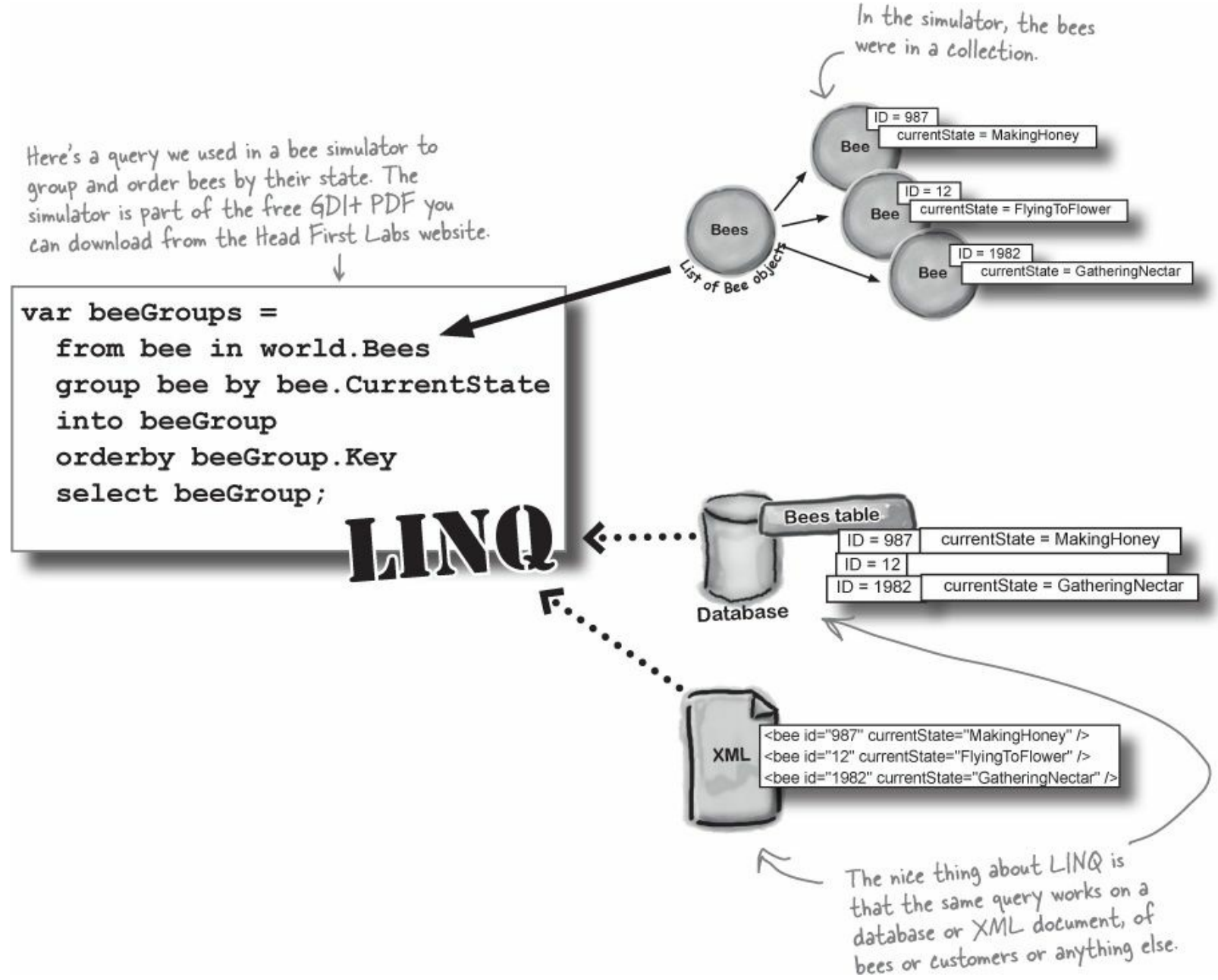
...but his collection's all over the place

Jimmy may be passionate, but he's not exactly organized. He's trying to keep track of the most prized "crown jewel" comics of his collection, but he needs help. Can you build Jimmy an app to manage his comics?

LINQ can pull data from multiple sources

LINQ to the rescue! LINQ (or **L**anguage **In**tegrated **Q**uery) is a flexible feature of C# that lets you **write queries to pull data out of a collection**. But LINQ also lets you work with more than just collections — in fact, you can use it to query *any* object that implements the `IEnumerable<T>` interface.

So let's use LINQ to help Jimmy get a handle on his comic book collection.



LINQ works with pretty much every kind of data source you could use in .NET. Your code needs a `using System.Linq;` line at the top of your file, but that's it. Even better, the IDE automatically puts a reference to LINQ in the header of the class files it creates.

.NET collections are already set up for LINQ

All of the collection types in .NET implement the `IEnumerable<T>` interface, which you learned about in Chapter 8. But take a minute to get a refresher: type **`System.Collections.Generic.IEnumerable<int>`** into your IDE window, right-click on the line, and select Go To Definition (or press F12). You'll see that the `IEnumerable` interface defines a `GetEnumerator()` method:

Notice how IEnumerable<T> extends an interface called IEnumerable? Use Go To Definition to explore it, too.

```
namespace System.Collections.Generic {
  interface IEnumerable<T> : IEnumerable {
    // Summary:
    //     Returns an enumerator that iterates through the collection.
    //
    // Returns:
    //     A System.Collections.Generic.IEnumerator<T> that can be
    //       used to iterate through the collection.
    IEnumerator<T> GetEnumerator();
  }
}
```

This is the only method in the interface. Each collection implements this method. You could create your own kind of object that implemented IEnumerable<T> too...and if you did, you could use LINQ with your object.

This method requires your object to define a way to move through the elements in it, one element at a time. That's all LINQ requires as a prerequisite. If you can move through a list of data, item by item, then you can implement `IEnumerable<T>`, and LINQ can query the collection.

BEHIND THE SCENES

LINQ uses **extension methods** to let you query, sort, and update data. Check it out for yourself. Create an `int` array called `linqtest`, put some numbers in the array, and then type this line of code (don't worry, you'll learn what it does in a minute):

```
IEnumerable<int> result = from i in linqtest where i < 3 select i;
```

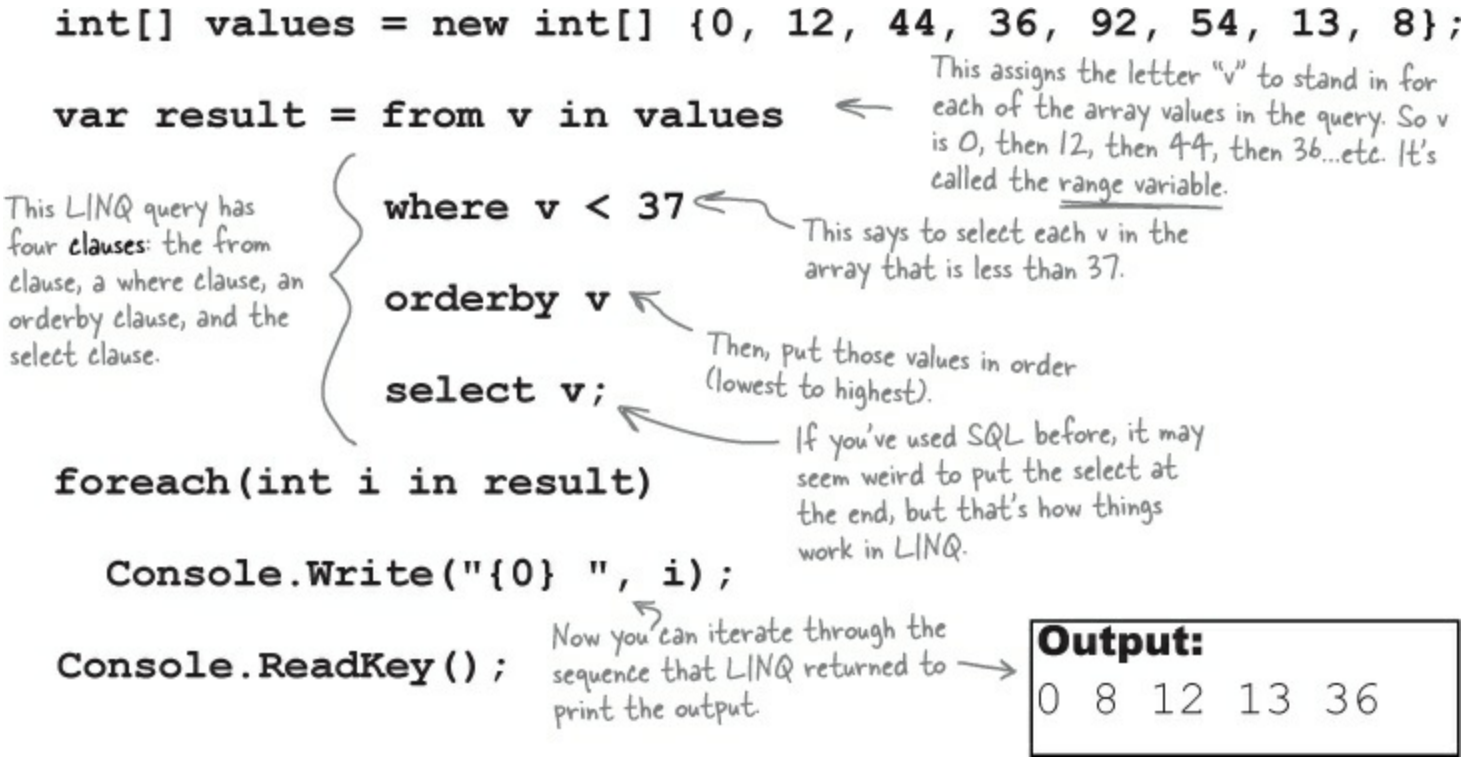
Now comment out the `using System.Linq;` line up in the header of the file you've created. When you try to rebuild the solution, you'll see that this line doesn't compile anymore. The methods you're calling when you use LINQ are just extension methods that are being used to extend the array.

NOTE

Now you can see why extension methods were so important in Chapter 13...they let .NET (and you) add all kinds of cool behavior to existing types.

LINQ makes queries easy

Here's a simple example of LINQ syntax. 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.



VAR

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.

In the example above, when this line is compiled:

```
var result = from v in values
```

The compiler replaces "var" with this:

```
IEnumerable<int>
```

And while we're on the subject of interfaces for collections, remember how we talked about how IEnumerable<T> is the interface that supports iteration? A lot of these great LINQ queries are implemented via extension methods that extend IEnumerable<T>, so you'll see that interface a lot.

Flip back to Chapter 8 to get a refresher on the `IEnumerable<T>` interface. Plus, you can read more about it in leftover #7 in the appendix.

LINQ is simple, but your queries don't have to be

Jimmy just sold his start-up that sells apps in the Windows Store to a big investor, and wants to take some of his profits and buy the most rare and expensive issues of Captain Amazing that he can find. How can LINQ help him scour his data and figure out which comics are the most expensive?

We're showing you each LINQ query in this chapter twice: first in a console app to help you see how it works, then in a larger Windows Store app so you can see how a LINQ query works in context—because your brain keeps track of things things better in context!

- ❶ Jimmy downloaded a list of Captain Amazing issues from a Captain Amazing fan page. He put them in a `List<T>` of `Comic` objects that have two fields, `Name` and `Issue`.

```
class Comic {
    public string Name { get; set; }
    public int Issue { get; set; }
}
```

Jimmy used object initializers and a collection initializer to build his catalog:

There's no special reason this method is static, other than to make it easy to call from a console application's entry point method.

```
private static IEnumerable<Comic> BuildCatalog()
{
    return 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 an Object", Issue = 97 },
    };
}
```

We left the () parentheses off of the collection and object initializers after <Comic>, because you don't need 'em.

Issue #74 of Captain Amazing is called "Black Monday."

Take a minute and flip to leftover #7 in the appendix to learn about a really useful bit of syntax that could come in handy here. This is a great opportunity to experiment!

- ❷ Luckily, there's a thriving marketplace for Captain Amazing comics on *Greg's List*, a website where people sell used comics. Jimmy knows that issue #57, "Hippie Madness," was misprinted and that almost all of the run was destroyed by the publisher, and he found that a rare copy recently sold on *Greg's List* for $13,525. After a few hours of searching, Jimmy was able to build a

`Dictionary<>` that mapped issue numbers to values.

```
private static Dictionary<int, decimal> GetPrices()
{
    return new Dictionary<int, decimal> {
        { 6, 3600M },
        { 19, 500M },
        { 36, 650M },
        { 57, 13525M },
        { 68, 250M },
        { 74, 75M },
        { 83, 25.75M },
        { 97, 35.25M },
    };
}
```

Remember this syntax for collection initializers for dictionaries from Chapter 8?

Issue #57 is worth $13,525.

BRAIN POWER

Look closely at the LINQ query in LINQ makes queries easy, then look at Jimmy's methods on this page. What do you think he would put into a query to find the most expensive issues?

ANATOMY OF A QUERY

Jimmy could analyze his comic book data with one LINQ query. The `where` clause tells LINQ which items from the collection should be included in the results. But that clause doesn't have to be just a simple comparison. It can include any valid C# expression — like using the `values` dictionary to tell it to return only comics worth more than $500. And the `orderby` clause works the same way — we can tell LINQ to order the comics by their value.

```
IEnumerable<Comic> comics = BuildCatalog();
Dictionary<int, decimal> values = GetPrices();

var mostExpensive =
    from comic in comics
    where values[comic.Issue] > 500
    orderby values[comic.Issue] descending
    select comic;

foreach (Comic comic in mostExpensive)
    Console.WriteLine("{0} is worth {1:c}",
        comic.Name, values[comic.Issue]);
```

The LINQ query pulls Comic objects out of the comics list, using the data in the values dictionary to decide which comics to select.

The first clause in the query is the from clause. This one tells LINQ to query the comics collection, and that the name comic will be used in the query to specify how to treat each individual piece of data in the collection.

You can choose any name you want when you use a from clause. We chose "comic."

The name comic was defined in the from clause specifically so it could be used in the where and orderby clauses.

The where and orderby clauses can include ANY C# statement, so we can use the values dictionary to select only those comics worth more than $500, and we can sort the results so the most expensive ones come first.

When you add "{1:c}" to the WriteLine output, that tells it to print the second parameter in the local currency format.

The query returned its results into an IEnumerable<T> called mostExpensive. The select clause determines what goes into the results—since it selected comic, the query returned Comic objects.

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
```

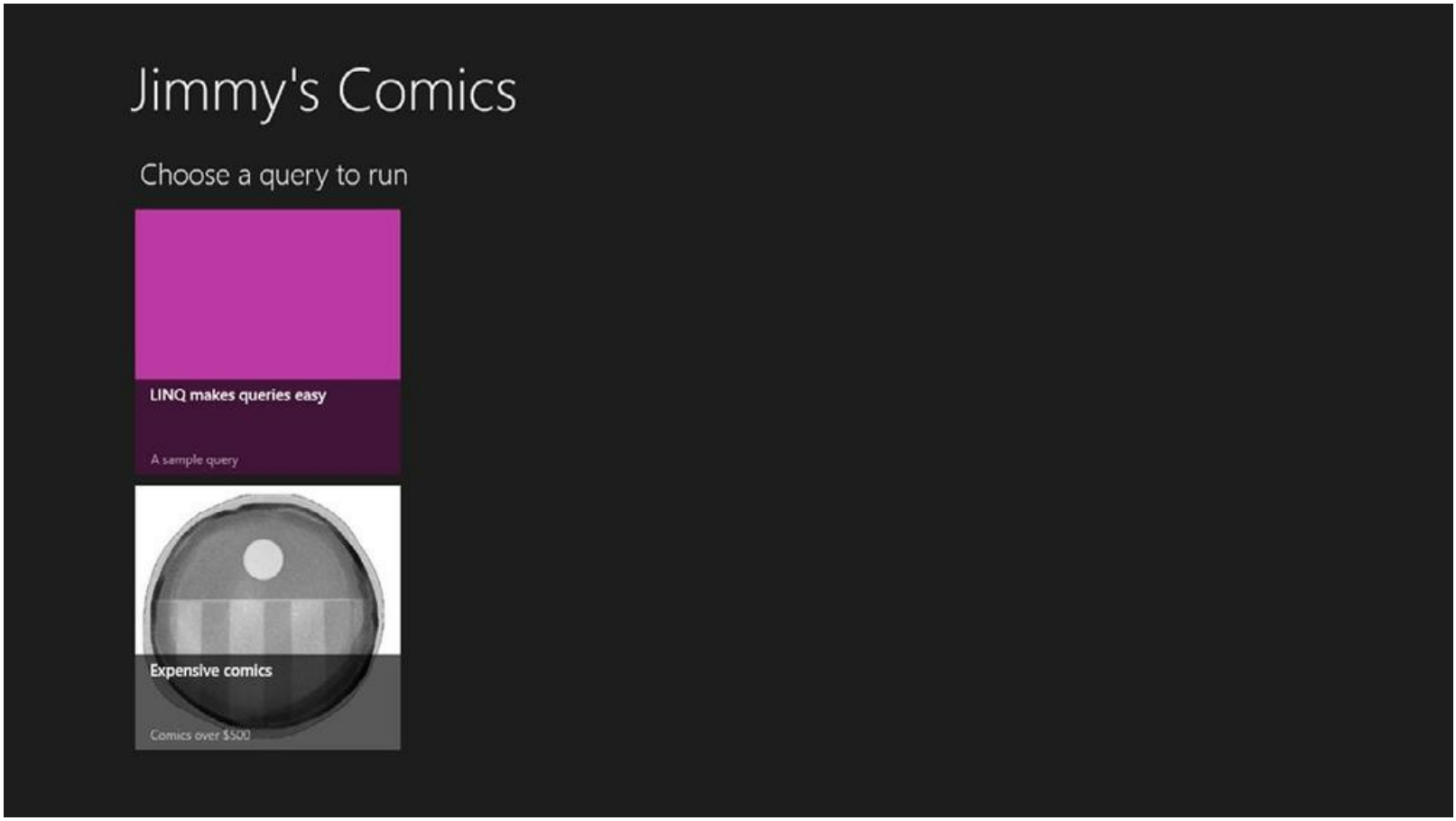
WATCH IT!

LINQ may *look* like SQL, but it doesn't *work* like SQL.

*If you've worked with SQL before, you'll recognize keywords like `select`, `from`, `where`, `descending`, and `join`. But LINQ is very dfiferent — treat it like SQL, and you'll end up with code that **doesn't do what you expect**! One big difference is that SQL operates on tables, while LINQ uses **enumerable objects** or **sequences**. Rows in a table aren't in any particular order; sequences have a specific order. And LINQ's ability to **call methods** on the objects in a sequence and **deferred execution** (which you'll learn about) make it a very different beast entirely!*

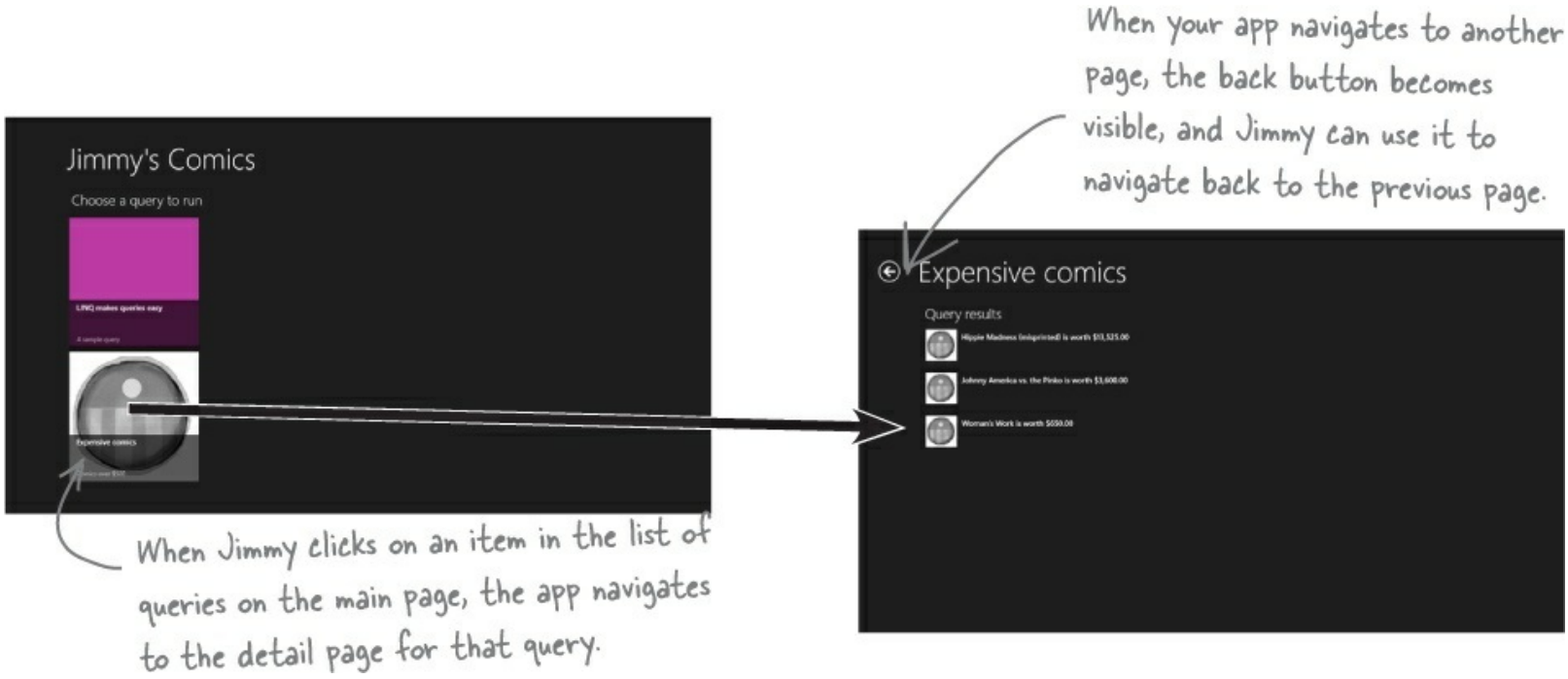
Jimmy could use some help

Let’s help Jimmy out by building him a Windows Store app to help him manage his comic collection — and to show him just how useful LINQ can be when it comes to data.



Windows Store apps use page-based navigation

Open up the toolbox and find a XAML equivalent of the WinForms TabControl. Can't find it? That's not an accident. Tabs are a staple of desktop applications, but when you're not using them they can clutter up the screen. Windows store apps use a **navigation system that's based on pages**, which can reduce that clutter and provide a more intuitive interface for your program.



Read more about navigation design for Windows Store apps here: http://msdn.microsoft.com/en-us/library/windows/apps/hh761500.aspx

Use the IDE to explore app page navigation

Here's another chance to use the IDE as a learning tool. Go to any Windows Store app that you've built and open up *App.xaml.cs*. That's your main application file, and every Windows Store app has one. It's a subclass of a class called `Application` in the `Windows.UI.Xaml` namespace, and it's always in a file named *App.xaml*. Your app's `Application` object initializes the app, and manages app lifetime: launching, suspending, and resuming. And it does another really useful thing: it creates a **`Frame`** object (from `Windows.UI.Xaml.Controls`), which is what your app uses to support navigation in your XAML pages.

Find the `OnLaunched()` method in your `App` class. It's run every time your app is launched, and it sets up the frame:

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
#if DEBUG
    if (System.Diagnostics.Debugger.IsAttached) {
        this.DebugSettings.EnableFrameRateCounter = true;
    }
#endif

    Frame rootFrame = Window.Current.Content as Frame;

    // Do not repeat app initialization when the Window already has content,
    // just ensure that the window is active
    if (rootFrame == null)
    {
        // Create a Frame to act as the navigation context and navigate to the first page
        rootFrame = new Frame();
        // Set the default language
        rootFrame.Language = Windows.Globalization.ApplicationLanguages.Languages[0];

        rootFrame.NavigationFailed += OnNavigationFailed;

        if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
            //TODO: Load state from previously suspended application
        }

        // Place the frame in the current Window
        Window.Current.Content = rootFrame;
    }

    if (rootFrame.Content == null)
    {
        // When the navigation stack isn't restored navigate to the first page,
        // configuring the new page by passing required information as a navigation
        // parameter
        rootFrame.Navigate(typeof(MainPage), e.Arguments);
    }
    // Ensure the current window is active
    Window.Current.Activate();
}
```

This line enables the frame rate counters in the corners of the screen. Try commenting it out.

Use the "Go To Definition" menu option to explore the Window and Frame classes that represent the main window of the current application and the navigation frame.

This is where the app creates a new navigation frame that will contain all of your app's pages.

When you delete MainPage.xaml and replace it with a Basic Page with the same name, you're adding a new MainPage class to replace the one you removed, so the Navigate() method can create an instance of your newly added page instead of the default one.

This is how the app brings up your main page. The `Frame.Navigate()` method creates a new instance of a page and then displays its contents. The `typeof` keyword returns the type of a class, so that's how it knows what page type to instantiate.

Your code-behind can use the same `Frame.Navigate()` method to navigate between pages, too. Every XAML page has a **property called `Frame`**. If you were to add a page called AnotherPage to your app, here's how you'd navigate to it. Notice the **argument `query`** passed to `Navigate()`. That's **a parameter being passed** to the newly created page.

```
if (this.Frame != null)
    this.Frame.Navigate(typeof(AnotherPage), query);
```

If you add a page called AnotherPage, the IDE adds the AnotherPage class to your project, and this code will navigate to a new instance of AnotherPage, passing it "query" as an argument.

Flip to leftover #5 in the appendix to learn more about the `typeof` keyword.

Start building Jimmy an app

You'll build an app that uses page navigation to execute different LINQ queries, starting with the two queries that you've seen so far.

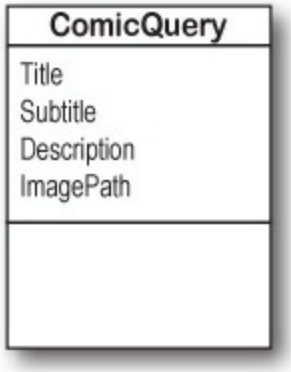
DO THIS!

- **❶ Create a new Windows Store app project.**
  Use the Blank App template, delete the *MainPage.xaml* file, and add a new Basic Page called *MainPage.xaml*. Then **add another Basic Page called *QueryDetail.xaml***. Don't forget to choose **Rebuild Solution** from the Build menu before you go on to step #2.
- **❷ Add the `Comic` class.**
  You already saw the `Comic` class from a few pages ago, so go ahead and add that class to your project.

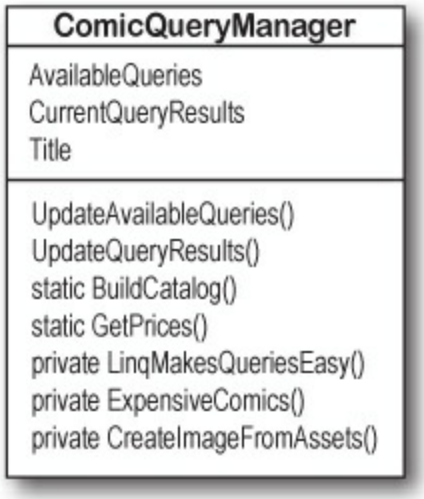
```
class Comic {
    public string Name { get; set; }
    public int Issue { get; set; }
}
```

- **❸ Add the `ComicQuery` class.**
  You'll need this class to represent a query, and when this app is done you'll have one instance of `ComicQuery` for each LINQ query in the chapter. Have a look at the screenshot two pages ago. Each of the queries has an icon, so you'll need a way to represent that in your class. XAML has an **Image control** and all you need to do is **set its Source property** to the a string that contains path of an image in your project. The images will live in the Assets folder, all we need to do is set the ImagePath property to `"Assets/"` plus the name of the image file and bind the Source property of an Image control to it.

```
class ComicQuery {
    public string Title { get; private set; }
    public string Subtitle { get; private set; }
    public string Description { get; private set; }
    public string ImagePath { get; private set; }

    public ComicQuery(string title, string subtitle,
                      string description, string imagePath) {
        Title = title;
        Subtitle = subtitle;
        Description = description;
        ImagePath = imagePath;
    }
}
```



- **❹ Add a query manager class so your controls have something to bind to.**
  Jimmy's app will follow the same pattern that you used with the last two apps. The `ComicQueryManager` class will do all of the work of running the queries and exposing properties that contain the results. Each XAML page will have a static resource that contains an instance of `ComicQueryManager`, calling its methods to run the queries and data binding the results to controls.



```
using System.Collections.ObjectModel;

class ComicQueryManager {

    public ObservableCollection<ComicQuery> AvailableQueries { get; private set; }

    public ObservableCollection<object> CurrentQueryResults { get; private set; }

    public string Title { get; set; }

    public ComicQueryManager() {
        UpdateAvailableQueries();
        CurrentQueryResults = new ObservableCollection<object>();
    }

    private void UpdateAvailableQueries() {
        AvailableQueries = new ObservableCollection<ComicQuery> {
            new ComicQuery("LINQ makes queries easy", "A sample query",
                "Let's show Jimmy how flexible LINQ is",
                "Assets/purple_250x250.jpg"),

            new ComicQuery("Expensive comics", "Comics over $500",
                "Comics whose value is over 500 bucks."
                 + " Jimmy can use this to figure out which comics are most coveted.",
                "Assets/captain_amazing_250x250.jpg"),
        };
    }

    public void UpdateQueryResults(ComicQuery query) {
        Title = query.Title;

        switch (query.Title) {
            case "LINQ makes queries easy": LinqMakesQueriesEasy(); break;
            case "Expensive comics": ExpensiveComics(); break;
        }
    }
```

The ListView of queries on the main page is bound to the AvailableQueries property.

The CurrentQueryResults and Title properties are used to display the query results on the QueryDetail page.

This collection initializer creates the ComicQuery objects to display in the main page.

The QueryDetail page uses this method to run a LINQ query.

Did you know that you can put the `using` statements inside of the `namespace` block?

Before you flip the page to see the rest of the class, can you figure out what the `LinqMakesQueriesEasy()` and `ExpensiveComics()` methods will look like? The app will call those methods to run LINQ queries. But first, read the next page to learn about anonymous types — you'll use them in those methods.

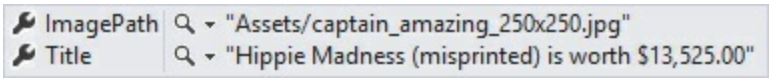
Use the new keyword to create anonymous types

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 the statement you'll use in the `ExpensiveComics` query on the next page that creates an instance of an anonymous type to add to the collection in the `CurrentQueryResults` property:

```
new {
    Title = String.Format("{0} is worth {1:c}",
    comic.Name, values[comic.Issue]),
    ImagePath = "Assets/captain_amazing_250x250.jpg",
}
```

When you run the program, you can see the objects that it creates just like any other objects. Here's what an instance of that anonymous type looks like in the Watch window:



This works just like any other object initializer. You can call methods like `CreateImageFromAssets()` and `String.Format()` in the object initializer to populate the object's properties. (Of course, you can also set them to values if you need to, too.)

The one thing you *can't* do is refer to the name of the type, because the type doesn't have a name! That's where the `var` keyword comes in very handy, because you can use it to hold a reference to an anonymous type — like this:

```
var myAnonymousObject = new {
     Name = "Bob",
     Cash = 186.3M,
     Age = 37,
};
Console.WriteLine(myAnonymousObject.Name);
```

NOTE

a-non-y-mous, adjective.

not identified by name. *Secret Agent Dash Martin uses his alias to become **anonymous** to keep the KGB agents from recognizing him.*

Flip to leftover #9 in the appendix to learn more about anonymous types.

That code creates an instance of an anonymous type, saves a reference to that new object in the `myAnonymousObject` variable, and uses it to write the `Name` property to the output.

Flip the page to finish the app

Here's the rest of the ComicQueryManager class:

```
    public static IEnumerable<Comic> BuildCatalog() {
        return 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 an Object", Issue = 97 },
        };
    }

    private static Dictionary<int, decimal> GetPrices() {
        return new Dictionary<int, decimal> {
            { 6, 3600M }, { 19, 500M }, { 36, 650M }, { 57, 13525M },
            { 68, 250M }, { 74, 75M }, { 83, 25.75M }, { 97, 35.25M },
        };
    }
```

These are the same BuildCatalog() and GetPrices() methods from a few pages ago.

The CurrentQueryResults property is used to display the query results with data binding. It doesn't need to be cleared because a new page object is created each time page navigation occurs, and it has its own new instance of ComicQueryManager, with a newly created empty collection.

```
    private void LinqMakesQueriesEasy() {
        int[] values = new int[]
                       { 0, 12, 44, 36, 92, 54, 13, 8 };
        var result = from v in values
                     where v < 37
                     orderby v
                     select v;

        foreach (int i in result)
            CurrentQueryResults.Add(
               new {
                     Title = i.ToString(),
                     ImagePath = "Assets/purple_250x250.jpg",
                   }
            );
    }
```

We still don't quite know how the CreateImageFromAssets() method works, but I bet we'll find out on the next page.

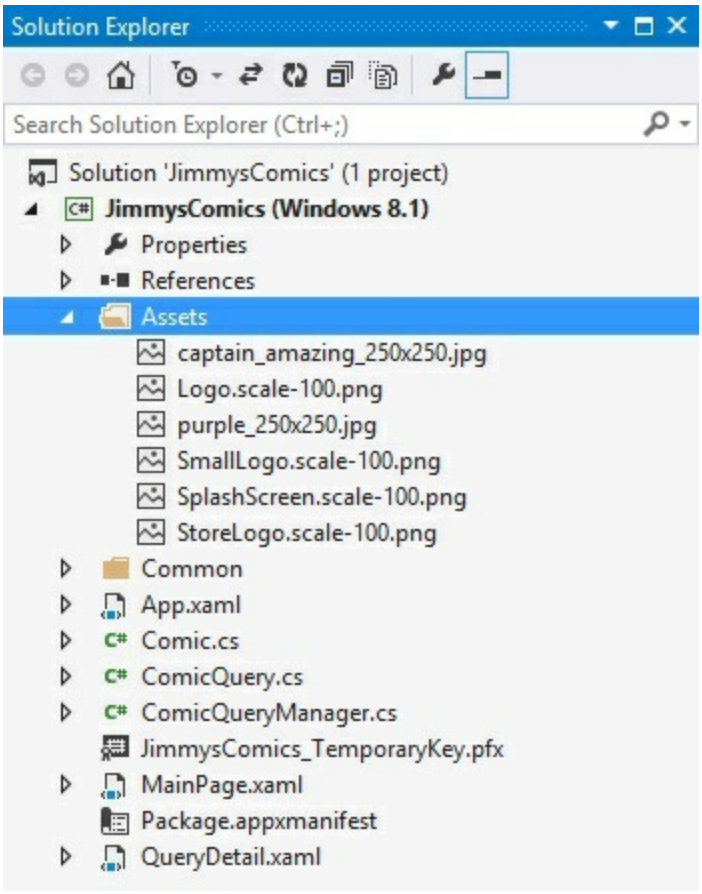
Each of these methods executes one of the LINQ queries from earlier in the chapter. Instead of writing each result to the console, it adds each result to the CurrentQueryResults property, an ObservableCollection<object> collection. But take a close look at the new { } statement. Somehow you're using the new keyword with an object initializer. Usually there's a type after the new keyword, but these statements leave it out so they can create instances of anonymous types.

```
    private void ExpensiveComics() {
        IEnumerable<Comic> comics = BuildCatalog();
        Dictionary<int, decimal> values = GetPrices();

        var mostExpensive = from comic in comics
            where values[comic.Issue] > 500
            orderby values[comic.Issue] descending
            select comic;

        foreach (Comic comic in mostExpensive)
            CurrentQueryResults.Add(
                    new {
                        Title = String.Format("{0} is worth {1:c}",
                                              comic.Name, values[comic.Issue]),
                        ImagePath = "Assets/captain_amazing_250x250.jpg",
                    }
                );
    }
}
```

- **❺ Add image files to the *Assets* folder in your project.**
  Find the image files *purple_250x250.jpg* and *captain_amazing_250x250. jpg* for this project (you can download them from the your class website and save them in a folder. Then go to the Solution Explorer, right-click on the folder, choose

Add→Existing Item from the menu, and add the files.
Now have a closer look at the code in the LinqMakesQueriesEasy() method that uses an anonymous type to add an object to the results:

```
new {
        Title = i.ToString(),
        ImagePath = "Assets/purple_250x250.jpg",
}
```



This object will be displayed in a ListView with an Image control in the item template. When one of these is displayed as an item in that list, its ImagePath property will be bound to the Source property.

- **❻ Add code-behind for the main page and query detail page.**
When the user clicks on the list of queries in the main page, we want the app to navigate to the query detail page. **Open *MainPage.xaml.cs* and add this event handler** to the code-behind. The `SelectionChanged` event handler for a ListView can access the items that were selected using the `e.AddedItems`. The ListView is bound to an `ObservableCollection` of `ComicQuery` objects, so `e.AddedItems[0]` will always contain the `ComicQuery` that the user clicked on. You'll pass that as a parameter to the new page using `Frame.Navigate()`.

```
private void ListView_ItemClick(object sender, ItemClickEventArgs e) {
     ComicQuery query = e.ClickedItem as ComicQuery;
     if (query != null)
         this.Frame.Navigate(typeof(QueryDetail), query);
}
```

NOTE

You can add an argument to the Frame.Navigate() method to pass an object as a parameter to the page you're navigating to.

When the main page calls `Frame.Navigate()` to navigate to the query detail page, a `ComicQuery` object is passed as a parameter that's used in the the `OnNavigatedTo()` method. This method is called when the page is navigated to, and its code-behind is part of the Basic Page template. **Find it in QueryDetail.xaml.cs** by expanding ⊞ NavigationHelper registration. Use `e.Parameter` to access the navigation parameter:

```
protected override void OnNavigatedTo(NavigationEventArgs e) {
    ComicQuery comicQuery = e.Parameter as ComicQuery;
    if (comicQuery != null) {
        comicQueryManager.UpdateQueryResults(comicQuery);
        pageTitle.Text = comicQueryManager.Title;
    }
    navigationHelper.OnNavigatedTo(e);
}
```

Add this code to the OnNavigatedTo() method to update the query results and set the page title.

- **❼ Finish the XAML for the main page.**
  Open *MainPage.xaml*. Here are the resources for the page:

```
<Page.Resources>
    <local:ComicQueryManager x:Name="comicQueryManager"/>
    <x:String x:Key="AppName">Jimmy's Comics</x:String>
</Page.Resources>
```

The ListView control automatically adds vertical scrollbars if the list items scroll off the bottom. Try adding `Height="*"` to the second RowDefinition in the Grid's row definitions. The scrollbars disappear! That's because the row expands to fit all of the ListView's items.

Use a `Grid` to lay out the content — it has a TextBlock and a GridView. The GridView control is just like a ListView. The only difference is that instead of laying out items And just like the ListView, it uses a DataTemplate to display an Image and StackPanel with two TextBlocks for each item:

Setting the SelectionMode property to None disables the list selection functionality, and IsItemClickEnabled set to true causes the ItemClick event to fire when an item is clicked.

This GridView's ItemsSource property is bound to the AvailableQueries property, an ObservableCollection. Any object it contains is displayed using the data template.

All we need to do to display an image for an item in the GridView is make sure that its ImagePath property is set to the name of an image file in the project's Assets folder.

```
<Grid Grid.Row="1" Margin="120,0"
      DataContext="{StaticResource ResourceKey=comicQueryManager}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
               Text="Choose a query to run" Margin="10,0,0,20"/>

    <GridView Grid.Row="1" ItemsSource="{Binding AvailableQueries}" SelectionMode="None"
              IsItemClickEnabled="True" ItemClick="ListView_ItemClick">
        <GridView.ItemTemplate>
            <DataTemplate>
                <Grid HorizontalAlignment="Left" Width="250" Height="250">

                    <Image Source="{Binding ImagePath}" Stretch="UniformToFill"/>

                    <StackPanel VerticalAlignment="Bottom"
                     Background="{ThemeResource ListViewItemOverlayBackgroundThemeBrush}">

                        <TextBlock Text="{Binding Title}"
                         Foreground="{ThemeResource ListViewItemOverlayForegroundThemeBrush}"
                         Style="{StaticResource BaseTextBlockStyle}" Height="60"
                         Margin="15,0,15,0" FontWeight="SemiBold"/>

                        <TextBlock Text="{Binding Subtitle}"
                         Foreground=
                           "{ThemeResource ListViewItemOverlaySecondaryForegroundThemeBrush}"
                         Style="{StaticResource BaseTextBlockStyle}" TextWrapping="NoWrap"
                         Margin="15,0,15,10" FontSize="12"/>

                    </StackPanel>
                </Grid>
            </DataTemplate>
        </GridView.ItemTemplate>
    </GridView>
</Grid>
```

BRAIN POWER

We used some pretty long style names in the data template. Where do you think they came from?

- **❽ Finish the XAML and code-behind for the query detail page.**
  Open *QueryDetail.xaml*. Here are the resources for the page:

```
<Page.Resources>
    <local:ComicQueryManager x:Name="comicQueryManager"/>
    <x:String x:Key="AppName">Query Detail</x:String>
</Page.Resources>
```

  Use another Grid to lay out the content:

```
<Grid Grid.Row="1" Margin="120,0"
      DataContext="{StaticResource ResourceKey=comicQueryManager}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <TextBlock Style="{StaticResource SubheaderTextBlockStyle}"
               Text="Query results" Margin="10,0,0,20"/>

    <ListView Grid.Row="1" Margin="0,-10,0,0" ItemsSource="{Binding CurrentQueryResults}"
       SelectionMode="None">
       <ListView.ItemTemplate>
          <DataTemplate>
             <Grid Margin="6">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto"/>
                    <ColumnDefinition Width="*"/>
                </Grid.ColumnDefinitions>

                <Image Source="{Binding ImagePath}" Stretch="UniformToFill" />

                <StackPanel Grid.Column="1" Margin="10,0,0,0">

                        <TextBlock Text="{Binding Title}"
                                   Style="{StaticResource TitleTextBlockStyle}"
                                   TextWrapping="NoWrap" MaxHeight="40"/>

                        <TextBlock Text="{Binding Subtitle}"
                                   Style="{StaticResource CaptionTextBlockStyle}"
                                   TextWrapping="NoWrap"/>

                    </StackPanel>
                </Grid>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</Grid>
```

This ListView displays the items in the query results. Each item is displayed using the DataTemplate, which works a lot like the one in the main page.

- **❾ Run your program! And then use the IDE to explore it and understand how it works.** Put a breakpoint in ListView_ItemClick() in *MainPage.xaml.cs* and add watches to see exactly what is in the ComicQueryManager object. Look at how the current query and the query results work, and match up the properties in the anonymous types to the bound properties in the GridView and ListView.

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.

All collections are enumerable — they implement `IEnumerable<T>` — but not everything that's enumerable is technically a collection unless it implements the `ICollection<T>` interface, which means implementing `Add()`, `Clear()`, `Contains()`, `CopyTo()`, and `Remove()` ... and, of course, `ICollection<T>` extends `IEnumerable<T>`. LINQ deals with sequences of values or objects, not collections, and all you need for a sequence is an object that implements `IEnumerable<T>`.

- **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.

```
string[] sandwiches = { "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);
```

This adds the string " on rye" to every item in the results from the query.

This change is made to the items in the results of your query...but not to the items in the original collection or database.

Notice that all the items returned have " on rye" added 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 collections.**
  Remember, we said LINQ provides extension methods for your collections (and anything else that implements `IEnumerable<T>`). And some of those are pretty handy on their own, without actually requiring a query:

```
Random random = new Random();
List<int> listOfNumbers = new List<int>();
int length = random.Next(50, 150);
for (int i = 0; i < length; i++)
    listOfNumbers.Add(random.Next(100));

Console.WriteLine("There are {0} numbers",
                  listOfNumbers.Count());
Console.WriteLine("The smallest is {0}",
                  listOfNumbers.Min());
Console.WriteLine("The biggest is {0}",
                  listOfNumbers.Max());
Console.WriteLine("The sum is {0}",
                  listOfNumbers.Sum());
Console.WriteLine("The average is {0:F2}",
                  listOfNumbers.Average());
```

None of these methods are part of the .NET collections classes...they're all defined by LINQ.

These are all extension methods for IEnumerable<T> in the System.Linq namespace using a static class called Enumerable. But **don't take our word for it!** Click on any of them and use Go To Definition to see for yourself.

NOTE

A sequence is an ordered set of objects or values, which is what LINQ returns in an IEnumerable<T>.

- **Store all or part of your results in a new sequence**.
  Sometimes you'll want to keep your results from a LINQ query around. You can use the `ToList()` command to do just that:

```
var under50sorted =
  from number in listOfNumbers
  where number < 50
  orderby number descending
  select number;

List<int> newList = under50sorted.ToList();
```

This time, we're sorting a list of numbers in descending order, from highest to lowest.

ToList() converts a LINQ var into a List<T> object, so you can keep results of a query around. There's also ToArray() and ToDictionary() methods, which do just what you'd expect.

You can even take just a subset of the results, using the `Take()` method:

```
var firstFive = under50sorted.Take(5);

List<int> shortList = firstFive.ToList();
foreach (int n in shortList)
    Console.WriteLine(n);
```

Take() pulls out the supplied number of items from the first set of the results from a LINQ query. You can put these into another var, and then convert that into a list.

- **Check out Microsoft's official 101 LINQ Samples page.**
  There's way more that LINQ can do. Luckily, Microsoft gives you a great little reference to help

you along.

http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b

WATCH IT!

LINQ queries aren't run until you access their results!

It's called "deferred evaluation" — the LINQ query doesn't actually do any looping until a statement is executed that uses the results of the query. That's why ToList() is important: it tells LINQ to evaluate the query immediately.

THERE ARE NO DUMB QUESTIONS

Q: **Q: That's a lot of new keywords — `from`, `where`, `orderby`, `select`...it's like a whole different language. Why does it look so different from the rest of C#?**

A: **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.
LINQ queries look different because a single LINQ query usually does a whole bunch of things at once. Let's take a closer look at a straightforward query:

```
var under10 =
 from number in numberArray
 where number < 10
 select number;
```

It looks really 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 `numberArray` that are less than 10. First, you need to loop through the entire array. Then, each number is compared to 10. Then those results need to be gathered together so your code can use them.
And that's why LINQ looks a little odd: because C# has to cram a whole lot of behavior into a very small space.

LINQ lets you write queries that do very complex things using very little code.

Add the new queries to Jimmy's app

Jimmy's curious about how LINQ can help manage his data. Add the three queries from the previous pages to the app to show him what LINQ can do. All you need to do is update the `ComicQueryManager` class (and add another image to the *Assets* folder). Start by adding three `ComicQuery` objects to the object initializer for the `AvailableQueries` property:

> **NOTE**
>
> This is an example of separation of concerns. You can add queries by modifying ComicQueryManager without changing any XAML or code-behind because you encapsulated all of the query code in that class.

Do this

```
private void UpdateAvailableQueries() {
    AvailableQueries = new ObservableCollection<ComicQuery> {
        new ComicQuery("LINQ makes queries easy", "A sample query",
            "Let's show Jimmy how flexible LINQ is",
            "Assets/purple_250x250.jpg"),

        new ComicQuery("Expensive comics", "Comics over $500",
            "Comics whose value is over 500 bucks."
                + " Jimmy can use this to figure out which comics are most coveted.",
            "Assets/captain_amazing_250x250.jpg"),

        new ComicQuery("LINQ is versatile 1", "Modify every item returned from the query",
            "This code will add a string onto the end of each string in an array.",
            "Assets/bluegray_250x250.jpg"),

        new ComicQuery("LINQ is versatile 2", "Perform calculations on collections",
            "LINQ provides extension methods for your collections (and anything else"
              + " that implements IEnumerable<T>).",
            "Assets/purple_250x250.jpg"),

        new ComicQuery("LINQ is versatile 3",
            "Store all or part of your results in a new sequence",
            "Sometimes you'll want to keep your results from a LINQ query around.",
            "Assets/bluegray_250x250.jpg"),
    };
}
```

Add these three queries so they show up on the main page.

Next, you'll need to update the `switch` statement to run the queries when they're selected in the ListView:

```
public void UpdateQueryResults(ComicQuery query) {
    Title = query.Title;

    switch (query.Title) {
        case "LINQ makes queries easy": LinqMakesQueriesEasy(); break;
        case "Expensive comics": ExpensiveComics(); break;
        case "LINQ is versatile 1": LinqIsVersatile1(); break;
        case "LINQ is versatile 2": LinqIsVersatile2(); break;
        case "LINQ is versatile 3": LinqIsVersatile3(); break;
    }
}
```

Add these three cases to the switch statement. They'll get executed by the query detail page when it's navigated to.

You'll need to add these three methods. Compare them with the LINQ queries on the previous two pages:

```
private void LinqIsVersatile1() {
    string[] sandwiches = { "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)
        CurrentQueryResults.Add(CreateAnonymousListViewItem(sandwich, "bluegray_250x250.jpg"));
}
```

You'll need to download the bluegray_250x250.jpg file from the Head First Labs website and add it to your Assets folder. If you don't some of the items in the main page won't have images!

```
private void LinqIsVersatile2() {
    Random random = new Random();
    List<int> listOfNumbers = new List<int>();
    int length = random.Next(50, 150);
    for (int i = 0; i < length; i++)
        listOfNumbers.Add(random.Next(100));

    CurrentQueryResults.Add(CreateAnonymousListViewItem(
        String.Format("There are {0} numbers", listOfNumbers.Count())));
    CurrentQueryResults.Add(
        CreateAnonymousListViewItem(String.Format("The smallest is {0}", listOfNumbers.Min())));
    CurrentQueryResults.Add(
       CreateAnonymousListViewItem(String.Format("The biggest is {0}", listOfNumbers.Max())));
    CurrentQueryResults.Add(
       CreateAnonymousListViewItem(String.Format("The sum is {0}", listOfNumbers.Sum())));
    CurrentQueryResults.Add(CreateAnonymousListViewItem(
              String.Format("The average is {0:F2}", listOfNumbers.Average())));
}
```

Do this

You need to add one more method to make this code work. Each of these three `LinqIsVersatile` methods calls another method called `CreateAnonymousListViewItem()`. Its first parameter is the title, and it should be used as the `Title` property in a new anonymous object. The second is an **optional** parameter. It's the name of the image file to load into the `ImagePath` property of the anonymous object, and it should default to *purple_250x250.jpg* if it's not included. Can you come up with this method? The answer is on the next page.

```
private void LinqIsVersatile3() {
    List<int> listOfNumbers = new List<int>();
    for (int i = 1; i <= 10000; i++)
        listOfNumbers.Add(i);

    var under50sorted =
            from number in listOfNumbers
            where number < 50
            orderby number descending
            select number;

    var firstFive = under50sorted.Take(6);

    List<int> shortList = firstFive.ToList();
    foreach (int n in shortList)
        CurrentQueryResults.Add(CreateAnonymousListViewItem(n.ToString(), "bluegray_250x250.jpg"));
}
```

BULLET POINTS

- **`from`** is how you specify the `IEnumerable<T>` that you're querying. It's always followed by the name of a variable, followed by **`in`** and the name of the input (`from value in values`).
- **`select`** is how you specify what goes into the results (`select value`).
- **`where`** generally follows the `from` clause. That's where you use normal C# conditions to tell LINQ which items to pull (`where value < 10`).
- **`orderby`** lets you order the results. It's followed by the criteria that you're using to sort them, and optionally **`descending`** to tell it to reverse the sort (`orderby value descending`).
- **`Take`** lets you pull the first items out of the results of a LINQ query (`results.Take(10)`). LINQ gives you other methods for each sequence: `Min()`, `Max()`, `Sum()`, and `Average()`.
- You can `select` anything — you're not limited to selecting the name that you created in the `from` clause. Here's an example: if your LINQ query pulls a set of prices out of an array of `int` values and names them `value` in the `from` clause, you can return a sequence of price strings like this: `select String.Format("{0:c}", value)`.

NOTE

This is just like the {0:x} you used in Chapter 9 when you built the hex dumper. There's also {0:d} and {0:D} for short and long dates, and {0:P} or {0:Pn} to print a percent (with n decimal places).

MICRO EXERCISE SOLUTION

```
private object CreateAnonymousListViewItem(string title,
                        string imageFilename = "purple_250x250.jpg") {
    return new {
        Title = title,
        ImagePath = "Assets/" + imageFilename,
    };
}
```

Here's the optional parameter. If it's left out, the purple box is used.

THERE ARE NO DUMB QUESTIONS

Q: **Q: How does the `from` clause work?**

A: **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. 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
```

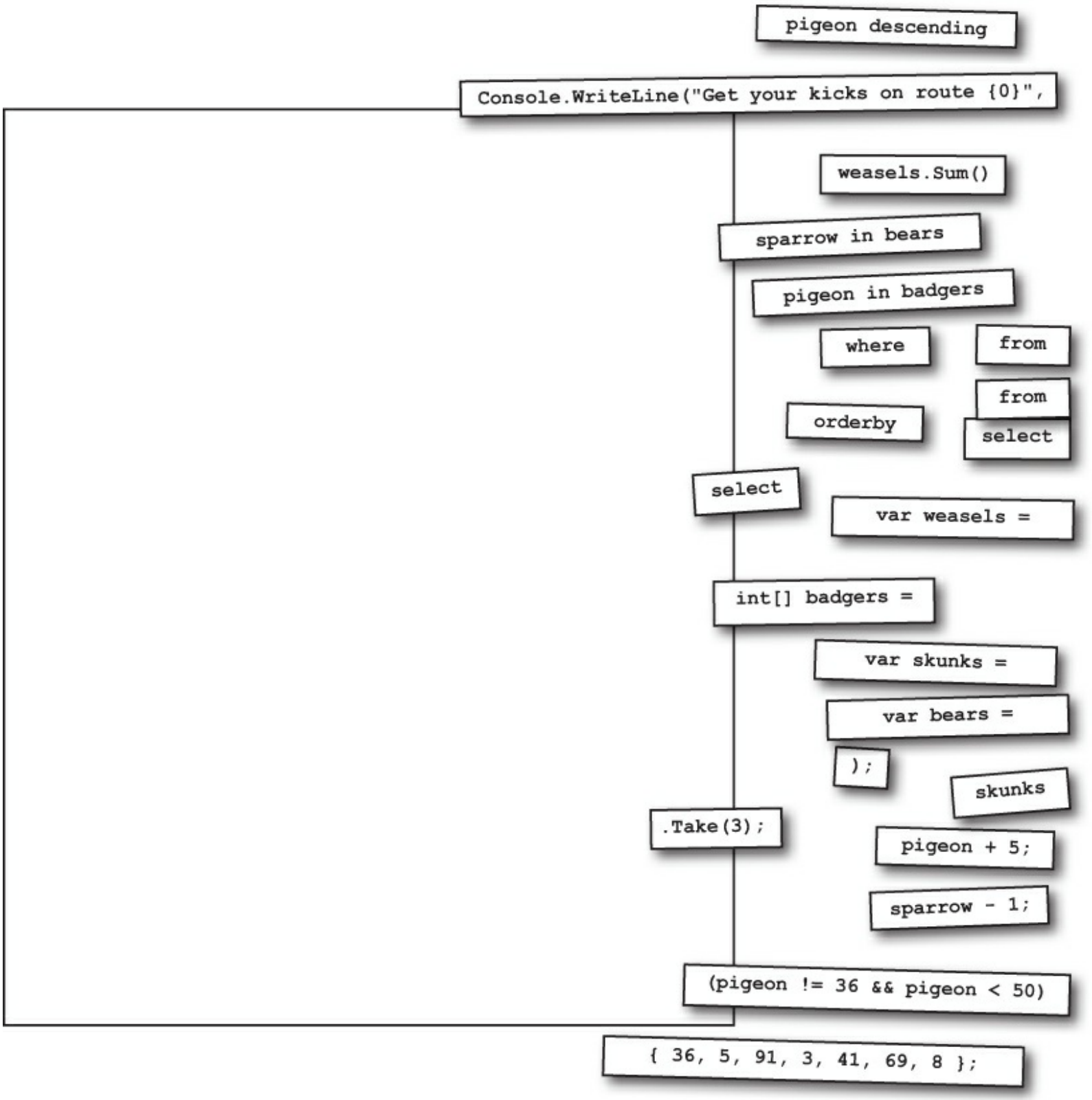
That clause does pretty much the same thing. It creates a temporary variable called `i` and assigns it sequentially to each item in the `values` collection. The `foreach` loop runs the same block of code for each item in the collection, while the LINQ query applies the same criteria in the `where` clause 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: **Q: How does LINQ decide what goes into the results?**

A: **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 `int`s 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.

LINQ MAGNETS

Rearrange the magnets so they produce the output at the bottom of the page.

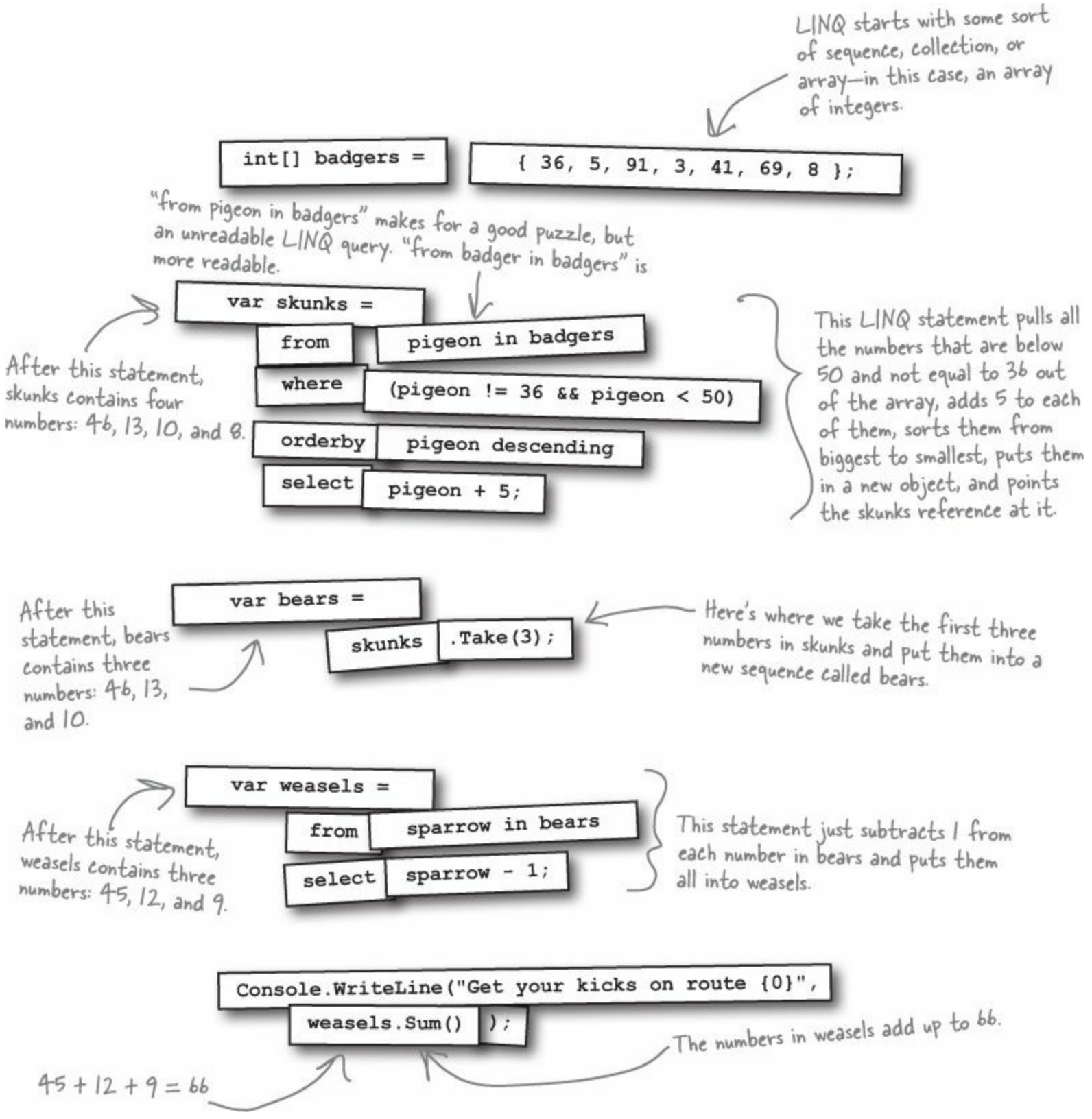


Output:

```
Get your kicks on route 66
```

LINQ MAGNETS SOLUTION

Rearrange the magnets so they produce the output at the bottom of the page.



Output:

```
Get your kicks on route 66
```

LINQ can combine your results into groups

You can use LINQ to build your results into groups, which can be really useful when you need to slice and dice your collections. Let's take a closer look at a query that breaks a collection into groups.

You can see this LINQ query in action (and learn more about how WinForms applications work) by building some nifty animation in a beehive simulator. Download the free GDI+ chapter from http://headfirstlabs.com/hfcsharp.

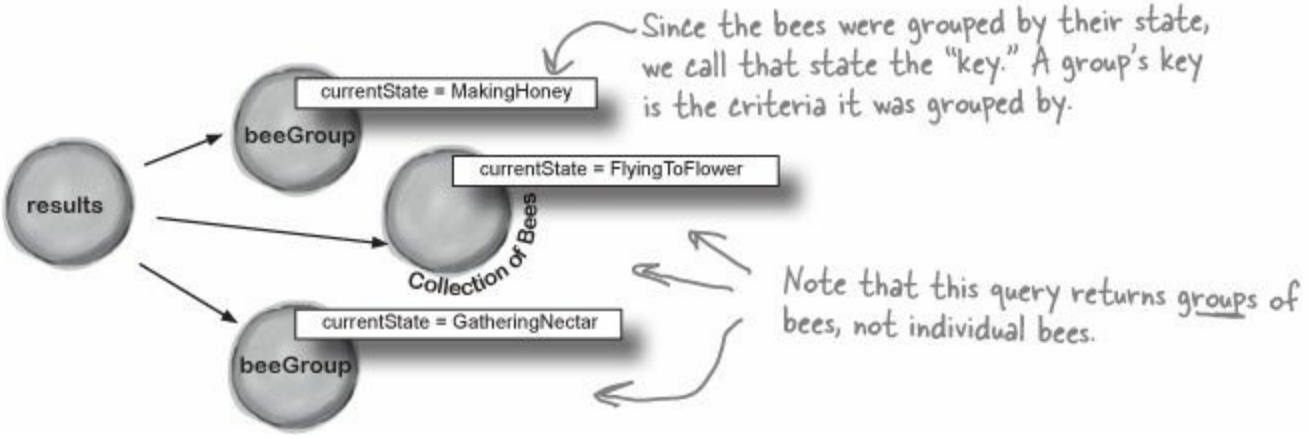
```
var beeGroups =

from bee in world.Bees

group bee by bee.CurrentState

into beeGroup

orderby beeGroup.Key

select beeGroup;
```

1 The query starts out just like the other queries you've seen—by pulling individual bee objects out of the `world.Bees` collection, a `List<Bee>` object.

2 The next line in the query has a new keyword: **`group`**. This tells the query to return ***groups*** of bees. What that means is that rather than returning one single sequence, the query will return a **sequence of sequences**. **`group bee by bee.CurrentState`** tells LINQ to return one group for each unique `CurrentState` property that it finds in the bees that it selects. Finally, we need to give LINQ a name for the group. That's what the next line is for: **`into beeGroup`** says that the name "beeGroup" refers to the new groups.

3 Now that we've got groups, we can manipulate them. Since we're returning a sequence of groups, we can use the `orderby` keyword to put the groups in the order of the `CurrentState` enum values (`Idle`, `FlyingToFlower`, etc.). **`orderby beeGroup.Key`** tells the query to put the sequence of groups in order, sorting them by the group key. Since we grouped the bees by their `CurrentState`, that's what is being used as a key.

4 Now we just have to use the `select` keyword to indicate what's being returned by the query. Since we're returning groups, we select the group name: **`select beeGroup;`**



Combine Jimmy's values into groups

Jimmy buys a lot of cheap comic books, some midrange comic books, and a few expensive ones, and he wants to know what his options are before he decides what comics to buy. He's taken those prices he got from recent sales on *Greg's List* and put them into a `Dictionary<int, decimal>` using his `GetPrices()` method. Let's now use LINQ to group them into three groups: one for cheap comics that cost under $100, one for midrange comics that cost between $100 and $1,000, and one for expensive comics that cost over $1,000. We'll create a `PriceRange` enum that we'll use as the key for the groups, and a method called `EvaluatePrice()` that'll evaluate a price and return a `PriceRange`.

- **❶ Every group needs a key — we'll use an enum for that.**
  The group's key is the thing that all of its members have in common. The key can be anything: a string, a number, even an object reference. We'll be looking at the prices that Jimmy got from *Greg's List*. Each group that the query returns will be a sequence of issue numbers, and the group's key will be a `PriceRange` enum. The `EvaluatePrice()` method takes a price as a parameter and returns a `PriceRange`:

```
enum PriceRange { Cheap, Midrange, Expensive }

static PriceRange EvaluatePrice(decimal price) {
    if (price < 100M) return PriceRange.Cheap;
    else if (price < 1000M) return PriceRange.Midrange;
    else return PriceRange.Expensive;
}
```

Try adding the code on this page to a new Console app — see if you can get it to work! You'll add the query to Jimmy's Windows Store app at the end of the chapter.

- **❷ Now we can group the comics by their price categories.**
  The LINQ query returns a **sequence of sequences**. Each of the sequences inside the results has a `Key` property, which matches the `PriceRange` that was returned by `EvaluatePrice()`. Look closely at the `group by` clause — we're pulling pairs out of the dictionary, and using the name `pair` for each of them: `pair.Key` is the issue number, and `pair.Value` is the price from *Greg's List*. Adding `group pair.Key` tells LINQ to create groups of issue numbers, and then bundles all of those groups up based on the price category that's returned by `EvaluatePrice()`:

```
Dictionary<int, decimal> values = GetPrices();
var priceGroups =
    from pair in values
    group pair.Key by EvaluatePrice(pair.Value)
        into priceGroup
        orderby priceGroup.Key descending
        select priceGroup;
foreach (var group in priceGroups) {
    Console.Write("I found {0} {1} comics: issues ", group.Count(), group.Key);
    foreach (var issueNumber in group)
        Console.Write(issueNumber.ToString() + " ");
    Console.WriteLine();
}
```

The query figures out which group a particular comic belongs to by sending its price to EvaluatePrice(). That returns a PriceRange enum, which it uses as the group's key.

Each of the groups is a sequence, so we added an inner foreach loop to pull each of the issue numbers out of the group.

Output:

```
I found 2 Expensive comics: issues 6 57
I found 3 Midrange comics: issues 19 36 68
I found 3 Cheap comics: issues 74 83 97
```

POOL PUZZLE

Your **job** is to take 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 code produce this ***output***:

Horses enjoy eating carrots, but they love eating apples.

```
class Line {
    public string[] Words;
    public int Value;
    public Line(string[] Words, int Value) {
        this.Words = Words; this.Value = Value;
    }
}
```

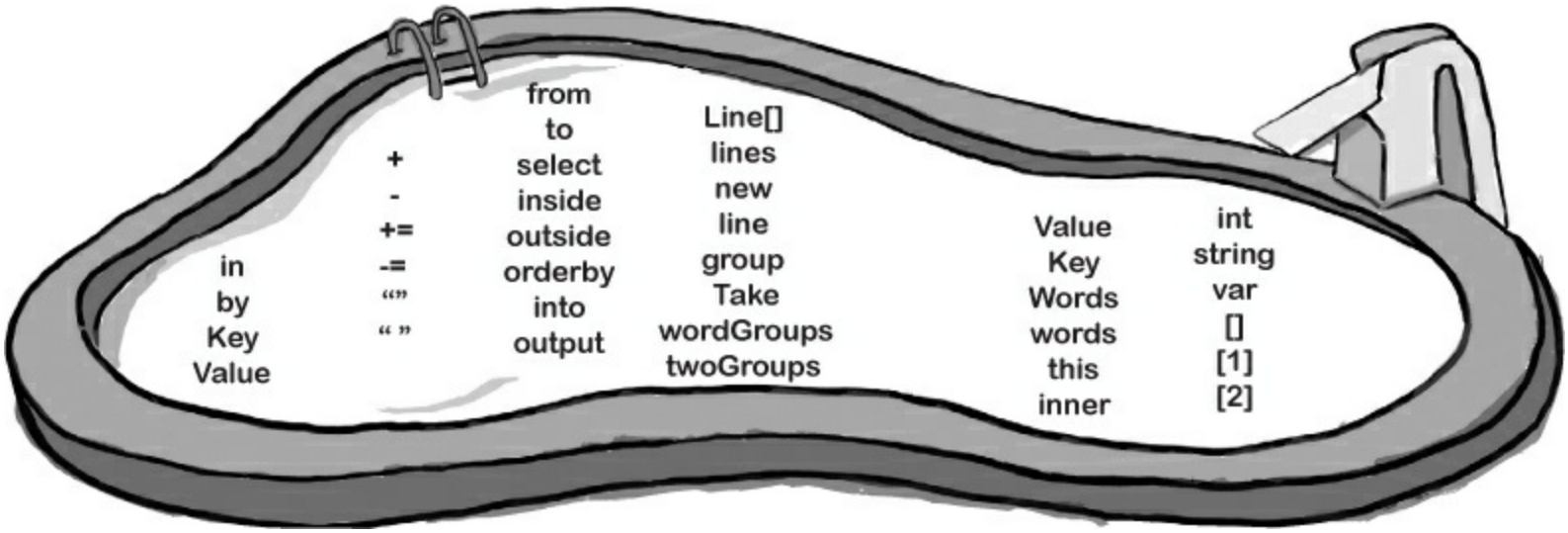
NOTE

Hint: LINQ sorts strings in alphabetical order.

```
Line[] lines = {
  new Line( new string[] { "eating", "carrots,",
          "but", "enjoy", "Horses" } , 1),
  new Line( new string[] { "zebras?", "hay",
         "Cows", "bridge.", "bolted" } , 2),
  new Line( new string[] { "fork", "dogs!",
         "Engine", "and" }, 3 ) ,
  new Line( new string[] { "love", "they",
         "apples.", "eating" }, 2 ) ,
  new Line( new string[] { "whistled.", "Bump" }, 1 ) };

var ______ =
    from ______ in ______
    ______ line by line.______
    into wordGroups
    orderby _________.______
    select _________;
____ __________ = words.______(2);
foreach (var group in twoGroups)
{
   int i = 0;
   foreach (______ inner in ______) {
      i++;
```

```
        if (i == ______.Key) {
           var poem =
              ______ word in ______.______
              _________ word descending
              ______ word + ____;
           foreach (var word in ______)
               Console.Write(word);
         }
      }
   }
```

Note: Each snippet from the pool can be used more than once!



POOL PUZZLE SOLUTION

```
class Line {
    public string[] Words;
    public int Value;
    public Line(string[] Words, int Value) {
        this.Words = Words; this.Value = Value;
    }
}

Line[] lines = {
  new Line( new string[] { "eating", "carrots,", "but", "enjoy", "Horses" } , 1),
  new Line( new string[] { "zebras?", "hay", "Cows", "bridge.", "bolted" } , 2),
  new Line( new string[] { "fork", "dogs!", "Engine", "and" }, 3 ) ,
  new Line( new string[] { "love", "they", "apples.", "eating" }, 2 ) ,
  new Line( new string[] { "whistled.", "Bump" }, 1 )
};

var words =
    from line in lines
    group line by line.Value
    into wordGroups
    orderby wordGroups.Key
    select wordGroups;

var twoGroups = words.Take(2);

foreach (var group in twoGroups)
{
    int i = 0;
    foreach (var inner in group) {
        i++;
        if (i == group.Key) {
            var poem =
                from word in inner.Words
                orderby word descending
                select word + " ";
            foreach (var word in poem)
                Console.Write(word);
        }
    }
}
```

This first LINQ query divides the Line objects in the lines[] array into groups, grouped by their Value, in ascending order of the Value key.

The first two groups are the lines with Values 1 and 2.

This loop does a LINQ query on the first Line object in the first group and the second Line object in the second group.

Did you figure out that the two phrases "Horses enjoy eating carrots, but" and "they love eating apples" are in descending alphabetical order?

Output: **Horses enjoy eating carrots, but they love eating apples.**

Use join to combine two collections into one sequence

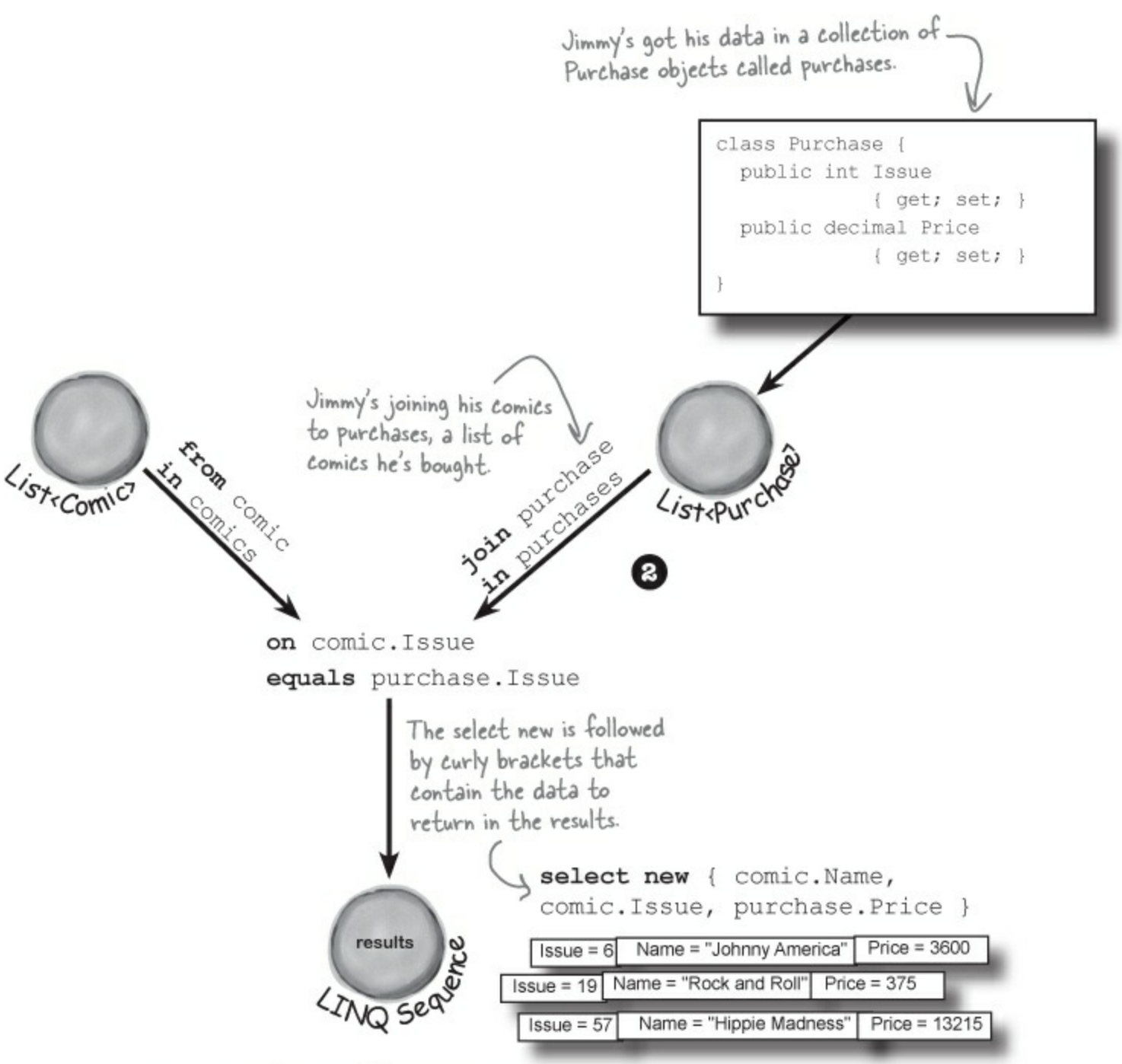
Jimmy's got a whole collection of comics he's purchased, and he wants to compare them with the prices he found from sales on *Greg's List* to see if the prices he's been paying are better or worse. He's been tracking his purchases using a `Purchase` class with two automatic properties, `Issue` and `Price`. And he's got a `List<Purchase>` called `purchases` that's got all the comics he's bought. But now he needs to match up the purchases he's made with the prices he found on *Greg's List*. How's he going to do it?

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.

- ❶ Start off your query with the usual `from` clause. But instead of following it up with the criteria it'll use to determine what goes into the results, you add:

  ```
  join name in collection
  ```

  The `join` clause tells LINQ to loop through both collections to match up pairs with one member from each collection. 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.



Flip to leftover #9 in the appendix to learn more about anonymous types!

- ❷ 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.
- ❸ 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. That's where you use **`select new`** to create a custom set of results using an **anonymous type**.

Jimmy saved a bunch of dough

It looks like Jimmy drives a hard bargain. This query creates a list of `Purchase` classes that contain his purchases, and compares them with the prices he found from recent sales on *Greg's List*.

- **❶ FIRST JIMMY CREATED HIS COLLECTION TO JOIN.**
  Jimmy already had his first collection — he just used his `BuildCatalog()` method from before. So all he had to do was write a `FindPurchases()` method to build his list of `Purchase` classes.

This is a static method in the Purchase class.

```
public static IEnumerable<Purchase> FindPurchases() {
    List<Purchase> purchases = new List<Purchase>() {
        new Purchase() { Issue = 68, Price = 225M },
        new Purchase() { Issue = 19, Price = 375M },
        new Purchase() { Issue = 6, Price = 3600M },
        new Purchase() { Issue = 57, Price = 13215M },
        new Purchase() { Issue = 36, Price = 660M },
    };
    return purchases;
}
```

Jimmy paid $13,215 for issue #57.

- **❷ NOW HE CAN DO THE JOIN!**
  You've seen all the parts of this query already...now here they are, put together in one piece.

```
IEnumerable<Comic> comics = BuildCatalog();
Dictionary<int, decimal> values = GetPrices();
IEnumerable<Purchase> purchases = Purchase.FindPurchases();
var results =
    from comic in comics
    join purchase in purchases
    on comic.Issue equals purchase.Issue
    orderby comic.Issue ascending
    select new { comic.Name, comic.Issue, purchase.Price };
decimal gregsListValue = 0;
decimal totalSpent = 0;
foreach (var result in results) {
    gregsListValue += values[result.Issue];
    totalSpent += result.Price;
    Console.WriteLine("Issue #{0} ({1}) bought for {2:c}",
            result.Issue, result.Name, result.Price);
}
Console.WriteLine("I spent {0:c} on comics worth {1:c}",
            totalSpent, gregsListValue);
```

When Jimmy used a join clause, LINQ compared every item in the comics collection with each item in purchases to see which ones have comic.Issue equal to purchase.Issue.

The select clause creates a result set with Name and Issue from the comic member, and Price from the purchase member.

Jimmy's real happy that he knows LINQ, because it let him see just how hard a bargain he can drive!

Output:

```
Issue #6 (Johnny America vs. the Pinko) bought for $3,600.00
Issue #19 (Rock and Roll (limited edition)) bought for $375.00
Issue #36 (Woman's Work) bought for $660.00
Issue #57 (Hippie Madness (misprinted)) bought for $13,215.00
Issue #68 (Revenge of the New Wave Freak (damaged)) bought for $225.00
I spent $18,075.00 on comics worth $18,525.00
```

BULLET POINTS

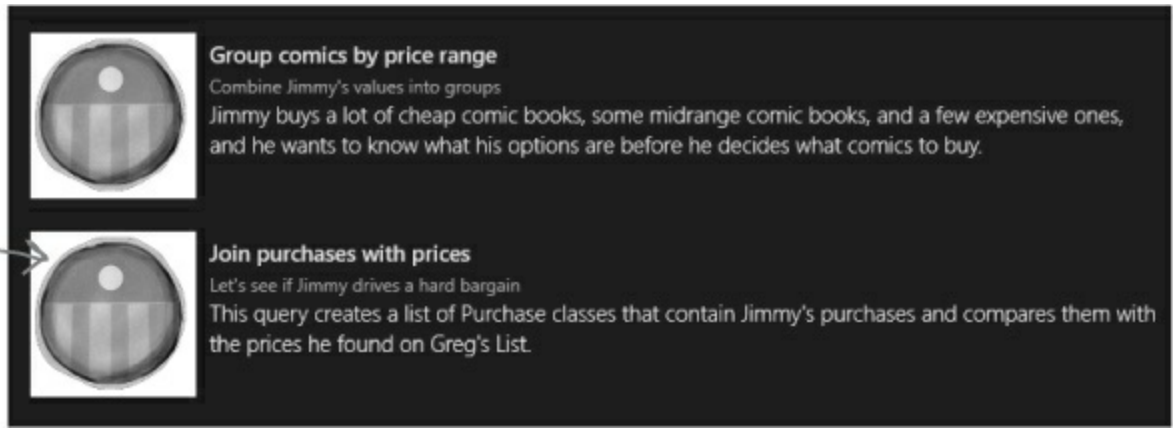
- The **`group`** clause tells LINQ to group the results together — when you use it, LINQ creates a sequence of group sequences.
- Every group contains members that have one member in common, called the group's **`key`**. Use the **`by`** keyword to specify the key for the group. Each group sequence 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.
- You can use **`select new`** to construct custom LINQ query results that include only the items that you want in your result sequence.

EXERCISE

Add the last two LINQ queries to Jimmy's app.

- **❶ ADD THE COMICQUERY OBJECTS TO UPDATEAVAILABLEQUERIES().**
  Update the `AvailableQueries` object initializer to instantiate two new `ComicQuery` objects so they get added to the main page. Here's what the new buttons should look like:

Can you figure out how to make the page title tell Jimmy how much he spent and how much they were worth?



- **❷ ADD METHODS TO EXECUTE THE QUERIES AND UPDATE THE OUTPUT.**
  You'll also need the `Purchase` class and `EvaluatePrice()` method from a few pages ago, and don't forget to add the `PriceRange` enum to the project. Add `EvaluatePrice()` as a static method to the `Purchases` class.
- **❸ ADD THE NEW QUERIES TO UPDATEQUERYRESULTS().**
  Once you add the two new query methods to the `switch` statement in `UpdateQueryResults`, you're in business.

THERE ARE NO DUMB QUESTIONS

Q: **Q: I don't quite get how `join` works.**

A: **A:** `join` works with any two sequences. Let's say you've got a collection of football players called `players` — its items are objects that have a `Name` property, a `Position` property, and a `Number` property. So we could pull out the players whose jerseys have a number bigger than 10 with this query:

```
var results =
 from player in players
 where player.Number > 10
 select player;
```

Let's say we wanted to figure out each player's shirt size, and we've got a `jerseys` collection whose items have a `Number` property and a `Size` property. A `join` would work really well for that:

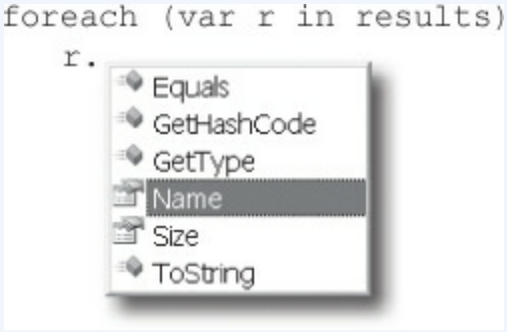
```
var results =
 from player in players
 where player.Number > 10
 join shirt in jerseys
    on player.Number
    equals shirt.Number
 select shirt;
```

Q: **Q: Hold on, that query will just give me a bunch of shirts. What if I want to connect each player to his shirt size, and I don't care about his number at all?**

A: **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 results =
 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. That's because the query would create a different anonymous type with different members.

Q: **Q: Can you rewind a minute and explain what `var` is again?**

A: **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 Jimmy's program:

```
var mostExpensive =
  from comic in comics
  where
     values[comic.Issue]
        > 500
  orderby
     values[comic.Issue]
  descending
  select comic;
```

What if you changed the last line to this:

```
  select new
   { Name = comic.Name,
     IssueNumber = "#" +
comic.Issue };
```

That returns a 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."

EXERCISE SOLUTION

Here's the code you need to add to Jimmy's app to make the last two LINQ queries show up.

```
private void UpdateAvailableQueries() {
    AvailableQueries = new ObservableCollection<ComicQuery> {
        new ComicQuery("LINQ makes queries easy", "A sample query",
            "Let's show Jimmy how flexible LINQ is",
            "Assets/purple_250x250.jpg"),

        new ComicQuery("Expensive comics", "Comics over $500",
            "Comics whose value is over 500 bucks."
                + " Jimmy can use this to figure out which comics are most coveted.",
            "Assets/captain_amazing_250x250.jpg"),

        new ComicQuery("LINQ is versatile 1", "Modify every item returned from the query",
            "This code will add a string onto the end of each string in an array.",
            "Assets/bluegray_250x250.jpg"),

        new ComicQuery("LINQ is versatile 2", "Perform calculations on collections",
            "LINQ provides extension methods for your collections (and anything else"
              + " that implements IEnumerable<T>).",
            "Assets/purple_250x250.jpg"),

        new ComicQuery("LINQ is versatile 3",
            "Store all or part of your results in a new sequence",
            "Sometimes you'll want to keep your results from a LINQ query around.",
            "Assets/bluegray_250x250.jpg"),


        new ComicQuery("Group comics by price range",
            "Combine Jimmy's values into groups",
            "Jimmy buys a lot of cheap comic books, some midrange comic books, and"
                + " a few expensive ones, and he wants to know what his options are"
                + " before he decides what comics to buy.",
            "Assets/captain_amazing_250x250.jpg"),

        new ComicQuery("Join purchases with prices",
            "Let's see if Jimmy drives a hard bargain",
            "This query creates a list of Purchase classes that contain Jimmy's purchases"
                + " and compares them with the prices he found on Greg's List.",
            "Assets/captain_amazing_250x250.jpg"),
    };
}
```

Adding these two ComicQuery objects to the AvailableQueries object initializer causes them to show up on the main page.

EXERCISE SOLUTION

```
public void UpdateQueryResults(ComicQuery query) {
    Title = query.Title;

    switch (query.Title) {
        case "LINQ makes queries easy": LinqMakesQueriesEasy(); break;
        case "Expensive comics": ExpensiveComics(); break;
        case "LINQ is versatile 1": LinqIsVersatile1(); break;
        case "LINQ is versatile 2": LinqIsVersatile2(); break;
        case "LINQ is versatile 3": LinqIsVersatile3(); break;
        case "Group comics by price range":
            CombineJimmysValuesIntoGroups();
            break;
        case "Join purchases with prices":
            JoinPurchasesWithPrices();
            break;
    }
}

enum PriceRange { Cheap, Midrange, Expensive }


class Purchase {
    public int Issue { get; set; }
    public decimal Price { get; set; }

    public static IEnumerable<Purchase> FindPurchases()
    {
        List<Purchase> purchases = new List<Purchase>() {
            new Purchase() { Issue = 68, Price = 225M },
            new Purchase() { Issue = 19, Price = 375M },
            new Purchase() { Issue = 6, Price = 3600M },
            new Purchase() { Issue = 57, Price = 13215M },
            new Purchase() { Issue = 36, Price = 660M },
        };
        return purchases;
    }

    public static PriceRange EvaluatePrice(decimal price)
    {
        if (price < 100M) return PriceRange.Cheap;
        else if (price < 1000M) return PriceRange.Midrange;
        else return PriceRange.Expensive;
    }
}
```

Here are the new cases for the switch statement that call the query methods.

Don't forget the PriceRange enum.

Here's the Purchase class. EvaluatePrice() is now a static method in the class.

```
private void CombineJimmysValuesIntoGroups() {
    Dictionary<int, decimal> values = GetPrices();
    var priceGroups =
        from pair in values
        group pair.Key by Purchase.EvaluatePrice(pair.Value)
            into priceGroup
            orderby priceGroup.Key descending
            select priceGroup;
    foreach (var group in priceGroups) {
        string message = String.Format("I found {0} {1} comics: issues ",
                                       group.Count(), group.Key);
        foreach (var issue in group)
            message += issue.ToString() + " ";
        CurrentQueryResults.Add(
           CreateAnonymousListViewItem(message, "captain_amazing_250x250.jpg"));
    }
}

private void JoinPurchasesWithPrices() {
    IEnumerable<Comic> comics = BuildCatalog();
    Dictionary<int, decimal> values = GetPrices();
    IEnumerable<Purchase> purchases = Purchase.FindPurchases();
    var results =
        from comic in comics
        join purchase in purchases
        on comic.Issue equals purchase.Issue
        orderby comic.Issue ascending
        select new {
            Comic = comic,
            Price = purchase.Price,
            Title = comic.Name,
            Subtitle = "Issue #" + comic.Issue,
            Description = String.Format("Bought for {0:c}", purchase.Price),
            ImagePath = "Assets/captain_amazing_250x250.jpg",
        };

    decimal gregsListValue = 0;
    decimal totalSpent = 0;
    foreach (var result in results) {
        gregsListValue += values[result.Comic.Issue];
        totalSpent += result.Price;
        CurrentQueryResults.Add(result);
    }

    Title = String.Format("I spent {0:c} on comics worth {1:c}",
             totalSpent, gregsListValue);
}
```

Here are the methods with the LINQ queries.

The page title is bound to the Title property in the ComicQueryManager object, so this line will change it to a message telling Jimmy how much he spent and how much the comics are worth.

Use semantic zoom to navigate your data

Use the button in the simulator to put it into pinch/zoom mode. Hold down the mouse button and use the scroll wheel to simulate pinch/zoom.

It's great to give Jimmy an overview of his collection, but let's give him a way to really drill down into the details. There's a very useful control that will let you add an extra dimension to your app's navigation. The **semantic zoom** is a scrollable control that lets your user switch between two different views of a sequence of data: a "zoomed out" view that shows an overview of the data, and a "zoomed in" view that shows more detail for each item in the sequence.

NOTE

You can use pinch to zoom in and out of the semantic zoom control, just like you pinch to zoom your photos on your phone or tablet. You can also use the scroll wheel or click on items.

Semantic zoom allows you to display two different views of the same sequence of data: a zoomedout view that shows many items, and a zoomed-in view that shows more detail.

You can learn more about how semantic zoom fits into your apps here: http://msdn.microsoft.com/en-us/library/windows/apps/hh465319.aspx

Here's the basic XAML pattern for the semantic zoom control. It uses a ListView or GridView control for the zoomed-out view, and another one for the zoomed-in view:

```
<SemanticZoom IsZoomedInViewActive="False" >
    <SemanticZoom.ZoomedOutView>
        <ListView>
            <!-- This section contains a ListView or GridView
                 to show the zoomed out overview view -->
        </ListView>
    </SemanticZoom.ZoomedOutView>
    <SemanticZoom.ZoomedInView>
        <GridView>
            <!-- This ListView or GridView shows
                 the zoomed zoomed in detail view -->
            <GridView.ItemTemplate>
                <DataTemplate>
                    <!-- This ListView or GridView in this section
                         shows the zoomed in detail view -->
                </DataTemplate>
            </GridView.ItemTemplate>
        </GridView>
    </SemanticZoom.ZoomedInView>
</SemanticZoom>
```

The GridView control is a lot like ListView. The main difference is that the ListView control scrolls its items vertically, while the GridView scrolls its items horizontally.

Both the zoomed-in and zoomed-out views contain a ListView or GridView control with a data template that contains controls to make the view work.

We happened to use a ListView for the zoomed-out view and a GridView for the zoomed-in view, but you can use either one for either of the views.

LISTVIEW AND GRIDVIEW IMPLEMENT ISEMANTICZOOMINFORMATION

The SemanticZoom control can only contain controls that implement the ISemanticZoomInformation interface, which has methods that let the SemanticZoom control initiate and complete the view change. Luckily, you don't need to implement this interface yourself. In the example on this page, we used a ListView to show the zoomed-out items, and a GridView to show the zoomed-in items.

Add semantic zoom to Jimmy's app

Jimmy would love to be able to see all of the comics in his collection, and zoom in on individual comics to see details.

DO THIS!

- **❶ ADD A NEW ITEM TO THE MAIN PAGE.**
  Jimmy needs something to click on, so the first thing you'll do is add a new item to return all of the comics in the collection. First, add a method to `ComicQueryManager` to show all of the comics:

```
private void AllComics() {
    foreach (Comic comic in BuildCatalog()) {
        var result = new {
            ImagePath = "Assets/captain_amazing_zoom_250x250.jpg",
            Title = comic.Name,
            Subtitle = "Issue #" + comic.Issue,
            Description = "The Captain versus " + comic.MainVillain,
            Comic = comic,
        };
        CurrentQueryResults.Add(result);
    }
}
```

  Next, add a new case to the `switch` statement in `UpdateQueryResults()`:

```
case "All comics in the collection": AllComics(); break;
```

  And finish it off by adding a new `ComicQuery` to the collection initializer in `UpdateAvailableQueries()`. You'll also need to add the *captain_amazing_zoom_250x250.jpg* image to the *Assets/* folder.

```
new ComicQuery("All comics in the collection",
               "Get all of the comics in the collection",
               "This query returns all of the comics",
               "Assets/captain_amazing_zoom_250x250.jpg"),
```

Download this image file from the Head First C# website.

- **❷ ADD MORE PROPERTIES TO THE COMIC CLASS.**
  Semantic zoom only makes sense if you have details to zoom in on. It'll still be displaying `Comic` objects from the `ComicQueryManager.CurrentQueryResults` collection, so we just need to add those details to the `Comic` class and make sure to bind to those new properties in the zoomed-in view.

```
class Comic {
    public string Name { get; set; }
    public int Issue { get; set; }
    public int Year { get; set; }
    public string CoverPrice { get; set; }
    public string Synopsis { get; set; }
    public string MainVillain { get; set; }
    public string Cover { get; set; }
}
```

- **❸ ADD THE DETAILED COMIC DATA.**
  Modify the `BuildCatalog()` method to add more details about each comic. You'll also need to add images of the covers to your project's Assets. Download them from your class website.

```
public static IEnumerable<Comic> BuildCatalog() {
  return new List<Comic> {
     new Comic { Name = "Johnny America vs. the Pinko", Issue = 6, Year = 1949, CoverPrice = "10 cents",
         MainVillain = "The Pinko", Cover = "Assets/Captain Amazing Issue 6 cover.png",
         Synopsis = "Captain Amazing must save America from Communists as The Pinko and his"
             + " communist henchmen threaten to take over Fort Knox and steal all of the nation's gold."
```

```
    },

        new Comic { Name = "Rock and Roll (limited edition)", Issue = 19, Year = 1957, CoverPrice = "10
cents",
            MainVillain = "Doctor Vortran", Cover = "Assets/Captain Amazing Issue 19 cover.png",
            Synopsis = "Doctor Vortran wreaks havoc with the nation's youth with his radio wave device that"
                + " uses the latest dance craze to send rock and roll fans into a mind-control trance." },

            new Comic { Name = "Woman's Work", Issue = 36, Year = 1968, CoverPrice = "12 cents",
                MainVillain = "Hysterianna", Cover = "Assets/Captain Amazing Issue 36 cover.png",
                Synopsis = "The Captain faces his first female foe, Hysterianna, whose incredible telepathic"
                    + " and telekinetic abilities have allowed her to raise an all-girl army that"
                            + " even the Captain has trouble resisting." },

            new Comic { Name = "Hippie Madness (misprinted)", Issue = 57, Year = 1973, CoverPrice = "20
cents",
                MainVillain = "The Mayor", Cover = "Assets/Captain Amazing Issue 57 cover.png",
                Synopsis = "A zombie apocalypse threatens Objectville when The Mayor rigs the election by"
                    + " introducing his zombification agent into the city's cigarette supply." },

            new Comic { Name = "Revenge of the New Wave Freak (damaged)", Issue = 68, Year = 1984,
                CoverPrice = "75 cents", MainVillain = "The Swindler",
                Cover = "Assets/Captain Amazing Issue 68 cover.png",
                Synopsis = "A tainted batch of eye makeup turns Dr. Alvin Mudd into the Captain's new
nemesis,"
                    + " in The Swindler's first appearance in a Captain Amazing comic." },

            new Comic { Name = "Black Monday", Issue = 74, Year = 1986, CoverPrice = "75 cents",
                MainVillain = "The Mayor", Cover = "Assets/Captain Amazing Issue 74 cover.png",
                Synopsis = "The Mayor returns to throw Objectville into a financial crisis by directing his"
                    + " zombie creation powers to the floor of the Objectville Stock Exchange." },

            new Comic { Name = "Tribal Tattoo Madness", Issue = 83, Year = 1996, CoverPrice = "Two bucks",
                MainVillain = "Mokey Man", Cover = "Assets/Captain Amazing Issue 83 cover.png",
                Synopsis = "Monkey Man escapes from his island prison and wreaks havoc with his circus
sideshow"
                    + " of tattooed henchmen that and their deadly grunge ray." },

            new Comic { Name = "The Death of an Object", Issue = 97, Year = 2013, CoverPrice = "Four bucks",
                MainVillain = "The Swindler", Cover = "Assets/Captain Amazing Issue 97 cover.png",
                Synopsis = "The Swindler's clone army attacks Objectville in a ruse to trap and kill the "
                    + " Captain. Can the scientists of Objectville find a way to bring him back?" },
        };
    }
```

Flip the page to finish the app

- **❹ ADD A NEW BASIC PAGE TO HOLD THE SEMANTIC ZOOM CONTROL.**

Jimmy's happy with the rest of the app, so instead of modifying the existing page, we'll add a whole new page. **Add a new Basic Page called *QueryDetailZoom.xaml*.**

Once it's added, **go back to *MainPage.xaml* and modify its `ItemClick` event handler** in the code-behind to navigate to a `QueryDetailZoom` page if the user clicks on the newly added query:

```
private void ListView_ItemClick(object sender, ItemClickEventArgs e) {
    ComicQuery query = e.ClickedItem as ComicQuery;
    if (query != null) {
        if (query.Title == "All comics in the collection")
            this.Frame.Navigate(typeof(QueryDetailZoom), query);
        else
            this.Frame.Navigate(typeof(QueryDetail), query);
    }
}
```

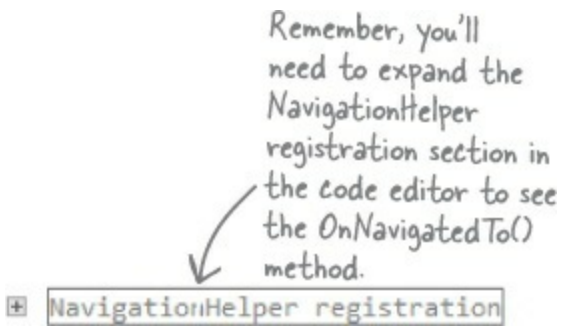
Here's the new ItemClick event handler for MainPage.xaml. It checks the title of the query to see if it should navigate to a QueryDetail or QueryDetailZoom page.

- **❺ ADD A STATIC COMICQUERYMANAGER RESOURCE TO THE NEW PAGE.**
  The new `QueryDetailZoom` page works exactly like the existing `QueryDetail` page. You'll need to add a `ComicQueryManager` to the `<Page.Resources>` section of *QueryDetailZoom.xaml*. You don't need to update the `AppName` resource because the page will set that using C# code:

```
<Page.Resources>
    <local:ComicQueryManager x:Name="comicQueryManager"/>
    <!-- TODO: Delete this line if the key AppName is declared in App.xaml -->
    <x:String x:Key="AppName">My Application</x:String>
</Page.Resources>
```

- **❻ ADD THE CODE-BEHIND FOR THE NEW DETAIL PAGE.**
  And you'll need to add exactly the same `OnNavigatedTo()` method to *QueryDetailZoom.xaml.cs*:

```
protected override void OnNavigatedTo(NavigationEventArgs e) {
    ComicQuery comicQuery = e.Parameter as ComicQuery;
    if (comicQuery != null) {
        comicQueryManager.UpdateQueryResults(comicQuery);
        pageTitle.Text = comicQueryManager.Title;
    }
    navigationHelper.OnNavigatedTo(e);
}
```



- **❼ MODIFY THE XAML FOR THE NEW DETAIL PAGE.**
  There's just one more thing you need to do: build out the XAML for the *QueryDetailZoom.xaml* page so that it contains a semantic zoom control that displays the comic book details. It's the biggest page you've created so far, so we've spread the code across two pages to make it easier for you to understand what's going on.

```
<Grid Grid.Row="1" Margin="120,0"
                  DataContext="{StaticResource ResourceKey=comicQueryManager}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <TextBlock Style="{StaticResource SubheaderTextBlockStyle}" Margin="0,0,0,20"
               Text="Pinch to zoom in or out" />

    <SemanticZoom IsZoomedInViewActive="False" Grid.Row="1">

        <SemanticZoom.ZoomedOutView>
          <ListView ItemsSource="{Binding CurrentQueryResults}"
                    Margin="0,0,20,0" SelectionMode="None">
            <ListView.ItemTemplate>
              <DataTemplate>
                <Grid Height="110" Width="480" Margin="10">
                  <Grid.ColumnDefinitions>
                     <ColumnDefinition Width="Auto"/>
                     <ColumnDefinition Width="*"/>
                  </Grid.ColumnDefinitions>
                  <Image Source="{Binding ImagePath}" Stretch="UniformToFill"/>
                  <StackPanel Grid.Column="1" VerticalAlignment="Top" Margin="10,0,0,0">
                    <TextBlock Text="{Binding Title}" TextWrapping="NoWrap"
                               Style="{StaticResource TitleTextBlockStyle}"/>
                    <TextBlock Text="{Binding Subtitle}" TextWrapping="NoWrap"
                               Style="{StaticResource CaptionTextBlockStyle}"/>
                    <TextBlock Text="{Binding Description}" MaxHeight="60"
                               Style="{StaticResource BodyTextBlockStyle}"/>
                  </StackPanel>
                </Grid>
              </DataTemplate>
            </ListView.ItemTemplate>
          </ListView>
        </SemanticZoom.ZoomedOutView>
```

We're putting the SemanticZoom in a grid row to make sure that its ListView and GridView controls can scroll.

Our Semantic Zoom's zoomed-out view is a ListView that's very similar to like the one on the QueryDetail page. It uses a DataTemplate. We added another textblock bound to the Description property.

Flip the page for the rest of the comic detail page XAML. ⟶

```
<SemanticZoom.ZoomedInView>
  <GridView ItemsSource="{Binding CurrentQueryResults}"
            Margin="0,0,20,0" SelectionMode="None" x:Name="detailGridView">
    <GridView.ItemTemplate>
      <DataTemplate>
        <Grid Height="780" Width="600" Margin="10">
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition/>
          </Grid.ColumnDefinitions>

          <Image Source="{Binding Comic.Cover}" Margin="0,0,20,0"
                 Stretch="UniformToFill" Width="326" Height="500"
                 VerticalAlignment="Top"/>

          <StackPanel Grid.Column="1">

            <TextBlock Text="Name"
                       Style="{StaticResource CaptionTextBlockStyle}" />
            <TextBlock Text="{Binding Comic.Name}"
                       Style="{StaticResource SubtitleTextBlockStyle}" />
            <TextBlock Text="Issue"
                       Style="{StaticResource CaptionTextBlockStyle}" Margin="0,10,0,0" />
            <TextBlock Text="{Binding Comic.Issue}"
                       Style="{StaticResource SubtitleTextBlockStyle}" />
            <TextBlock Text="Year"
                       Style="{StaticResource CaptionTextBlockStyle}" Margin="0,10,0,0" />
            <TextBlock Text="{Binding Comic.Year}"
                       Style="{StaticResource SubtitleTextBlockStyle}" />
            <TextBlock Text="Cover Price"
                       Style="{StaticResource CaptionTextBlockStyle}" Margin="0,10,0,0" />
            <TextBlock Text="{Binding Comic.CoverPrice}"
                       Style="{StaticResource SubtitleTextBlockStyle}" />
            <TextBlock Text="Main Villain"
                       Style="{StaticResource CaptionTextBlockStyle}" Margin="0,10,0,0" />
            <TextBlock Text="{Binding Comic.MainVillain}"
                       Style="{StaticResource SubtitleTextBlockStyle}" />
            <TextBlock Text="Synopsis"
                       Style="{StaticResource CaptionTextBlockStyle}" Margin="0,10,0,0" />
            <TextBlock Text="{Binding Comic.Synopsis}"
                       Style="{StaticResource SubtitleTextBlockStyle}" />
          </StackPanel>
        </Grid>
      </DataTemplate>
    </GridView.ItemTemplate>
  </GridView>
</SemanticZoom.ZoomedInView>
</SemanticZoom>
</Grid>
```

Here's the rest of the XAML, which defines the Semantic Zoom's zoomed-in view. It displays the rest of the details in a set of TextBlock controls that are bound to properties on the Comic object.

The zoomed-in view is based on a GridView. Its data template is a grid that contains an Image control for the cover and a StackPanel of TextBlock controls for the other properties.

This is just another DataTemplate. It's longer than the ones you've seen before, but it's all familiar stuff. It has Grid with an image on the left and a StackPanel with a set of TextBlocks on the right.

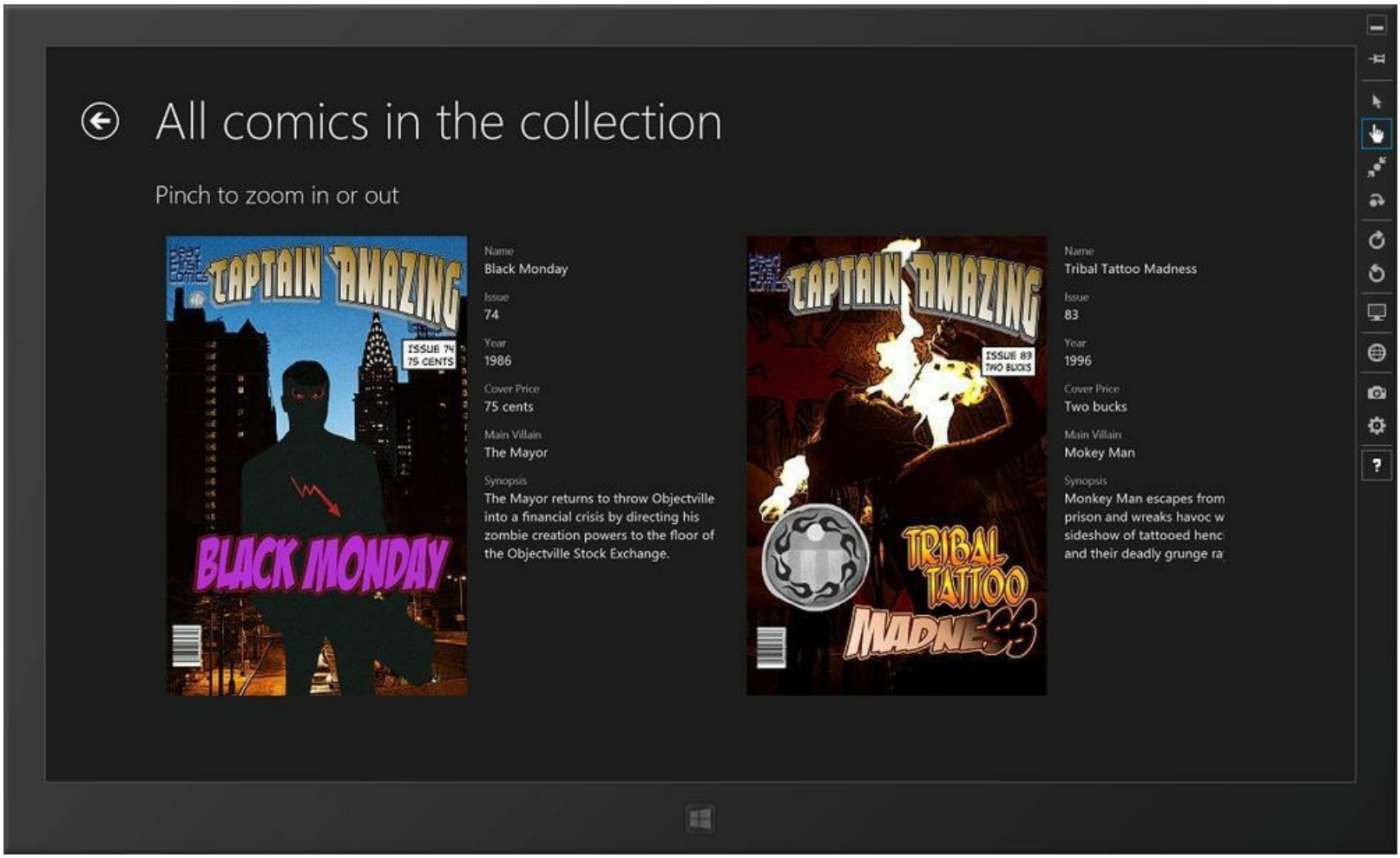
EDIT QUERIES WITH LINQPAD

There's a great learning tool for exploring and using LINQ. It's called LINQPad, and it's available for free from Joe Albahari (one of our superstar "Head First C#" technical reviewers who kept a lot of bugs out of this book). You can download it here:

http://www.linqpad.net/

You made Jimmy's day

Thanks to the new app you built, Jimmy has his collection totally organized. Nice work!

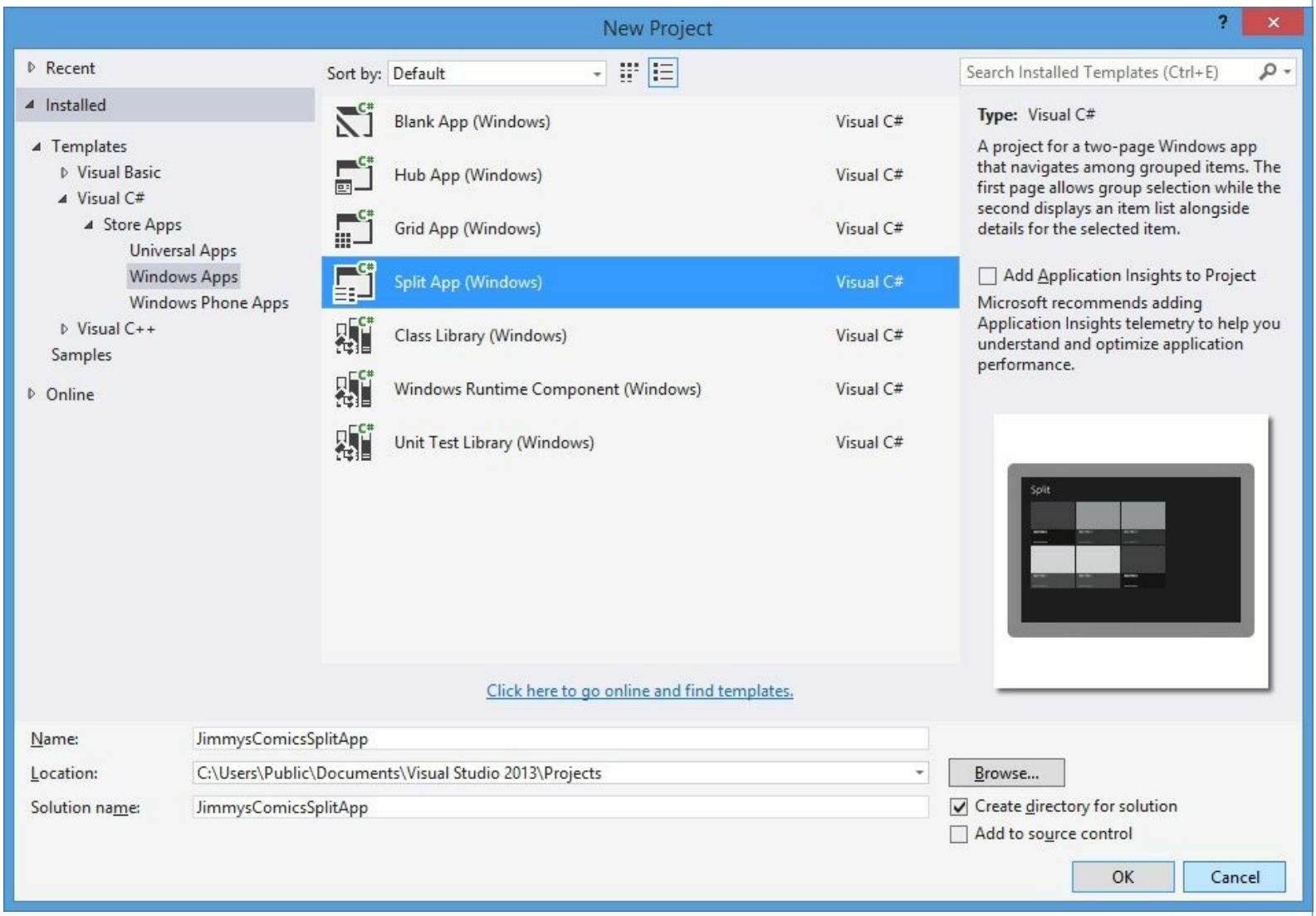


The IDE's Split App template helps you build apps for navigating data

There's an easier way to build two-page apps that navigate between overview and detail pages to display grouped items. When you create a new project using the **Split App template**, the IDE automatically creates a project that lets the user navigate between an overview items page and a split page with the details. We can explore the Split App template by adapting the app that you built for Jimmy to use it.

DO THIS!

- **❶ Create a new Split App project and run it.**
  The Split App project template contains a class that generates sample data, which means that you can actually run it as soon as it's created.
  Create a new **Split App (XAML) project and name it JimmysComicsSplitApp** so the namespaces match the code on the next few pages.

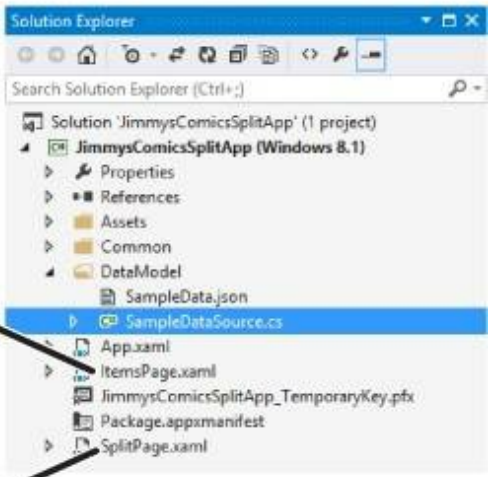


Change the app name to "Jimmy's Comics" using the `AppName` resource. In projects created with the Split App project template, **the `AppName` resource is in the *App.xaml* file**:

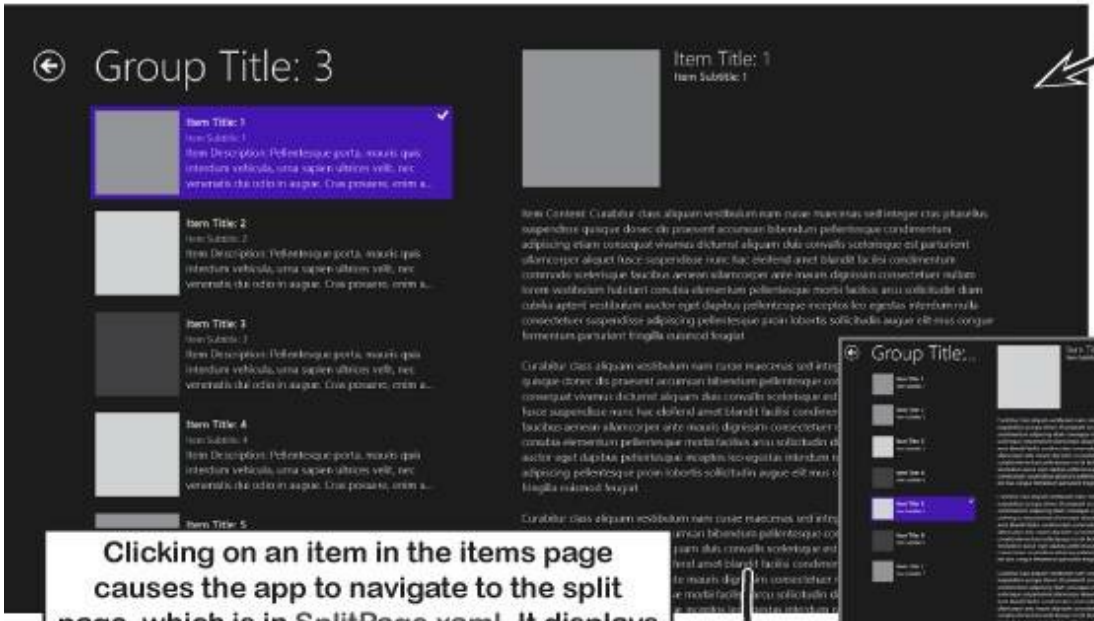
```
<x:String x:Key="AppName">Jimmy's Comics</x:String>
```

Now run the app. A Split App has two pages. The first page is the items page, which shows you groups of items that you can drill down into:

The XAML for the items page is in the ItemsPage.xaml file. The XAML for the page is set up to display groups using a GridView control. Have a look at the XAML for the page—it should look very familiar.



Click on one of the items to navigate to the split page:

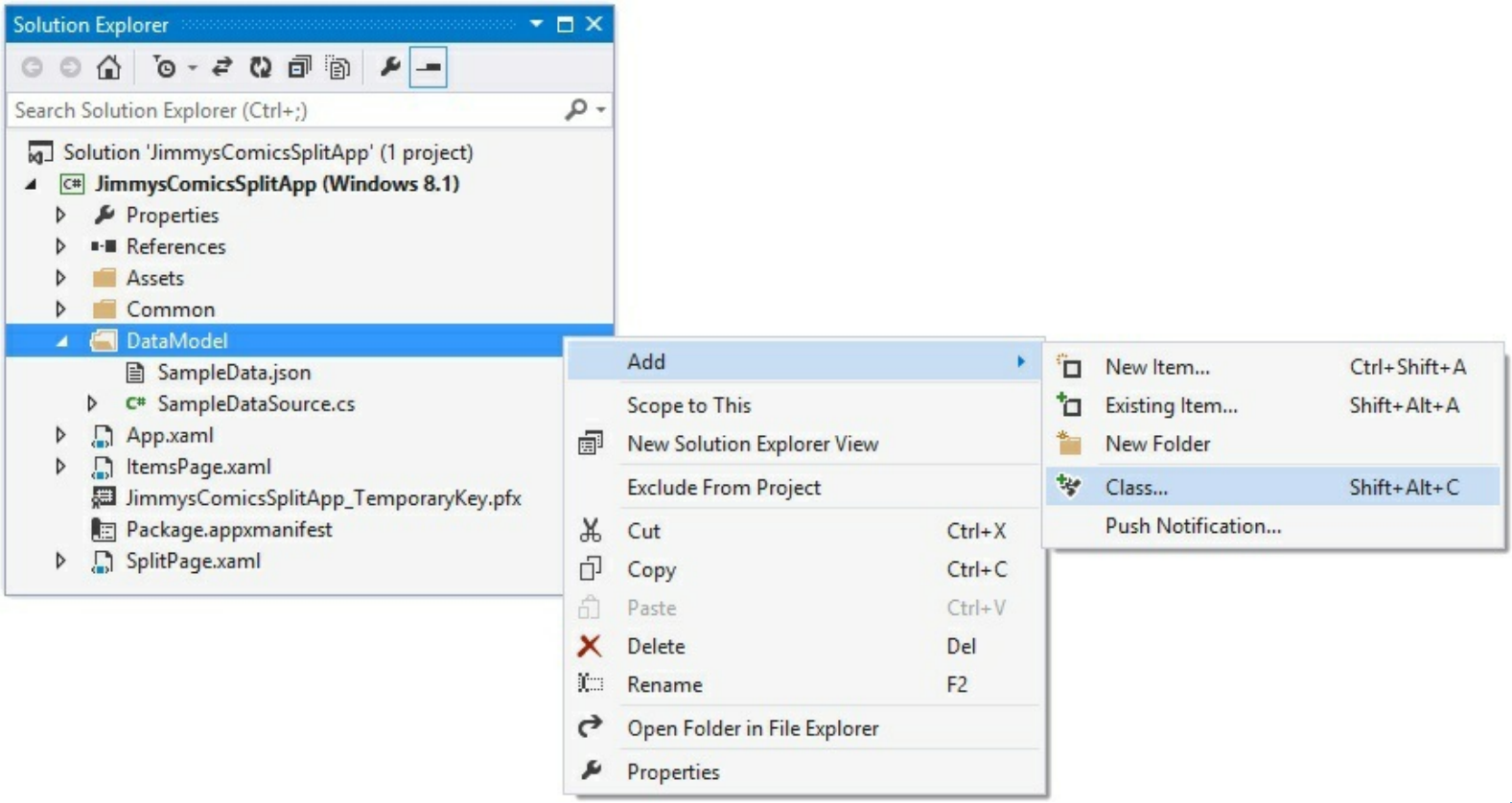


Clicking on an item in the items page causes the app to navigate to the split page, which is in SplitPage.xaml. It displays the contents of the group on the left using a ListView control. Details for the selected item are displayed on the right. This XAML should look familiar, too—the app that you built for Jimmy uses almost identical XAML.

The XAML for the split page automatically adjusts itself when the screen is rotated to portrait. You can try this yourself: run the app in the simulator and use the and buttons to rotate the simulated device.

The large block of text is a single TextBlock, which you'll replace in step 6 with the XAML code from your app that displays comic details.

- **❷ Add data classes to the DataModel folder.**
  Right-click on the *DataModel* folder in the Solution Explorer and choose Add→Class... to add a new class to the folder.

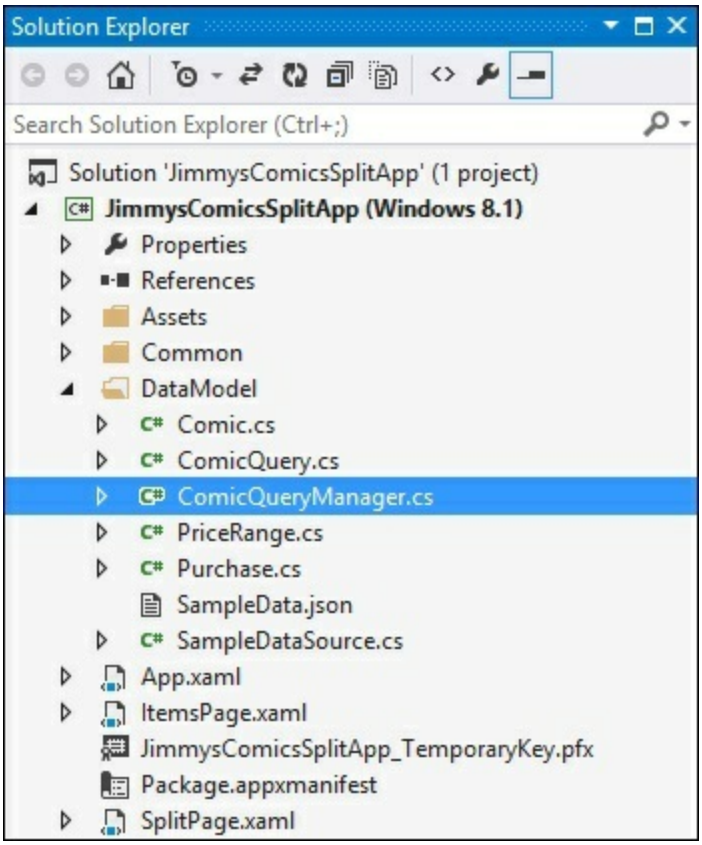


Create the `ComicQueryManager` class. When you create a class inside a folder, the IDE automatically generates it using a namespace that includes the folder name:

```
namespace JimmysComicsSplitApp.DataModel
{
    class ComicQueryManager
    {

    }
}
```

Copy the contents of the `ComicQueryManager` class from your working Jimmy's Comics app and paste them into the newly generated *ComicQueryManager.cs* file in the *DataModel* folder. Make sure you keep the `JimmysComicsSplitApp.DataModel` namespace. (Don't forget the `using` statement!)

Next, repeat the same steps to **create the `Comic`, `ComicQuery`, and `Purchase` classes, as well as the `PriceRange` enum**. They should all be created inside the *DataModel* folder, which means they should all be in the same `JimmysComicsSplitApp.DataModel` namespace. Here's what your Solution Explorer should look like after the files are all added:



There was already a file inside the *DataModel* folder called *SampleDataSource.cs*, which contains code to generate all of the sample data that you saw when you ran the app.

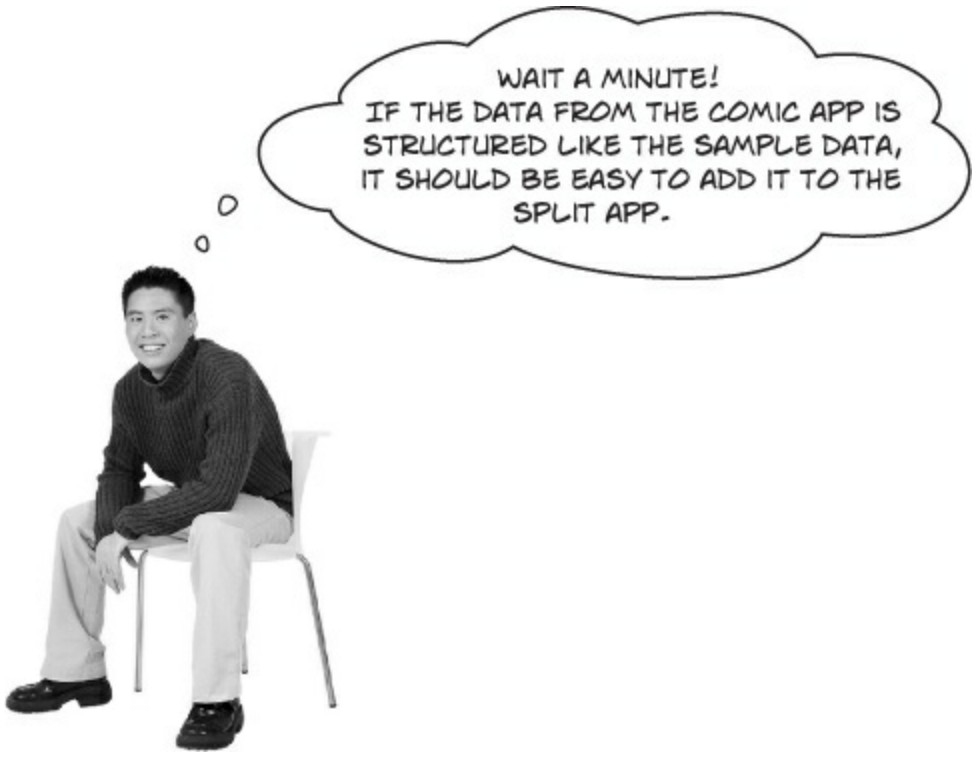
Open it up — it actually works a lot like the data classes app you built for Jimmy. The file contains several classes, including a `SampleDataGroup` class (which represents higher-level groups, similar to your `ComicQuery` class) and a `SampleDataItem` class (which represents individual items, like your `Comic` class). The actual sample data is created in the constructor of the `SampleDataSource` class at the very bottom of the file.

The code-behind for the items page creates a new instance of the `SampleDataSource` class, and uses it to populate a dictionary called `DefaultViewModel`.

NOTE

You'll learn more about what a ViewModel is and how to build one in Chapter 16.

You've already seen how objects can be serialized to binary objects and XML. The Split App uses yet another format called JSON. The SampleDataSource class has a method called GetSampleDataAsync() that reads the JSON data from a file called SampleData.json. Double-click on the data file and look at the JSON data in the IDE. Can you figure out how JSON works just from looking at that file? You can learn more about JSON here: http://json.org/



That's right. The Split App template is built to make it easy for you to add your data.

All you need to do to get your data into the Split App is make a few tweaks to the code-behind in the items page and split page, so that's what we'll do next. We'll also modify the split page so that it uses the same XAML to display the comic book cover and information in the detail page.

- **❸ Modify the code-behind in ItemsPage.xaml.cs.**
  Open *ItemsPage.xaml.cs* and use Edit→Find and Replace to search for "TODO:" in the code. Have a look at the comments — the template is letting you know that this is where you replace the sample data. Comment out the next two lines that set the Items value in the `DefaultViewModel` dictionary, and **replace them with your own code** that reads the `AvailableQueries` property from a new `ComicQueryManager` object:

Comment out these lines.

```
// TODO: Create an appropriate data model for your problem domain to replace the sample data
// var sampleDataGroups = await SampleDataSource.GetGroupsAsync();
// this.DefaultViewModel["Items"] = sampleDataGroups;
this.DefaultViewModel["Items"] = new DataModel.ComicQueryManager().AvailableQueries;
```

Add this line

  You'll also need to comment out the code in the `ItemView_ItemClick()` event handler method, which attempts to cast the item that was clicked on to the `SampleDataGroup` type (it's passed into the event arguments as `e.ClickedItem`). The `AvailableQueries` property returns a collection of ComicQuery objects, so here's the new `ItemClick()` event handler:

```
void ItemView_ItemClick(object sender, ItemClickEventArgs e) {
    // Navigate to the appropriate destination page, configuring the new page
    // by passing required information as a navigation parameter
    //var groupId = ((SampleDataGroup)e.ClickedItem).UniqueId;
    //this.Frame.Navigate(typeof(SplitPage), groupId);
    DataModel.ComicQuery query = e.ClickedItem as DataModel.ComicQuery;
    if (query != null)
        this.Frame.Navigate(typeof(SplitPage), query);
}
```

You created ComicQuery and the other classes in the DataModel folder, so they live in the DataModel namespace.

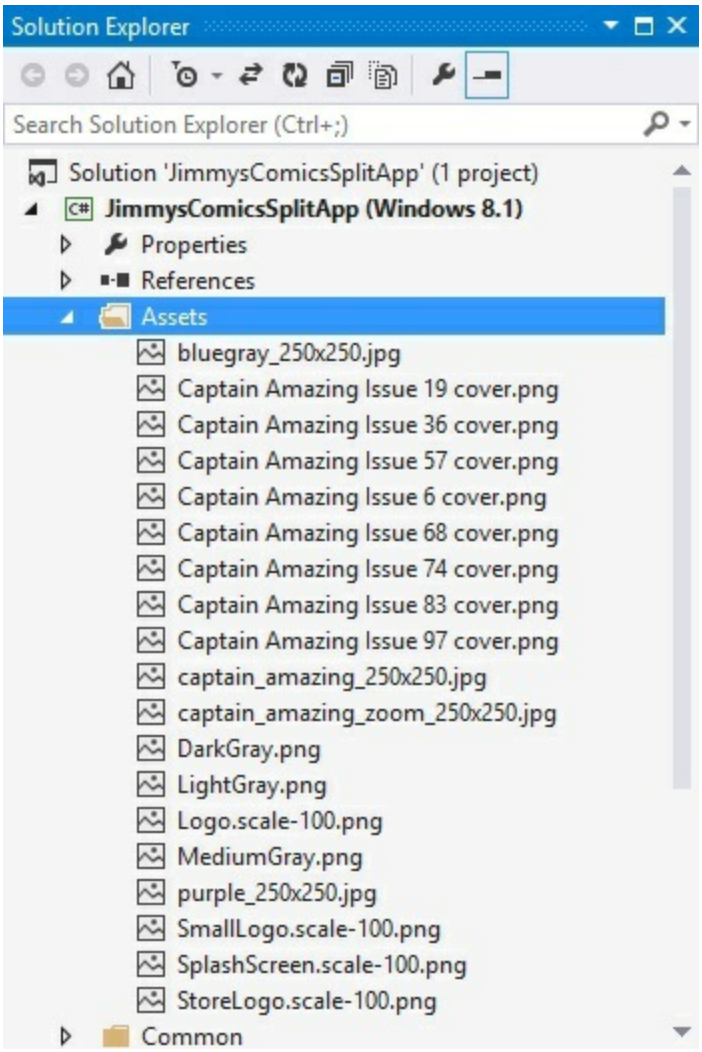
- **❹ Modify the code-behind in SplitPage.xaml.cs.**
  The split page has a method called `navigateionHelper_LoadState()` that uses the navigation parameter to the Group and Items values in the `DefaultViewModel` dictionary. Modify it with code to set the group and items for the page using your comic book data model:

```
private async void navigationHelper_LoadState(object sender, LoadStateEventArgs e)
{
    //var group = await SampleDataSource.GetGroupAsync((String)e.NavigationParameter);
    //this.DefaultViewModel["Group"] = group;
    //this.DefaultViewModel["Items"] = group.Items;DataModel.ComicQueryManager
    DataModel.ComicQueryManager comicQueryManager = new DataModel.ComicQueryManager();
    DataModel.ComicQuery query = e.NavigationParameter as DataModel.ComicQuery;
    comicQueryManager.UpdateQueryResults(query);
    this.DefaultViewModel["Group"] = query;
    this.DefaultViewModel["Items"] = comicQueryManager.CurrentQueryResults;
```

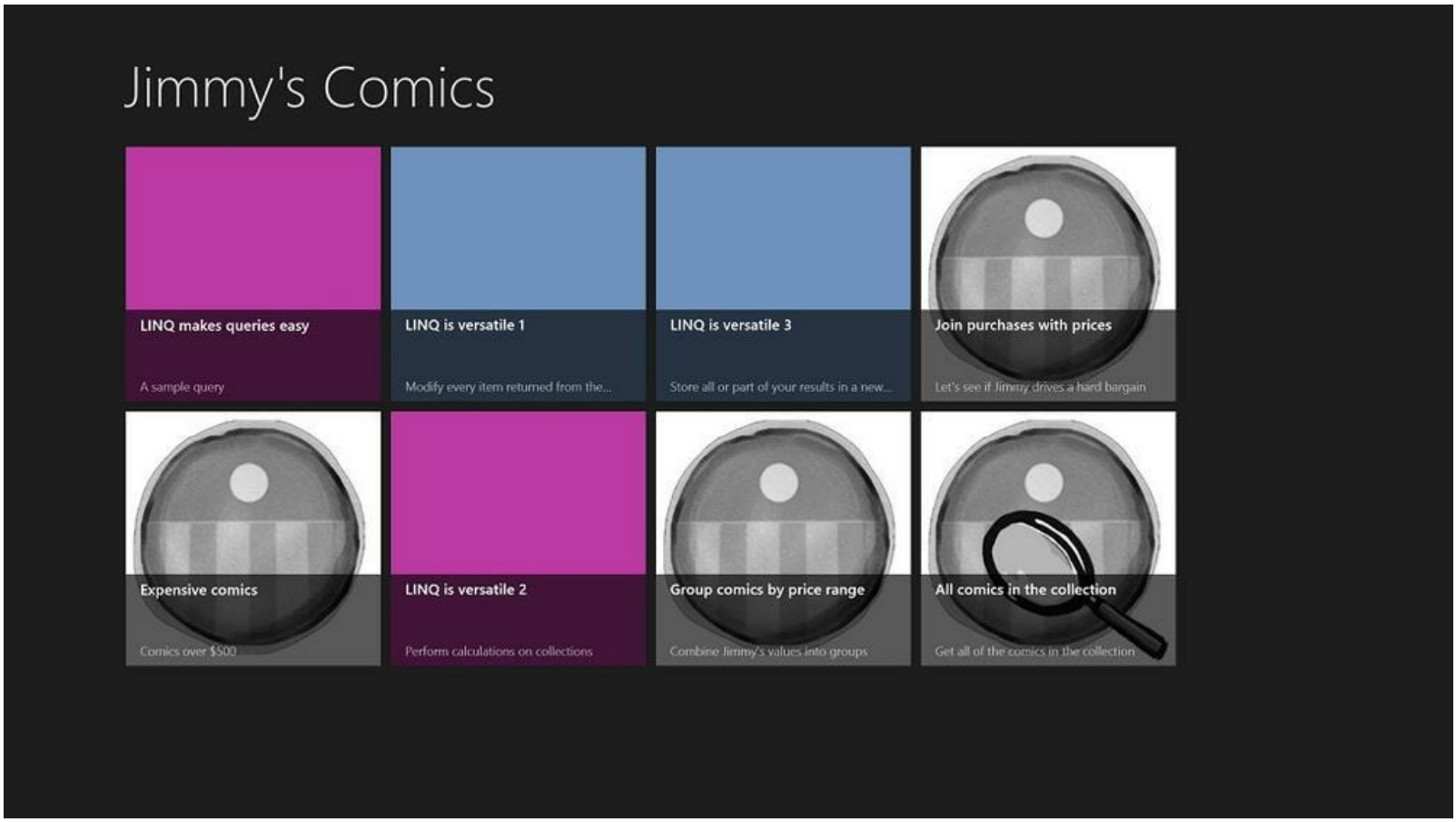
  There's one other thing you need to do. The split page overrides the `SaveState` method, which allows it to remember which item was clicked on. The code that's generated casts the selected item to `SampleDataItem`, so comment out all of the code in the method to avoid casting exceptions.

```
protected override void SaveState(Dictionary<String, Object> pageState) {
    // Comment out all of the code in this method
}
```

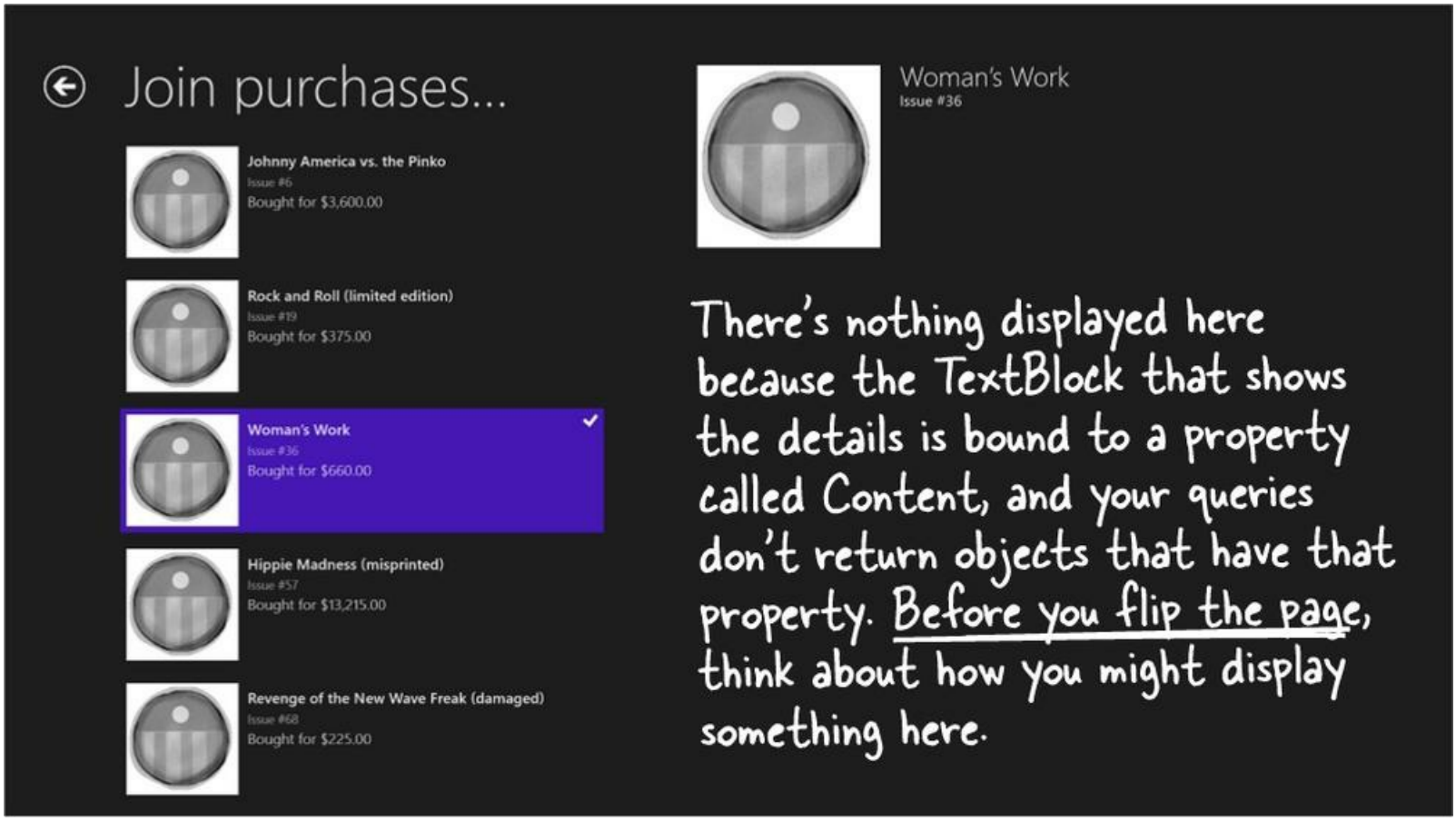
- **❺ Add the image files to the Assets folder.**
  You'll need all of the image files from the app you built for Jimmy. **Right-click on the *Assets* folder and choose Add→Existing Item...** to bring up the Add Existing Item window. Navigate to the folder that has the code for the app you created earlier in the chapter and use Control-click to multiselect all of the files except for *Logo.scale-100.png*, *SmallLogo.scale-100.png*, *SplashScreen.scale-100.png*, and *StoreLogo.scale-100.png*. Click Add to add all of the files to your project's *Assets* folder.



Your app now runs! The items page displays the available queries from the ComicQueryManager object...



...and the split page shows you the query results and lets you drill down into the details of each item.



- **❻ Modify SplitPage.xaml to show the comic book details.**
  The XAML in *SplitPage.xaml* uses templates to display the items on the lefthand side of the split,

but it just uses straightforward, out-of-the-box XAML with data binding to show the details for the selected item on the righthand side. It includes this TextBlock that's bound to a `Content` property:

```
<TextBlock Grid.Row="2" Grid.ColumnSpan="2" Margin="0,20,0,0"
      Text="{Binding Content}" Style="{StaticResource BodyTextBlockStyle}"/>
```

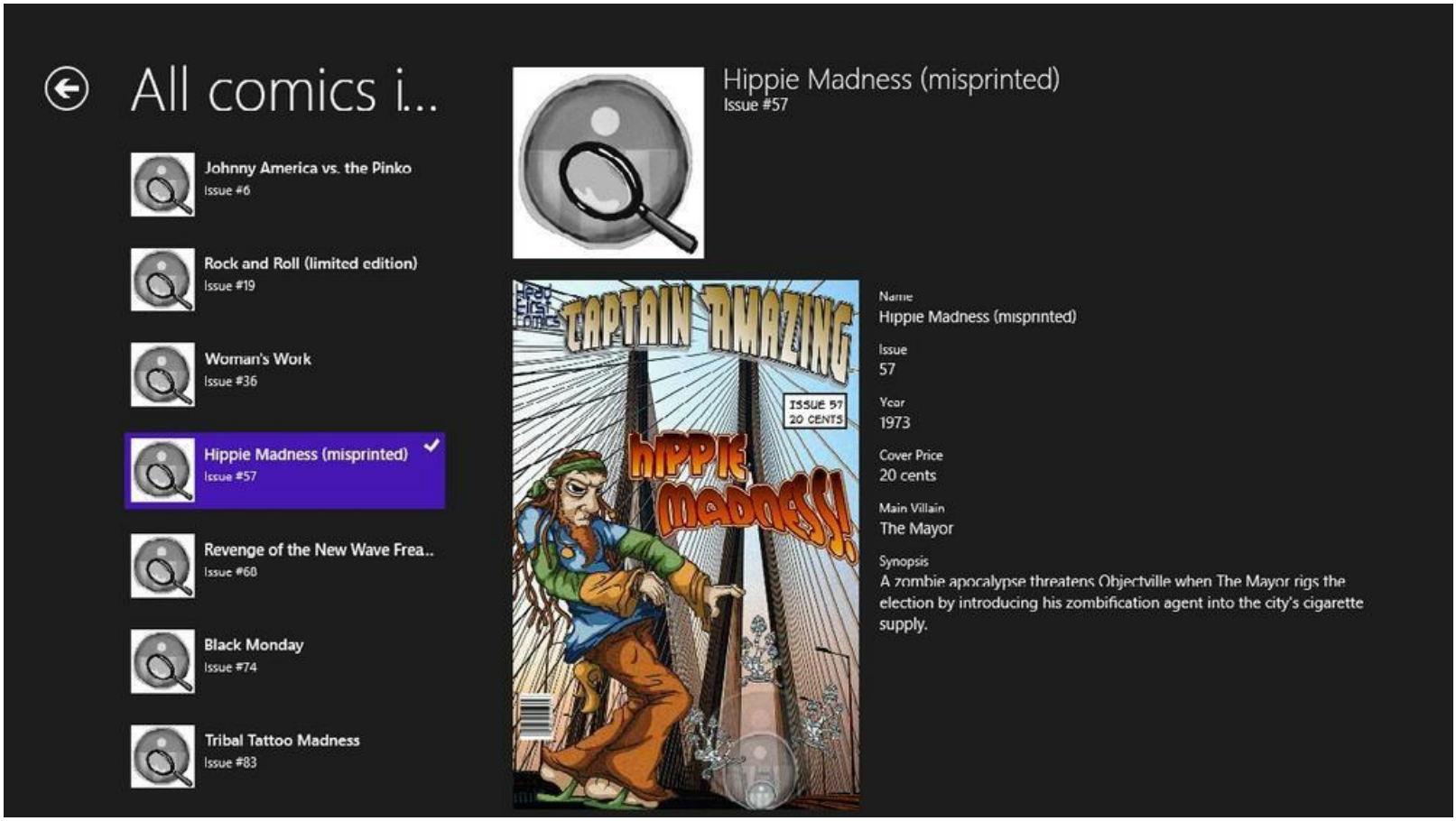
We'll reuse these properties so the comic information appears in the same place on the page.

The sample data in *SampleDataSource.cs* exposes a `Content` property that contains a large block of text. But we want our app to display information about comics in Jimmy's collection. Luckily, we already have a block of XAML that displays information nicely when it's bound to a `Comic` object. **Find the content TextBlock and replace it with the comic detail XAML**. Make sure you add the `Grid.Row`, `Grid.ColumnSpan`, and `Margin` properties to the inner `<Grid>`:

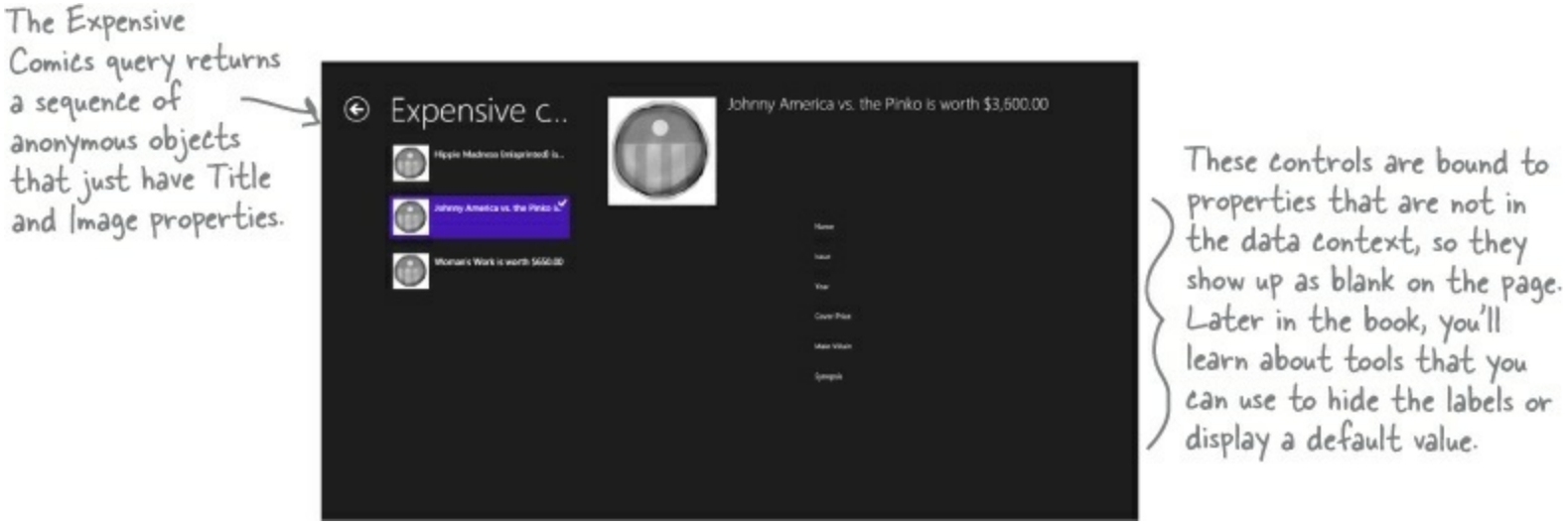
```
<Grid Grid.Row="2" Grid.ColumnSpan="2" Margin="0,20,0,0"
      Height="780"  HorizontalAlignment="Left">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Image Source="{Binding Comic.Cover}" Margin="0,0,20,0"
           Stretch="UniformToFill" Width="326" Height="500"
           VerticalAlignment="Top"/>

    <StackPanel Grid.Column="1">
        <TextBlock Text="Name"
            Style="{StaticResource CaptionTextBlockStyle}" />
        <TextBlock Text="{Binding Comic.Name}"
            Style="{StaticResource SubtitleTextBlockStyle}" />
        <TextBlock Text="Issue"
            Style="{StaticResource CaptionTextBlockStyle}" Margin="0,10,0,0" />
        <TextBlock Text="{Binding Comic.Issue}"
            Style="{StaticResource SubtitleTextBlockStyle}" />
        <TextBlock Text="Year"
            Style="{StaticResource CaptionTextBlockStyle}" Margin="0,10,0,0" />
        <TextBlock Text="{Binding Comic.Year}"
            Style="{StaticResource SubtitleTextBlockStyle}" />
        <TextBlock Text="Cover Price"
            Style="{StaticResource CaptionTextBlockStyle}" Margin="0,10,0,0" />
        <TextBlock Text="{Binding Comic.CoverPrice}"
            Style="{StaticResource SubtitleTextBlockStyle}" />
        <TextBlock Text="Main Villain"
            Style="{StaticResource CaptionTextBlockStyle}" Margin="0,10,0,0" />
        <TextBlock Text="{Binding Comic.MainVillain}"
            Style="{StaticResource SubtitleTextBlockStyle}" />
        <TextBlock Text="Synopsis"
            Style="{StaticResource CaptionTextBlockStyle}" Margin="0,10,0,0" />
        <TextBlock Text="{Binding Comic.Synopsis}"
            Style="{StaticResource SubtitleTextBlockStyle}" />

    </StackPanel>
</Grid>
```

Since you're replacing a TextBlock with this Grid, by keeping these properties from the TextBlock you make sure that the Grid ends up in the same place on the page as the TextBlock.

Now your Split App lets you drill down into the results of any query that returns comics, displaying the details of the selected comic on the split page.



Some of the queires don't return a `Comic`, so any field bound to the field will be empty. In Chapter 16, you'll learn about value converters, which you can use to hide these fields or display a default value.



You can also add pages to your project using Items Page and Split Page templates using the same Add New Item feature in the IDE that you use to add the Basic Page. And there's another valuable template called Grid App that has three levels of navigation. You can learn more about the Grid App and Split App templates here: http://msdn.microsoft.com/en-us/library/windows/apps/hh768232.aspx

Chapter 15. Events and Delegates: What your code does when you're not looking

Your objects are starting to think for themselves.

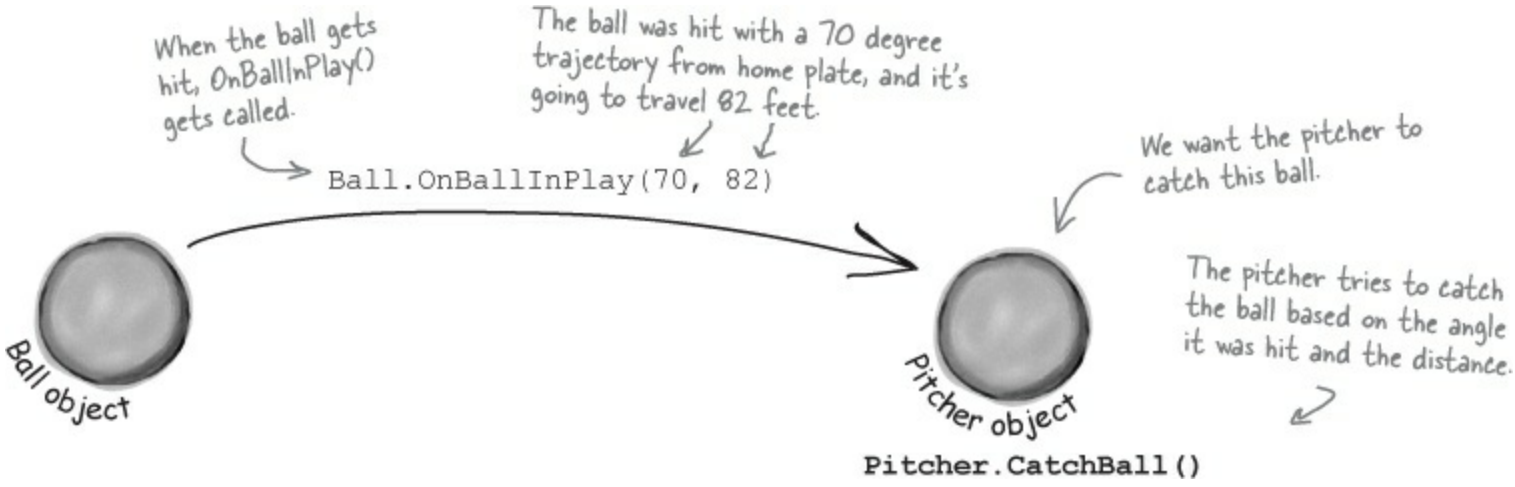
You can't always control what your objects are doing. Sometimes things...happen. And when they do, you want your objects to be smart enough to **respond to anything** that pops up. And that's what events are all about. One object *publishes* an event, other objects *subscribe*, and everyone works together to keep things moving. Which is great, until you want your object to take control over who can listen. That's when **callbacks** will come in handy.

Ever wish your objects could think for themselves?

Suppose you're writing a baseball simulator. You're going to model a game, sell the software to the Yankees (they've got deep pockets, right?), and make a million bucks. You create your `Ball`, `Pitcher`, `Umpire`, and `Fan` objects, and a whole lot more. You even write code so that the `Pitcher` object can catch a ball.

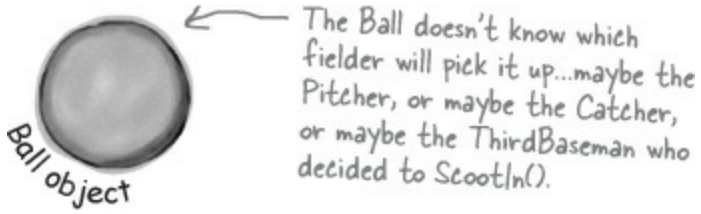
Now you just need to connect everything together. You add an `OnBallInPlay()` method to `Ball`, and now you want your `Pitcher` object to respond with its event handler method. Once the methods are written, you just need to tie the separate methods together:

> **NOTE**
>
> That's a commonly used way of naming methods — we'll talk more about it later.



But how does an object KNOW to respond?

Here's the problem. You really want your `Ball` object to only worry about getting hit, and your `Pitcher` object to only worry about catching balls that come its way. In other words, you really don't want the `Ball` telling the `Pitcher`, "I'm coming to you."



You want an object to worry about itself, not other objects. You're separating the concerns of each object.

NOTE

This doesn't mean that objects can't interact. It just means that a Ball shouldn't determine who fields it. That's not the Ball's job.

When an EVENT occurs...objects listen

What you need to do when the ball is hit is to use an **event**. An event is simply ***something that's happened*** in your program. Then, other objects can respond to that event — like our `Pitcher` object.

> **NOTE**
>
> event, noun.
>
> a **thing** that happens, especially something of importance. *The solar eclipse was an amazing **event** to behold.*

Even better, more than one object can listen for events. So the `Pitcher` could listen for a ball-being-hit event, as well as a `Catcher`, a `ThirdBaseman`, an `Umpire`, even a `Fan`. And each object can respond to the event differently.

So what we want is a `Ball` object that can **raise an event**. Then, we want to have other objects to **subscribe to that particular type of event** — that just means to listen for it, and to get notified when that event occurs.

Want to DO SOMETHING with an event? You need an event handler

Once your object "hears" about an event, you can set up some code to run. That code is called an **event handler**. An event handler gets information about the event, and runs every time that event occurs.

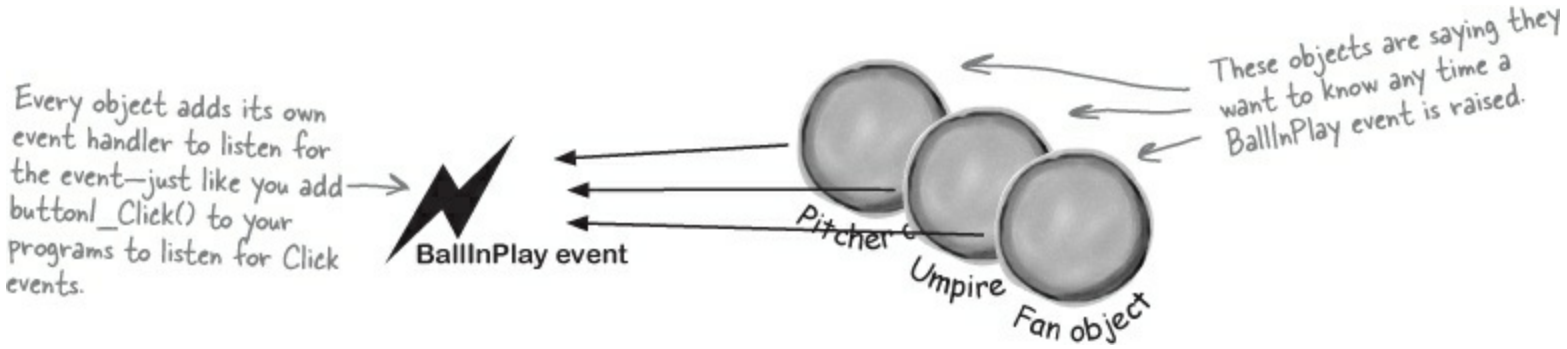
> **NOTE**
>
> We've been doing this all along. Every time you click a button, an event is raised, and your code responds to that event.

Remember, all this happens ***without your intervention*** at runtime. So you write code to raise an event, and then you write code to handle those events, and fire up your application. Then, whenever an event is raised, your handler kicks into action...*without you doing anything*. And, best of all, your objects have separate concerns. They're worrying about themselves, not other objects.

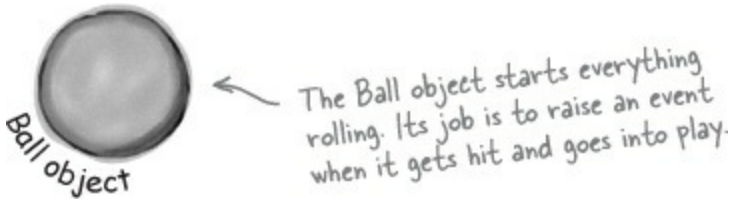
One object raises its event, others listen for it...

An event has a **publisher** and can have multiple **subscribers**. Let's take a look at how events, event handlers, and subscriptions work in C#:

- **❶ First, other objects subscribe to the event.**
  Before the `Ball` can raise its `BallInPlay` event, other objects need to subscribe to it. That's their way of saying that any time a `BallInPlay` event occurs, we want to know about it.

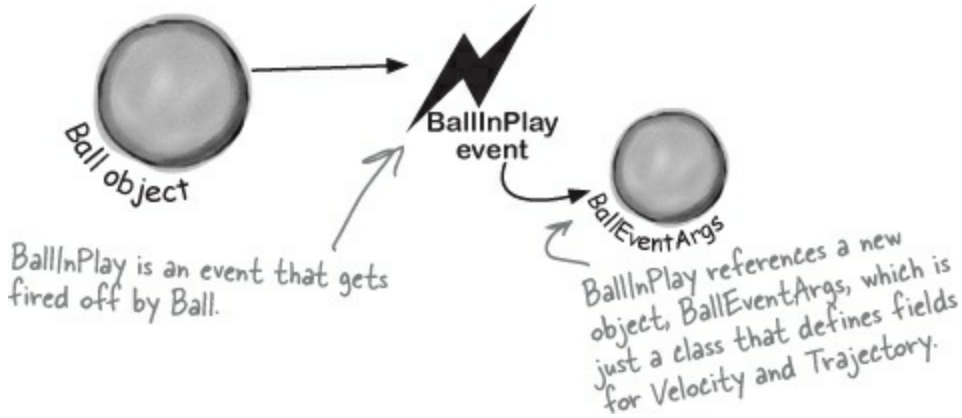


- **❷ Something triggers an event.**
  The ball gets hit. It's time for the `Ball` object to raise a new event.



> **NOTE**
>
> Sometimes we'll talk about raising an event, or firing it, or invoking it — they're all the same thing. People just use different names for it.

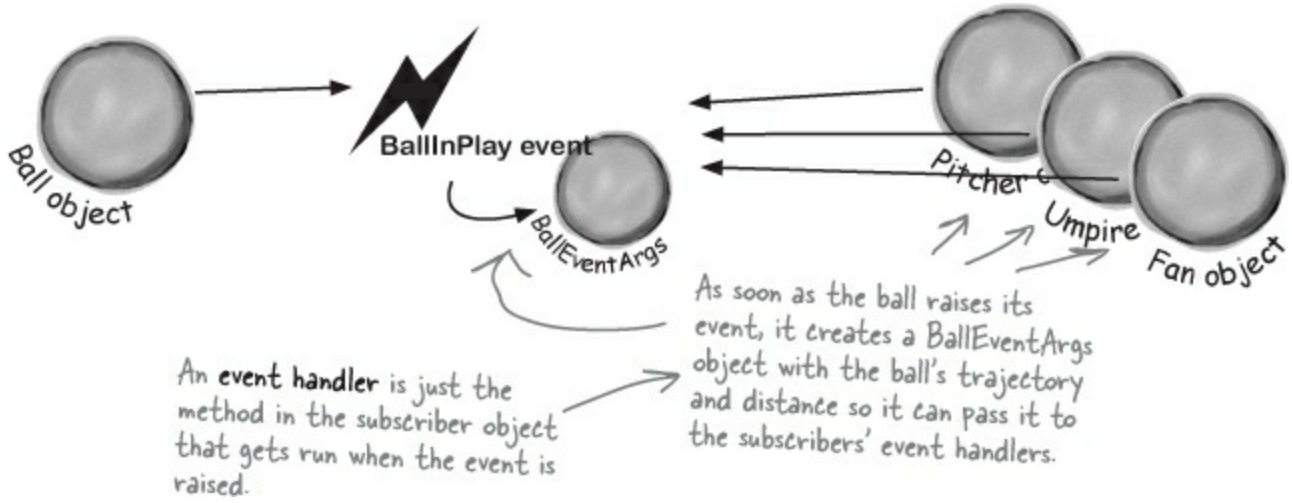
- **❸ The ball raises an event.**
  A new event gets raised (we'll talk about exactly how that works in just a minute). That event also has some arguments, like the velocity of the ball, as well as its trajectory. Those arguments are attached to the event as an instance of an `EventArgs` object, and then the event is sent off, available to anyone listening for it.



Then, the other objects handle the event

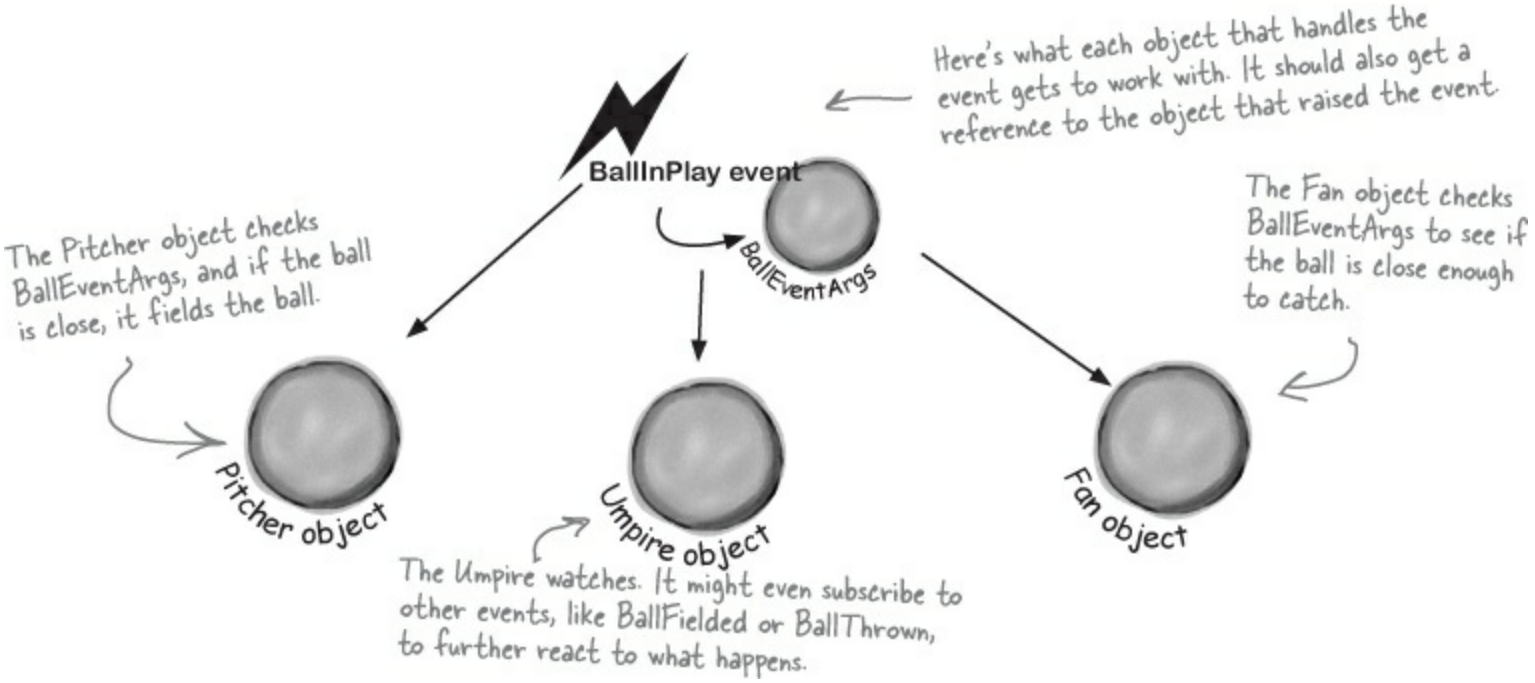
Once an event is raised, all the objects subscribed to that event get notification, and can do something:

- **❹ Subscribers get notification.**
  Since the `Pitcher`, `Umpire`, and `Fan` object subscribed to the `Ball` object's `BallInPlay` event, they all get notified — all of their event handler methods get called one after another.



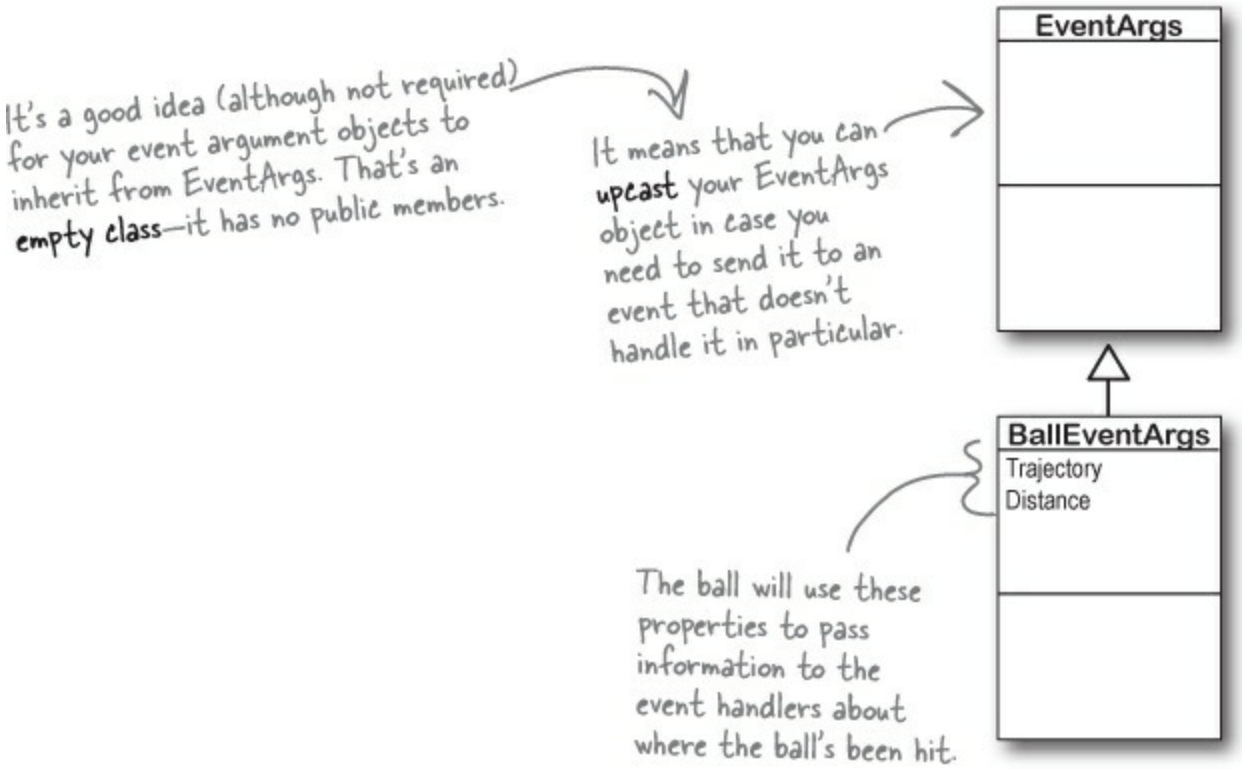
- **❺ Each object handles the event.**
  Now, `Pitcher`, `Umpire`, and `Fan` can all handle the `BallInPlay` event in their own way. But they don't all run at the same time — their event handlers get called one after another, with a reference to a `BallEventArgs` object as its parameter.

> **NOTE**
>
> Events are handled on a first-come, first-served basis — the object that subscribes first gets notified first.



Connecting the dots

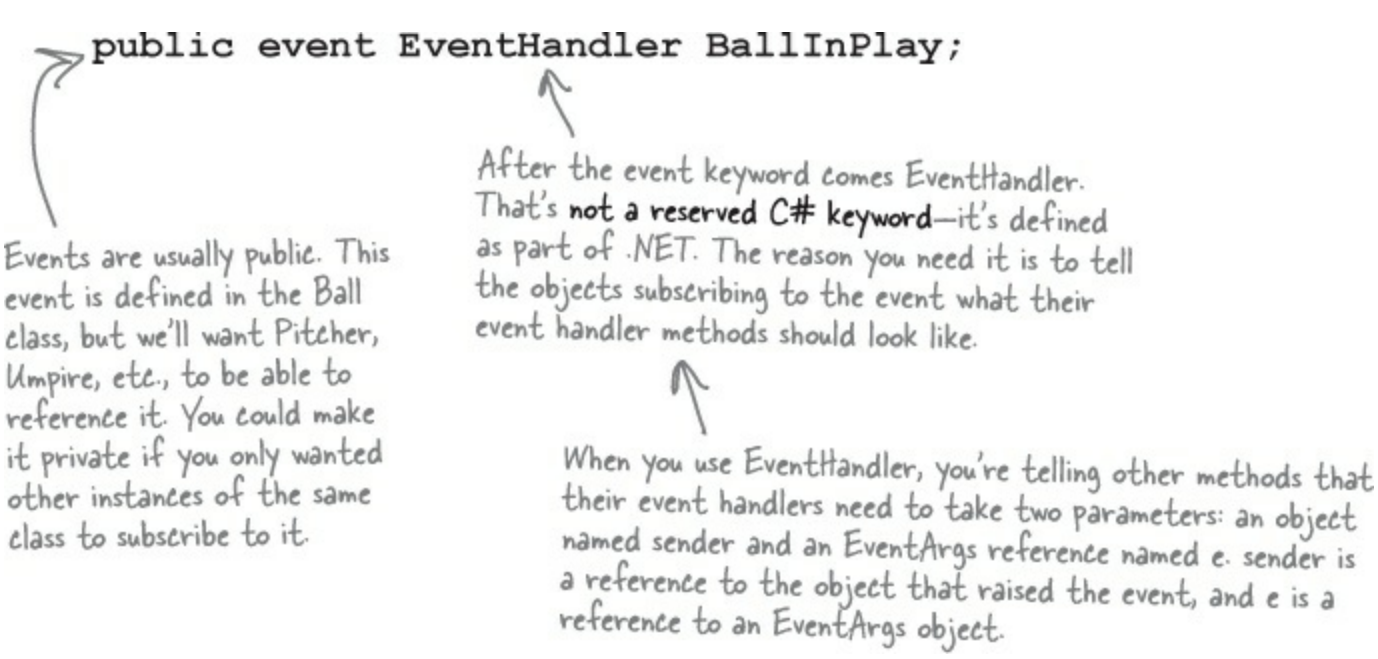
Now that you've got a handle on what's going on, let's take a closer look at how the pieces fit together. Luckily, there are only a few moving parts.



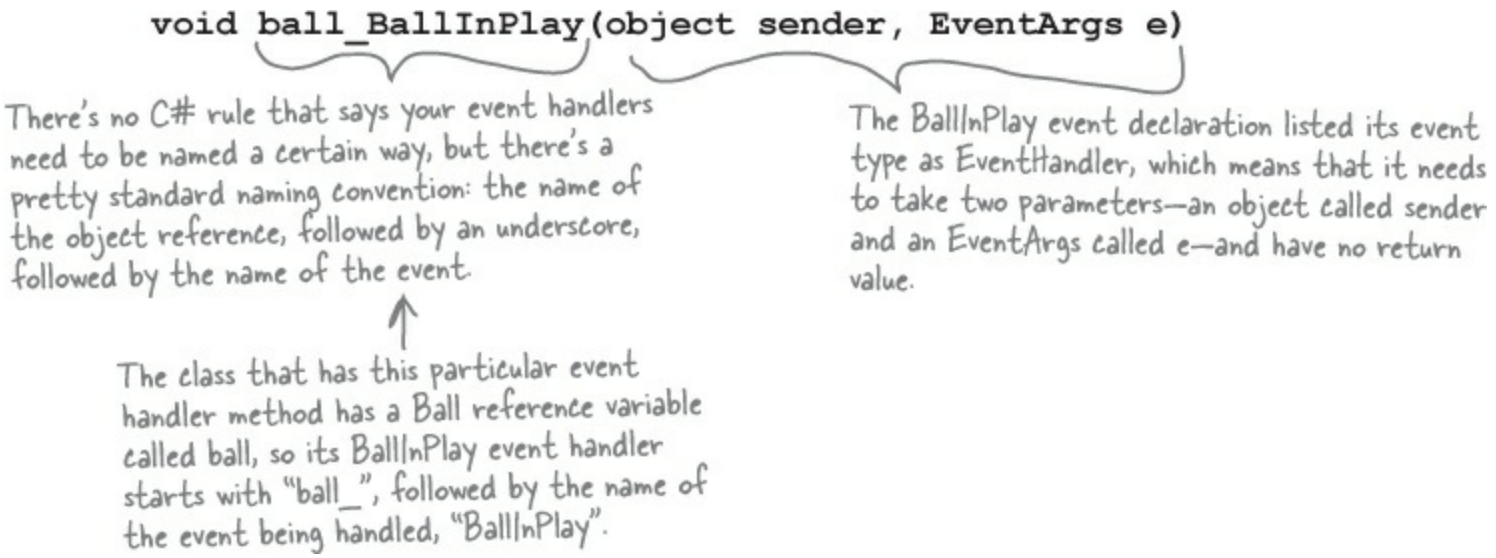
- **❶ We need an object for the event arguments.**
  Remember, our `BallInPlay` event has a few arguments that it carries along. So we need a very simple object for those arguments. .NET has a standard class for it called **`EventArgs`**, but that class **has no members.** Its sole purpose is to allow your event arguments object to be passed to the event handlers that use it. Here's the class declaration:

```
class BallEventArgs : EventArgs
```

- **❷ Next we'll need to define the event in the class that'll raise it.**
  The ball class will have a line with the **`event` keyword** — this is how it informs other objects about the event, so they can subscribe to it. This line can be anywhere in the class — it's usually near the property declarations. But as long as it's in the `Ball` class, other objects can subscribe to a ball's event. You saw the event keyword when you fired PropertyChanged events. Here's the BallInPlay `event` declaration:



- **❸ The subscribing classes need event handler methods.**
  Every object that has to subscribe to the `Ball`'s `BallInPlay` event needs to have an event handler. You already know how event handlers work — every time you added a method to handle a button's `Click` event or a `NumericUpDown`'s `ValueChanged` event, the IDE added an **event handler method** to your class. The `Ball`'s `BallInPlay` event is no different, and an event handler for it should look pretty familiar:



- **❹ Each individual object subscribes to the event.**
  Once we've got the event handler set up, the various `Pitcher`, `Umpire`, `ThirdBaseman`, and `Fan` objects need to hook up their own event handlers. Each one of them will have its own specific `ball_BallInPlay` method that responds differently to the event. So if there's a `Ball` object reference variable or field called `ball`, then the += operator will hook up the event handler:

```
ball.BallInPlay += new EventHandler(ball_BallInPlay);
```

This tells C# to hook the event handler up to the BallInPlay event of whatever object the ball reference is pointing to.

The += operator tells C# to subscribe an event handler to an event.

This part specifies which event handler method to subscribe to the event.

The event handler method's signature (its parameters and return value) has to **match** the one defined by EventHandler or the program **won't compile**.

Turn the page; there's a little more....

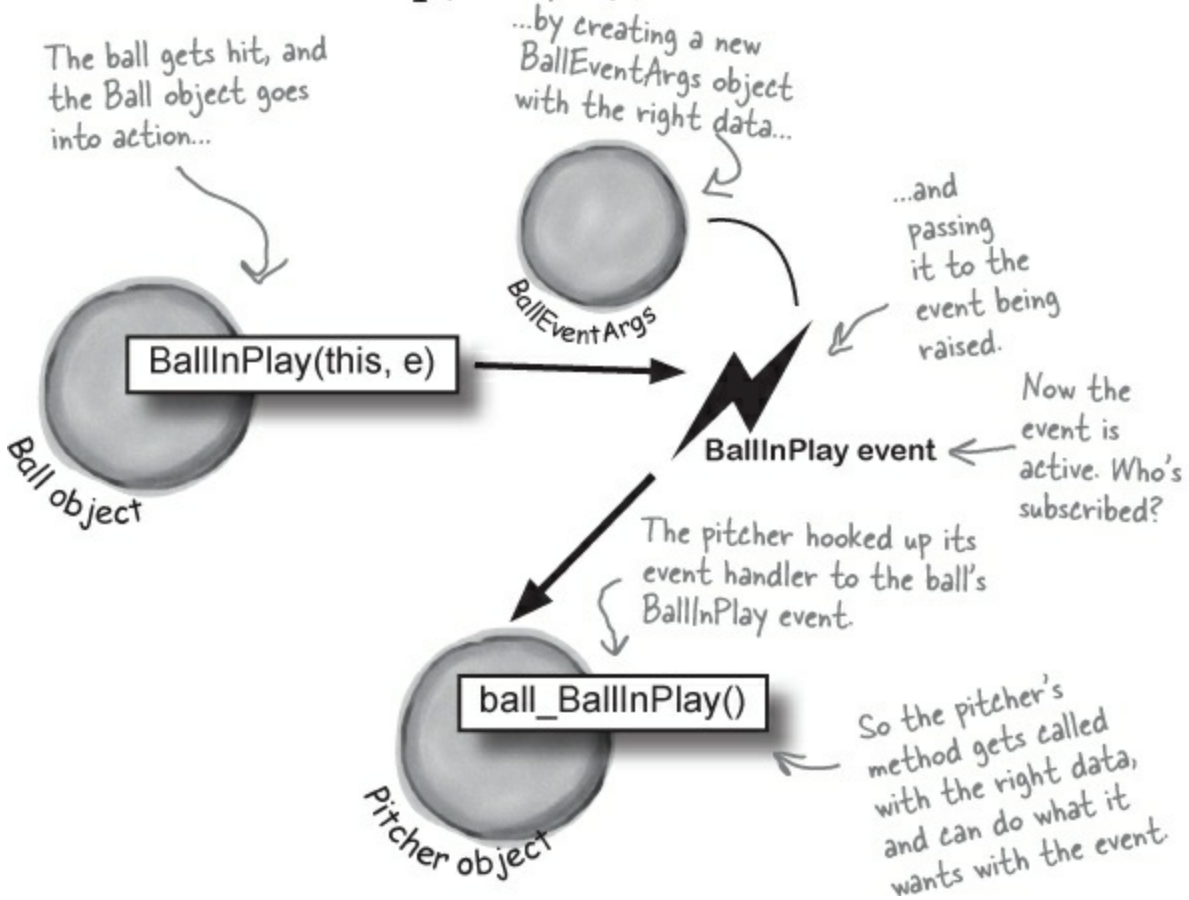
- **❺ A `Ball` object raises its event to notify subscribers that it's in play.**

  Now that the events are all set up, the `Ball` can **raise its event** in response to something else that happens in the simulator. Raising an event is easy — it just calls the `BallInPlay` event.

```
EventHandler ballInPlay = BallInPlay;
if (ballInPlay != null)
    ballInPlay(this, e);
```

BallInPlay is copied to a variable, ballInPlay, which is null-checked and used to raise the event.

e is a new BallEventArgs object.



WATCH IT!

If you raise an event with no handlers, it'll throw an exception.

If no other objects have added their event handlers to an event, it'll be null. So always check to make sure your event handler isn't equal to null before you raise it. If you don't, it'll throw a `NullReferenceException`*. That's also why you should* ***copy the event to a variable*** *before you check to see if it's null — in extremely rare cases, the event can become null between the null check and the time that it's called.*

Use a standard name when you add a method to raise an event

Take a minute and go to the code for any page in a Windows Store app, and type the keyword **`override`** any place you'd declare a method. As soon as you press space, an IntelliSense window pops up:

There are a huge number of events that a XAML `Page` object can raise, and every one of them has its own method that raises it. The page's `OnDoubleTapped()` raises the `DoubleTappedEvent` event (which it inherits from a superclass called `UIElement`), and that's the whole reason it's there. So the `Ball` event will follow the same convention: we'll make sure it **has a method called `OnBallInPlay`** that takes a `BallEventArgs` object as a parameter. The baseball simulator will call that method any time it needs the ball to raise its `BallInPlay` event — so when the simulator detects that the bat hit the ball, it'll create a new instance of `BallEventArgs` with the ball's trajectory and distance and pass it to `OnBallInPlay()`.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Why do I need to include the word `EventHandler` when I declare an event? I thought the event handler was what the other objects used to subscribe to the events.**

A: **A:** That's true — when you need to subscribe to an event, you write a method called an event handler. But did you notice how we used `EventHandler` in the event declaration (step #2) ***and*** in the line to subscribe the event handler to it (step #4)? What `EventHandler` does is define the **signature** of the event — it tells the objects subscribing to the event exactly how they need to define their event handler methods. Specifically, it says that if you want to subscribe a method to this event, it needs to take two parameters (an `object` and an `EventArgs` reference) and have a `void` return value.

Q: **Q: What happens if I try to use a method that doesn't match the ones that are defined by `EventHandler`?**

A: **A:** Then your program won't compile. The compiler will make sure that you don't ever accidentally subscribe an incompatible event handler method to an event. That's why the standard event handler, `EventHandler`, is so useful — as soon as you see it, you know exactly what your event handler method needs to look like.

Q: **Q: Wait, "standard" event handler? There are other kinds of event handlers?**

A: **A:** Yes! Your events don't ***have to*** send an object and an `EventArgs`. In fact, they can send anything at all — or nothing at all! Look at the IntelliSense window at the bottom of the facing page. Notice how the `OnDragEnter` method takes a `DragEventArgs` reference instead of an `EventArgs` reference? `DragEventArgs` inherits from `EventArgs`, just like `BallEventArgs` does. The page's `DragDrop` event doesn't use `EventHandler`. It uses something else, `DragEventHandler`, and if you want to handle it, your event handler method needs to take an object and a `DragEventArgs` reference.
The parameters of the event are defined by a *delegate* — `EventHandler` and `DragEventHandler` are two examples of delegates. But we'll talk more about that in a minute.

Q: **Q: So I can probably have my event handlers return something other than void, too, right?**

A: **A:** Well, you can, but it's often a bad idea. If you don't return `void` from your handler, you can't *chain* event handlers. That means you can't connect more than one handler to each event. Since chaining is a handy feature, you'd do best to always return `void` from your event handlers.

Q: **Q: Chaining? What's that?**

A: **A:** It's how more than one object can subscribe to the same event — they chain their event handlers onto the event, one after another. We'll talk a lot more about that in a minute, too.

Q: **Q: Is that why I used += when when I added my event handler? Like I'm somehow adding a new handler to existing handlers?**

A: **A:** Exactly! Any time you add an event handler, you want to use `+=`. That way, your handler doesn't replace existing handlers. It just becomes one in what may be a very long chain of other event handlers, all of which are listening to the same event.

Q: **Q: Why does the ball use "this" when it raises the `BallInPlay()` event?**

A: **A:** Because that's the first parameter of the standard event handler. Have you noticed how every `Click` event handler method has a parameter "object sender"? That parameter is a **reference to the object that's raising the event**. So if you're handling a button click, `sender` points to the button that was clicked. And if you're handling a `BallInPlay` event, `sender` will point to the `Ball` object that's in play — and the ball sets that parameter to `this` when it raises the event.

A SINGLE event is always raised by a SINGLE object.
But a SINGLE event can be responded to by MULTIPLE objects.

The IDE generates event handlers for you automatically

Many programmers follow the same convention for naming their event handlers. If there's a `Ball` object that has a `BallInPlay` event and the name of the reference holding the object is called `ball`, then the event handler would typically be named `ball_BallInPlay()`. That's not a hard-and-fast rule, but if you write your code like that, it'll be a lot easier for other programmers to read.

Luckily, the IDE makes it really easy to name your event handlers this way. It has a feature that **automatically adds event handler methods for you** when you're working with a class that raises an event. It shouldn't be too surprising that the IDE can do this for you — after all, this is exactly what it does when you double-click on a button in the designer. (This may seem familiar because you've done it in earlier chapters.)

DO THIS

- **❶ Start a new blank Windows Store app and add the `Ball` and `BallEventArgs`.**
  Here's the `Ball` class:

```
class Ball {
    public event EventHandler BallInPlay;
    public void OnBallInPlay(BallEventArgs e) {
        EventHandler ballInPlay = BallInPlay;
        if (ballInPlay != null)
            ballInPlay(this, e);
    }
}
```

  And here's the `BallEventArgs` class:

```
class BallEventArgs : EventArgs {
    public int Trajectory { get; private set; }
    public int Distance { get; private set; }
    public BallEventArgs(int trajectory, int distance) {
        this.Trajectory = trajectory;
        this.Distance = distance;
    }
}
```

- **❷ Start adding the `Pitcher`'s constructor.**
  Add a new `Pitcher` class to your project. Then give it a constructor that takes a `Ball` reference called `ball` as a parameter. There will be one line of code in the constructor to add its event handler to `ball.BallInPlay`. Start typing the statement, but **don't type += yet**.

```
public Pitcher(Ball ball) {
    ball.BallInPlay
}
```

- **❸ Type += and the IDE will finish the statement for you.**
  As soon as you type **+=** in the statement, the IDE displays a very useful little box:

  When you press the Tab key, the IDE will finish the statement for you. It'll look like this:

```
public Pitcher(Ball ball) {
    ball.BallInPlay += ball_BallInPlay;
}
```

- **❹ The IDE will add your event handler, too.**

You're not done — you still need to add a method to chain onto the event. Luckily, the IDE takes care of that for you, too. After the IDE finishes the statement, it shows you another box:

```
ball.BallInPlay += ball_BallInPlay;
```

Press TAB to generate handler 'ball_BallInPlay' in this class

Hit the Tab key again to make the IDE add this event handler method to your `Pitcher` class. The IDE will always follow the `objectName_HandlerName()` convention:

```
void ball_BallInPlay(object sender, EventArgs e) {
    throw new NotImplementedException();
}
```

The IDE always fills in this NotImplementedException() as a placeholder, so if you run the code it'll throw an exception that tells you that you still need to implement something it filled in automatically.

- **❺ Finish the pitcher's event handler.**
  Now that you've got the event handler's skeleton added to your class, fill in the rest of its code. The pitcher should catch any low balls; otherwise, he covers first base.

```
void ball_BallInPlay(object sender, EventArgs e) {
    if (e is BallEventArgs) {
        BallEventArgs ballEventArgs = e as BallEventArgs;
        if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
            CatchBall();
        else
            CoverFirstBase();
    }
}
```

Since BallEventArgs is a subclass of EventArgs, we'll downcast it using the as keyword so we can use its properties.

You'll add these methods in a minute.

EXERCISE

It's time to put what you've learned so far into practice. Your job is to complete the `Ball` and `Pitcher` classes, add a `Fan` class, and make sure they all work together with a very basic version of your baseball simulator.

- **❶ COMPLETE THE PITCHER CLASS.**
  Below is what we've got for `Pitcher`. Add the `CatchBall()` and `CoverFirstBase()` methods. Both should create a string saying that the catcher has either caught the ball or run to first base, and add that string to a public `ObservableCollection<string>` called `PitcherSays`.

```
class Pitcher {
    public Pitcher(Ball ball) {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e) {
        if (e is BallEventArgs){
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
                CatchBall();
            else
                CoverFirstBase();
        }
    }
}
```

You'll need to implement these two methods to add a string to the PitcherSays ObservableCollection.

- **❷ WRITE A FAN CLASS.**
  Create another class called `Fan`. `Fan` should also subscribe to the `BallInPlay` event in its constructor. The fan's event handler should see if the distance is greater than 400 feet and the trajectory is greater than 30 (a home run), and grab for a glove to try to catch the ball if it is. If not, the fan should scream and yell. Everything that the fan screams and yells should be added to an `ObservableCollection<string>` called `FanSays`.

> **NOTE**
>
> Look at the output on the facing page to see exactly what it should print.

- **❸ BUILD A VERY SIMPLE SIMULATOR.**
  If you didn't do it already, create a new Windows Store Blank App, replace *MainPage.xaml* with a Basic Page, and add the following `BaseballSimulator` class. Then add it as a static resource to the page.

```
using System.Collections.ObjectModel;

class BaseballSimulator {
    private Ball ball = new Ball();
    private Pitcher pitcher;
    private Fan fan;
    public ObservableCollection<string> FanSays { get { return fan.FanSays; } }
    public ObservableCollection<string> PitcherSays { get { return pitcher.PitcherSays; } }
    public int Trajectory { get; set; }
    public int Distance { get; set; }
    public BaseballSimulator() {
        pitcher = new Pitcher(ball);
        fan = new Fan(ball);
    }
    public void PlayBall() {
        BallEventArgs ballEventArgs = new BallEventArgs(Trajectory, Distance);
        ball.OnBallInPlay(ballEventArgs);
    }
```

```
}
```

- **❹ BUILD THE MAIN PAGE.**
  Can you come up with the XAML just from looking at the screenshot to the right? The two TextBox controls are bound to the `Trajectory` and `Distance` properties of the `BaseballSimulator` static resource, and the pitcher and fan chatter are ListView controls bound to the two `ObservableCollections`.
  See if you can make your simulator generate the above fan and pitcher chatter with three successive balls put into play. Write down the values you used to get the result below:

Ball 1:	**Ball 2:**	**Ball 3:**
Trajectory: _____	Trajectory: _____	Trajectory: _____
Distance: _____	Distance: _____	Distance: _____

EXERCISE SOLUTION

It's time to put what you've learned so far into practice. Your job is to complete the `Ball` and `Pitcher` classes, add a `Fan` class, and make sure they all work together with a very basic version of your baseball simulator.

Here are the `Ball` and `BallEventArgs` from earlier, and the new `Fan` class that needed to be added:

```
class Ball {
    public event EventHandler BallInPlay;
    public void OnBallInPlay(BallEventArgs e) {
        EventHandler ballInPlay = BallInPlay;
        if (ballInPlay != null)
            ballInPlay(this, e);
    }
}
```

The OnBallInPlay() method just raises the BallInPlay event—but it has to check to make sure it's not null; otherwise, it'll throw an exception.

Read-only automatic properties work really well in event arguments because the event handlers only read the data passed to them.

```
class BallEventArgs : EventArgs {
    public int Trajectory { get; private set; }
    public int Distance { get; private set; }
    public BallEventArgs(int trajectory, int distance)
    {
        this.Trajectory = trajectory;
        this.Distance = distance;
    }
}

using System.Collections.ObjectModel;
class Fan {
    public ObservableCollection<string> FanSays = new ObservableCollection<string>();
    private int pitchNumber = 0;

    public Fan(Ball ball) {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e)  {
        pitchNumber++;
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if (ballEventArgs.Distance > 400 && ballEventArgs.Trajectory > 30)
                FanSays.Add("Pitch #" + pitchNumber
                            + ": Home run! I'm going for the ball!");
            else
                FanSays.Add("Pitch #" + pitchNumber + ": Woo-hoo! Yeah!");
        }
    }
}
```

The Fan object's constructor chains its event handler onto the BallInPlay event.

The fan's BallInPlay event handler looks for any ball that's high and long.

The only code-behind that the page needs is this **`Button_Click()` event handler method**:

```
private void Button_Click(object sender, RoutedEventArgs e) {
    baseballSimulator.PlayBall();
}
```

> **NOTE**
>
> This static resource goes in the Page.Resources section.

Here's the XAML for the page. It also needs: `<local:BaseballSimulator x:Name="baseballSimulator"/>`

```
<Grid Grid.Row="1" Margin="120,0" DataContext="{StaticResource ResourceKey=baseballSimulator}">
  <Grid.ColumnDefinitions>
      <ColumnDefinition Width="200" />
      <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <StackPanel Margin="0,0,40,0">
    <TextBlock Text="Trajectory" Style="{StaticResource SubheaderTextBlockStyle}" Margin="0,0,0,20"/>
    <TextBox Text="{Binding Trajectory, Mode=TwoWay}" Margin="0,0,0,20"/>
    <TextBlock Text="Distance" Style="{StaticResource SubheaderTextBlockStyle}" Margin="0,0,0,20"/>
    <TextBox Text="{Binding Distance, Mode=TwoWay}" Margin="0,0,0,20"/>
```

```
      <Button Content="Play ball!" Click="Button_Click"/>
    </StackPanel>
    <StackPanel Grid.Column="1">
      <TextBlock Text="Pitcher says" Style="{StaticResource SubheaderTextBlockStyle}" Margin="0,0,0,20"/>
      <ListView ItemsSource="{Binding PitcherSays}" Height="150"/>
      <TextBlock Text="Fan says" Style="{StaticResource SubheaderTextBlockStyle}" Margin="0,0,0,20"/>
      <ListView ItemsSource="{Binding FanSays}" Height="150"/>
    </StackPanel>
  </Grid>
```

And here's the `Pitcher` class (it needs `using System.Collections.ObjectModel;` at the top):

```
class Pitcher {
    public ObservableCollection<string> PitcherSays = new ObservableCollection<string>();
    private int pitchNumber = 0;

    public Pitcher(Ball ball)     {
        ball.BallInPlay += ball_BallInPlay;
    }

    void ball_BallInPlay(object sender, EventArgs e) {
        pitchNumber++;
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
                CatchBall();
            else
                CoverFirstBase();
        }
    }
    private void CatchBall() {
        PitcherSays.Add("Pitch #" + pitchNumber + ": I caught the ball");
    }
    private void CoverFirstBase() {
        PitcherSays.Add("Pitch #" + pitchNumber + ": I covered first base");
    }
}
```

We gave you the pitcher's BallInPlay event handler. It looks for any low balls.

Here are the values we used to get the output. Yours might be a little different.

Ball 1:
Trajectory: 75
Distance: 105

Ball 2:
Trajectory: 48
Distance: 80

Ball 3:
Trajectory: 40
Distance: 435

Generic EventHandlers let you define your own event types

Take a look at the event declaration in your `Ball` class:

```
public event EventHandler BallInPlay;
```

Now open up any Windows Forms app and take a look at the `Click` event declaration from a button form, and most of the other controls you used in the first part of this book:

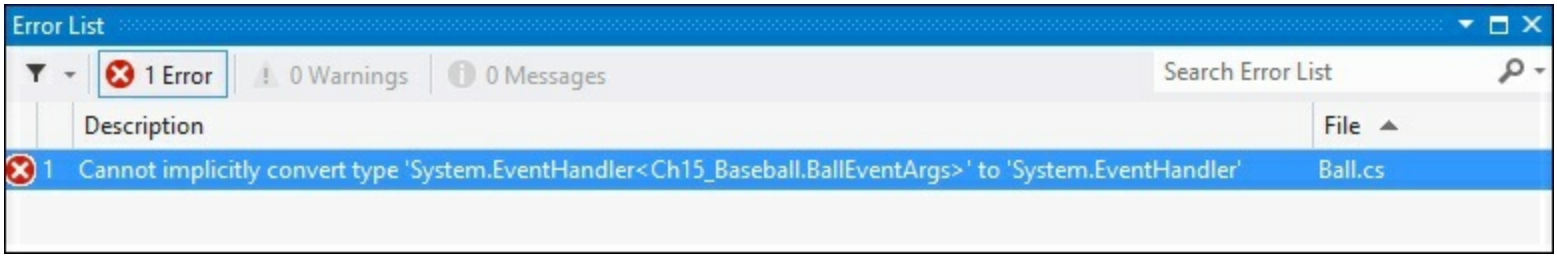
```
public event EventHandler Click;
```

Notice anything? They have different names, but they're declared exactly the same way. And while that works just fine, someone looking at your class declaration doesn't necessarily know that the `BallEventHandler` will always pass it a `BallEventArgs` when the event is fired. Luckily, .NET gives us a great tool to communicate that information very easily: a generic `EventHandler`. Change your ball's `BallInPlay` event handler so it looks like this:

The generic argument to EventHandler has to be a subclass of EventArgs.

```
public event EventHandler<BallEventArgs> BallInPlay;
```

You'll also need to change the `OnBallInPlay` method to replace `EventHandler` with `EventHandler<BallEventArgs>`. Now rebuild your code. You should see this error:



Now that you changed the event declaration, your reference to it in the `Ball` class needs to be updated too:

```
EventHandler<BallEventArgs> ballInPlay = BallInPlay;
if (ballInPlay != null)
    ballInPlay(this, e);
```

C# does implicit conversion when you leave out the new keyword and type

You used the IDE to automatically create this event handler method a few pages ago:

```
ball.BallInPlay += ball_BallInPlay;
```

When you use this syntax, C# does an **implicit conversion** and figures out the type for you. Try replacing that line in the `Pitcher` or `Fan` class with it:

```
ball.BallInPlay += new EventHandler<BallEventArgs>(ball_BallInPlay);
```

Your program still runs just fine because the IDE automatically generated code that used implicit conversion. That way, you didn't have to modify the type when you changed the type of the event.

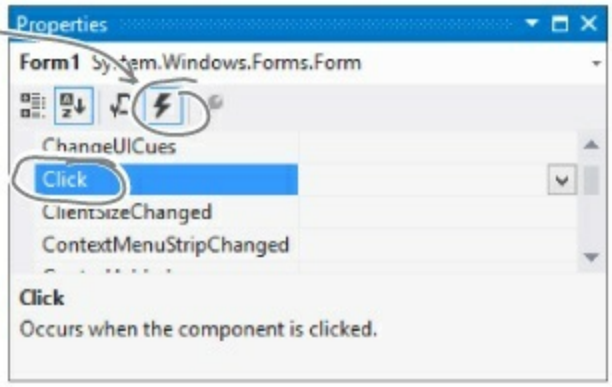
Windows Forms use many different events

We're going to switch gears and go back to desktop applications for the next two projects, because they give us a really good learning tool. That's because every time you've created a button, double-clicked on it in the designer, and written code for a method like `button1_Click()`, you've been working with events. (Windows Store apps use events too.)

DO THIS

- ❶ Create a new **Windows Forms Application project**. Go to the Properties window for the form. Remember those icons at the top of the window? Click on the Events button (it's the one with the lightning bolt icon) to bring up **the events page in the Properties window**:

You can see all of the events for a control: just click on it and then click on this Events button in the Properties window.



Scroll down to Click and double-click on the word Click to make the IDE add a new click event handler to your form that gets fired every time you click on it. And it'll add a line to Form1.Designer.cs to hook the event handler up to the event.

You can create an event that will fire every time someone clicks on the form by selecting Form1_Click next to Click in the events window.

- ❷ Double-click on the Click row in the events page. The IDE will automatically add an event handler method to your form called `Form1_Click`. Add this line of code to it:

```
private void Form1_Click(object sender, EventArgs e) {
    MessageBox.Show("You just clicked on the form");
}
```

- ❸ Visual Studio did more than just write a little method declaration for you, though. It also hooked the event handler up to the `Form` object's `Click` event. Open up *Form1.Designer.cs* and use the Quick Find (Edit→Find and Replace→Quick Find) feature in the IDE to search for the text `Form1_Click` in the current project. You'll find this line of code:

```
this.Click += new System.EventHandler(this.Form1_Click);
```

Now run the program and make sure your code works!

You're not done yet — flip the page!

One event, multiple handlers

Here's a really useful thing that you can do with events: you can **chain** them so that one event or delegate calls many methods, one after another. Let's add a few buttons to your application to see how it works.

- ❹ Add these two methods to your form class:

```
private void SaySomething(object sender, EventArgs e) {
    MessageBox.Show("Something");
}
private void SaySomethingElse(object sender, EventArgs e) {
    MessageBox.Show("Something else");
}
```

- ❺ Now add two buttons to your form. Double-click on each button to add its event handler. Here's the code for both event handlers:

```
private void button1_Click(object sender, EventArgs e) {
    this.Click += new EventHandler(SaySomething);
}
private void button2_Click(object sender, EventArgs e) {
    this.Click += new EventHandler(SaySomethingElse);
}
```

Before you go on, take a minute and think about what those two buttons do. Each button **hooks up a new event handler to the form's `Click` event**. In the first three steps, you used the IDE to add an event handler as usual to pop up a message box every time the form fired its `Click` event — it added code to *Form1.Designer.cs* that used the += operator to hook up its event handler.
Now you added two buttons that use the exact same syntax to chain additional event handlers onto the same `Click` event. So **before you go on**, try to guess what will happen if you run the program, click the first button, then click the second button, and then click on the form. Can you figure it out before you run the program?

THERE ARE NO DUMB QUESTIONS

Q: **Q: When I added a new event handler to the Pitcher object, why did the IDE make it throw an exception?**

A: **A:** It added code to throw a NotImplementedException to remind you that you still need to implement code there. That's a really useful exception, because you can use it as a placeholder just like the IDE did. For example, you'll typically use it when you need to build the skeleton of a class but you don't want to fill in all the code yet. That way, if your program throws that exception, you know it's because you still need to finish the code, and not because your program is broken.

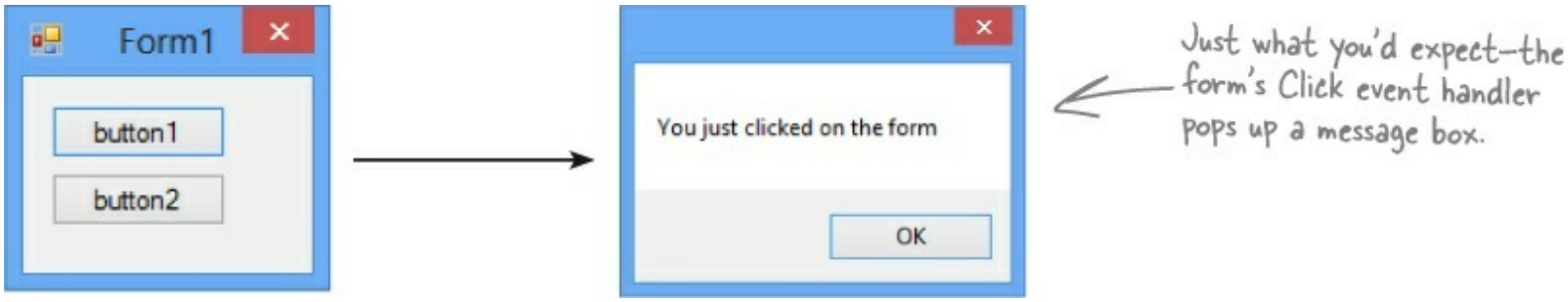
WATCH IT!

Event handlers always need to be "hooked up."

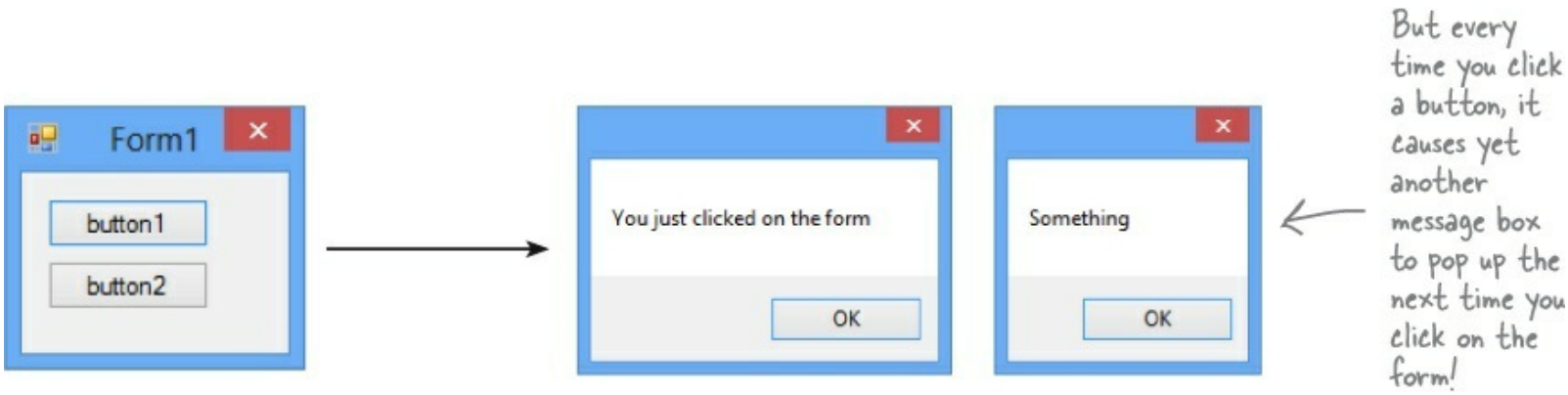
If you drag a button onto your form and add a method called `button1_Click()` *that has the right parameters but* ***isn't registered to listen to your button****, the method won't ever get called. Double-click on the button in the designer — the IDE will see the default event handler name is taken, so it'll add an event handler for the button called* `button1_Click_1()`.

Now run your program and do this:

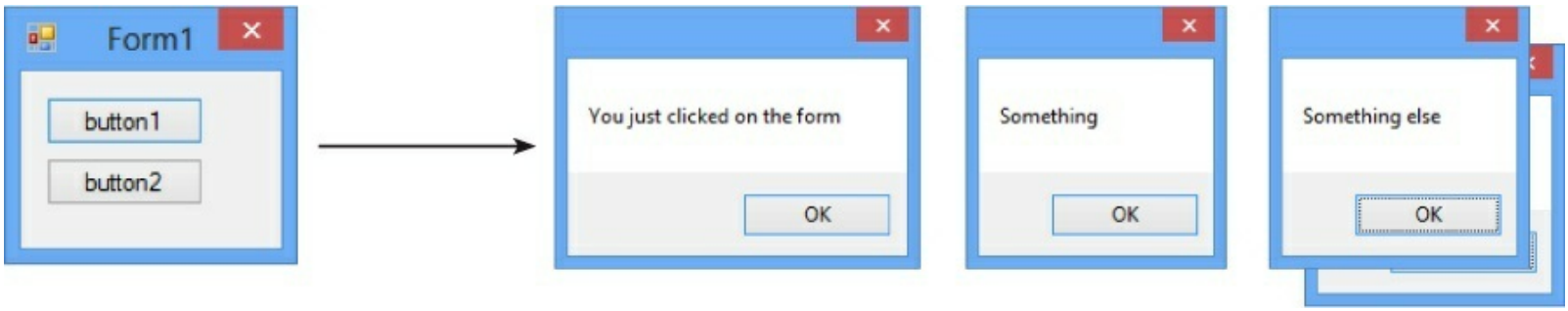
- **Click the form** — you'll see a message box pop up that says, "You just clicked on the form."



- Now **click button1** and then **click on the form again**. You'll see two message boxes pop up: "You just clicked on the form" and then "Something."

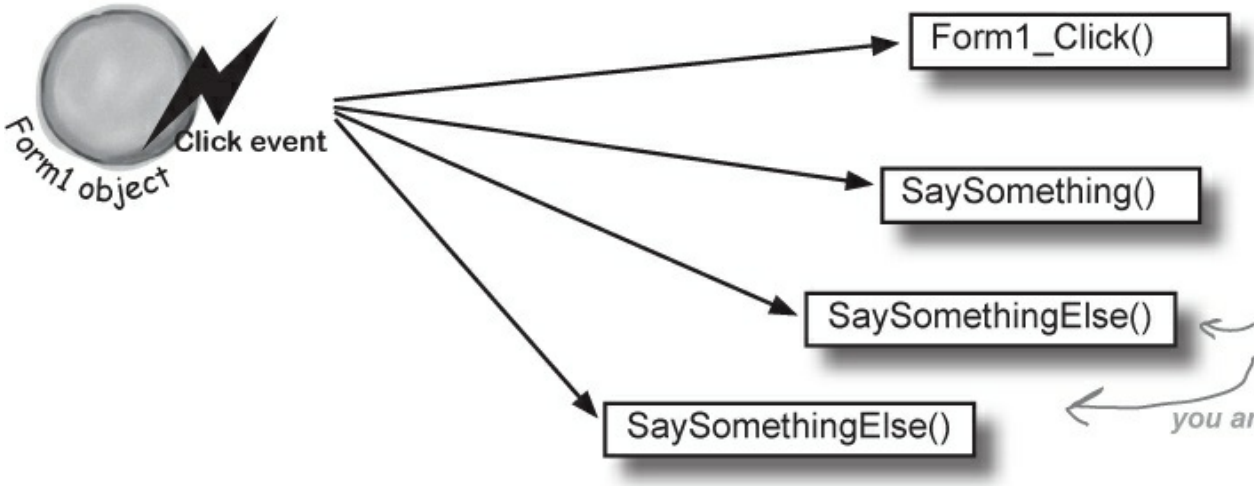


- **Click button2 twice** and then **click on the form again**. You'll see four message boxes: "You just clicked on the form," "Something," "Something else," and "Something else."



So what happened?

Every time you clicked one of the buttons, you chained another method — either `Something()` or `SomethingElse()` — onto the form's `Click` event. You can keep clicking the buttons, and they'll keep **chaining the same methods** onto the event. The event doesn't care how many methods are chained on, or even if the same method is in the chain more than once. It'll just call them all every time the event fires, one after another, in the order they were added.



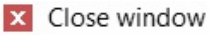
When you click these buttons, they chain different event handlers onto the form's Click event.

That means you won't see anything when you click the buttons! You'll need to click on the form, because the buttons change the form's behavior by modifying its Click event.

The same method can be chained on to an event more than once.

Windows Store apps use events for process lifetime management

> **NOTE**
>
> It's possible for a Windows Store app to terminate itself by calling Application.Current.Exit(), but a well-designed Windows Store app doesn't need to because it can use process lifetime management.

How do you close a Windows Store app? You can click the button in the upper right-hand corner or right-click on the app in the deslktop taskbar and choose 

. But are you really closing the app? Try this: "close" a Windows Store app, then right-click on the time in the desktop and choose Task Manager to see the processes that are currently running. You'll probably see the app you just "closed."

When you switch away from an app, Windows **suspends** it, and while an app is suspended it stays in memory, with all of the objects and resources it needs kept alive. If Windows needs to free up that memory, it will **terminate** the app, unloading it and freeing up any resources it's using. But as a user, do you really care if your app is suspended or terminated? In most cases, users actually don't care — as long as when the app resumes, it returns to a state that makes sense to the user. When an app responds to Windows suspending and resuming it, that's called **process lifetime management**.

Use the IDE to explore process lifetime management events

Open up any Windows Store app and double-click on *App.xaml.cs* in the Solution Explorer. Find the `App` constructor:

```
public App()
{
    this.InitializeComponent();
    this.Suspending += OnSuspending;
}
```

You should recognize what's going on here. `App`, which is a subclass of the `Application` class in the `Windows.UI.Xaml` namespace, has an event called `Suspending`, and it's being hooked up in the constructor to an event handler called OnSuspending. Right-click on and choose Go To Definition to open the Application [from metadata] tab with the members of the `Application` class, and jump to the **`Suspending` event**:

```
//
// Summary:
//     Occurs when the application transitions to Suspended state from some other
//     state.
[SupportedOn(100794368, Platform.Windows)]
[SupportedOn(100859904, Platform.WindowsPhone)]
public event SuspendingEventHandler Suspending;
```

Every time Windows suspends a Windows Store app, the app's Suspending event is fired so that it can save its state.

This event is fired any time the user switches away from your app. This means that the `OnSuspending()` method in *App.xaml.cs* is called every time your app is suspended. And similarly, the `OnLaunched()` method in *App.xaml.cs* is called every time your app is launched.

Once your app is suspended, Windows can terminate the app at any time. So you should build your app to **act like it's going to be terminated every time it's suspended** by saving its current state. The `OnLaunched()` method can check its arguments to see if it's starting again after a previous suspension.

Add process lifetime management to Jimmy's comics

Let's modify Jimmy's comic book app to save and restore the current page. We'll modify its `Suspending` event handler so it writes the name of the current query to a file in the app's local folder when Jimmy switches away from the app. If Windows terminates the app, we'll make sure to switch back to that page when it's launched again.

DO THIS!

- **❶ Add a class to manage saving and loading the state.**
  Add a class called `SuspensionManager`. It has a static property to keep track of the current query, and two static methods to read and write the name of the query to a file called *_sessionState.txt* in the app's *local* folder.

```
using Windows.Storage;
class SuspensionManager {
    public static string CurrentQuery { get; set; }

    private const string filename = "_sessionState.txt";

    static async public Task SaveAsync() {
        if (String.IsNullOrEmpty(CurrentQuery))
            CurrentQuery = String.Empty;
        IStorageFile storageFile =
               await ApplicationData.Current.LocalFolder.CreateFileAsync(
                           filename, CreationCollisionOption.ReplaceExisting);
        await FileIO.WriteTextAsync(storageFile, CurrentQuery);
    }

    static async public Task RestoreAsync() {
        IStorageFile storageFile =
               await ApplicationData.Current.LocalFolder.GetFileAsync(filename);
        CurrentQuery = await FileIO.ReadTextAsync(storageFile);
    }
}
```

- **❷ Make the main page update `SuspensionManager` when a query is loaded.**
  The ListView in *MainPage.xaml* causes the app to navigate when an item is clicked, so **add a line to its `ItemClick` event handler** to set `SuspensionManager`'s static `CurrentQuery` property to the title of the query being loaded:

```
private void ListView_ItemClick(object sender, ItemClickEventArgs e) {
    ComicQuery query = e.ClickedItem as ComicQuery;
    if (query != null) {
        SuspensionManager.CurrentQuery = query.Title;
        if (query.Title == "All comics in the collection")
            this.Frame.Navigate(typeof(QueryDetailZoom), query);
        else
            this.Frame.Navigate(typeof(QueryDetail), query);
    }
}
```

Every time a query is clicked, the event handler updates the static CurrentQuery property.

- **❸ Modify the detail pages' `OnNavigatedFrom()` methods to clear the saved query.**
  When the user clicks the back arrow to navigate away from a page, one of the things that it does is fire the **`NavigatedFrom`** event. In Chapter 14, you modified the `OnNavigatedFrom()` method in code-behind for the detail pages by expanding ⊞ `NavigationHelper registration` in the code editor. Now you should recognize the pattern — the Basic Page template overrides these methods, just you saw a few pages ago.
  We want the two Query Detail pages to clear the SuspensionManager.CurrentQuery property any time the user navigates back to the main page. That way, CurentQuery is only set to a value if the user suspends the app while it's displaying a detail page. Go ahead and expand ⊞ `NavigationHelper registration` and **add this line to the OnNavigatedFrom() method in QueryDetail.xaml.cs**:

```
protected override void OnNavigatedFrom(NavigationEventArgs e) {
    SuspensionManager.CurrentQuery = null;
    base.OnNavigatedFrom(e);
}
```

When the QueryDetail and QueryDetailZoom pages fire their OnNavigatedFrom events to navigate back to the main page, this will clear the SuspensionManager's CurrentQuery property.

Next, do the same thing for **QueryDetailZoom.xaml.cs**.

- **❹ Modify the `Suspending` event handler to save the state.**
  Open *App.xaml.cs* and find the `OnSuspending()` event handler method that was hooked up to the `Suspending` event. It has a comment that starts with `TODO`:

```
private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //TODO: Save application state and stop any background activity
    deferral.Complete();
}
```

Some of the IDE's templates include these "TODO" lines to tell you where it's safe to add code.

Replace the TODO line with a call to `SaveAsync()`. Make sure you add **`async`** to the beginning of the declaration so you can use the `await` keyword to make an asynchronous call:

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    await SuspensionManager.SaveAsync();
    deferral.Complete();
}
```

You're building your own suspension manager to help you learn how Windows Store apps use events to save and restore their state. But have a look in the Common folder—it already has a SuspensionManager class with its own `OnSuspending()` method. When you add a Basic Page to your app, it automatically adds its own suspension manager into the Common folder and namespace. Have a look at this page to learn how the common SuspensionManager works (it will get in your brain faster now that you've written one yourself!):
http://msdn.microsoft.com/en-us/library/windows/apps/Hh986968.aspx

NOTE

Have a look at App.xaml.cs in the Split App you added at the end of Chapter 14. It calls the SuspensionManager's SaveAsync() method in exactly the same way.

- **❺ Update the `OnLaunched` method to restore the state.**
  All of the changes you've made so far keep the `SuspensionManager`'s static `CurrentQuery` property up to date. Now ou can **update the `Launched` event handler in App.xaml.cs** to restore the state. You used the IDE to explore app page navigation in Chapter 14 and saw how the `OnLaunched()` method created a new Frame and navigated to it. Now you'll use the SuspensionManager to restore the state and navigate to the last query that was displayed.
  Start by **adding the `async` keyword** to the method declaration:

```
async protected override void OnLaunched(LaunchActivatedEventArgs args)
{
```

You'll need this to add the async operator because you're using await.

Next, find this comment: `//TODO: Load state from previously suspended application`
That's where you can add code to have the SuspensionManager restore the application state. **Replace the TODO line** with a call to SuspensionManager.RestoreAsync():

```
if (args.PreviousExecutionState == ApplicationExecutionState.Terminated) {
    await SuspensionManager.RestoreAsync();
}
```

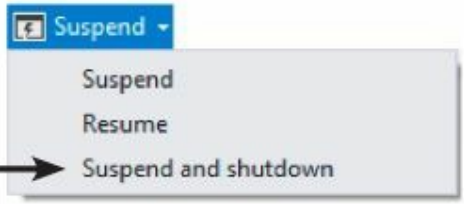
Now that the SuspensionManager has restored the state, its CurrentQuery property will contain the name of the last query that the user dispalyed (or `null` if the user was on the main page when the app was supsended). **Modify the code in `OnLaunched()` that navigates to the root frame** so that it navigates to the QueryDetail or QueryDetailZoom page if CurrentQuery is not null:

Add ths code to compare the previously saved state to the list of queries. If it matches a known query, the app navigates to that query's detail page.

```
// When the navigation stack isn't restored navigate to the first page,
// configuring the new page by passing required information as a navigation
// parameter
rootFrame.Navigate(typeof(MainPage), e.Arguments);
if (!String.IsNullOrEmpty(SuspensionManager.CurrentQuery)) {
    var currentQuerySequence =
        from query in new ComicQueryManager().AvailableQueries
        where query.Title == SuspensionManager.CurrentQuery
        select query;

    if (currentQuerySequence.Count() == 1) {
        ComicQuery query = currentQuerySequence.First();
        if (query != null) {
            if (query.Title == "All comics in the collection")
                rootFrame.Navigate(typeof(QueryDetailZoom), query);
            else
                rootFrame.Navigate(typeof(QueryDetail), query);
        }
    }
}
```

Take a close look at this LINQ query. Do you see how it works?

You can use the Suspend dropdown to test the app. It's in the Debug Location toolbar, which only shows up when the app is running in the debugger (if you don't see it, select it from View→Toolbars). Click Suspend and shutdown to terminate the app and call its `OnSuspending` event. You can also suspend the app using the ✕ button closing the window from the taskbar. Use your modified app to see how apps are suspended without being terminated.



XAML controls use routed events

Flip back a few pages and have a closer look at the IntelliSense window that popped up when you typed `override` into the IDE. Two of the names of the event argument types are a little different than the others. The `DoubleTapped` event's second argument has the type `DoubleTappedRoutedEventArgs`, and the `GotFocus` event's is a `RoutedEventArgs`. The reason is that the `DoubleTapped` and `GotFocus` events are **routed events**. These are like normal events, except for one difference: when a control object responds to a routed event, first it fires off the event handler method as usual. Then it does something else: if the event hasn't been handled, it **sends the routed event up to its container**. The container fires the event, and then if it isn't handled, it sends the routed event up to its container. The event keeps **bubbling up** until it's either handled or it hits the **root**, or the container at the very top. Here's a typical routed event handler method signature.

```
private void EventHandler(object sender, RoutedEventArgs e)
```

The `RoutedEventArgs` object has a property called **`Handled`** that the event handler can use to indicate that it's handled the event. Setting this property to `true` **stops the event from bubbling up**.

In both routed and standard events, the `sender` parameter always contains a reference to the object that called the event handler. So if an event is bubbled up from a control to a container like a Grid, then when the Grid calls its event handler, `sender` will be a reference to the Grid control. But what if you want to find out which control fired the original event? No problem. The `RoutedEventArgs` object has a property called **`OriginalSource`** that contains a reference to the control that initially fired the event. If `OriginalSource` and `sender` point to the same object, then the control that called the event handler is the same control that originated the event and started it bubbling up.

IsHitTestVisible determines if an element is “visible” to the pointer or mouse

Typically, any element on the page can be “hit” by the pointer or mouse — as long as it meets certain criteria. It needs to be visible (which you can change with the `Visibility` property), it has to have a `Background` or `Fill` property that’s not null (but can be `Transparent`), it must be enabled (with the `IsEnabled` property), and it has to have a `height` and `width` greater than zero. If all of these things are true, then the **`IsHitTestVisible` property will return `True`**, and that will cause it to respond to pointer or mouse events.

This property is especially useful if you want to make your events “invisible” to the mouse. If you set `IsHitTestVisible` to `False`, then any pointer taps or mouse clicks will **pass right through the control**. If there’s another control below it, that control will get the event instead.

> **The structure of controls that contain other controls that in turn contain yet more controls is called an object tree, and routed events bubble up the tree from the child to parent until they hit the root element at the top.**

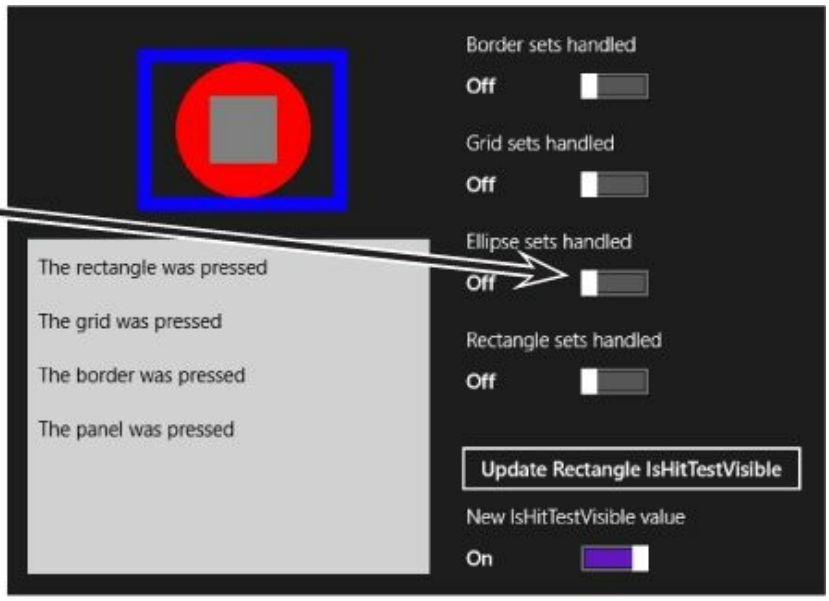
You can see a list of input events that are routed events here: http://msdn.microsoft.com/en-us/library/windows/apps/Hh758286.aspx

Create an app to explore routed events

NOTE

Make sure you replace MainPage.xaml with a new Basic Page.

Here's a Windows Store app that you can use to experiment with routed events. It's got a StackPanel that contains a Border, which contains a Grid, and inside that grid are an Ellipse and a Rectangle. Have a look at the screenshot. See how the Rectangle is on top of the Ellipse? If you put two controls into the same cell, they'll stack on top of each other. But both of those controls have the same parent: the Grid, whose parent is the Border, and the Border's parent is the StackPanel. Routed events from the Rectangle or Ellipse bubble up through the parents to the root of the **object tree**.



This is a **ToggleSwitch** control, which you can use to toggle a value on and off. The header text is set with the `Header` property, and you can set and get its value using the `IsOn` property.

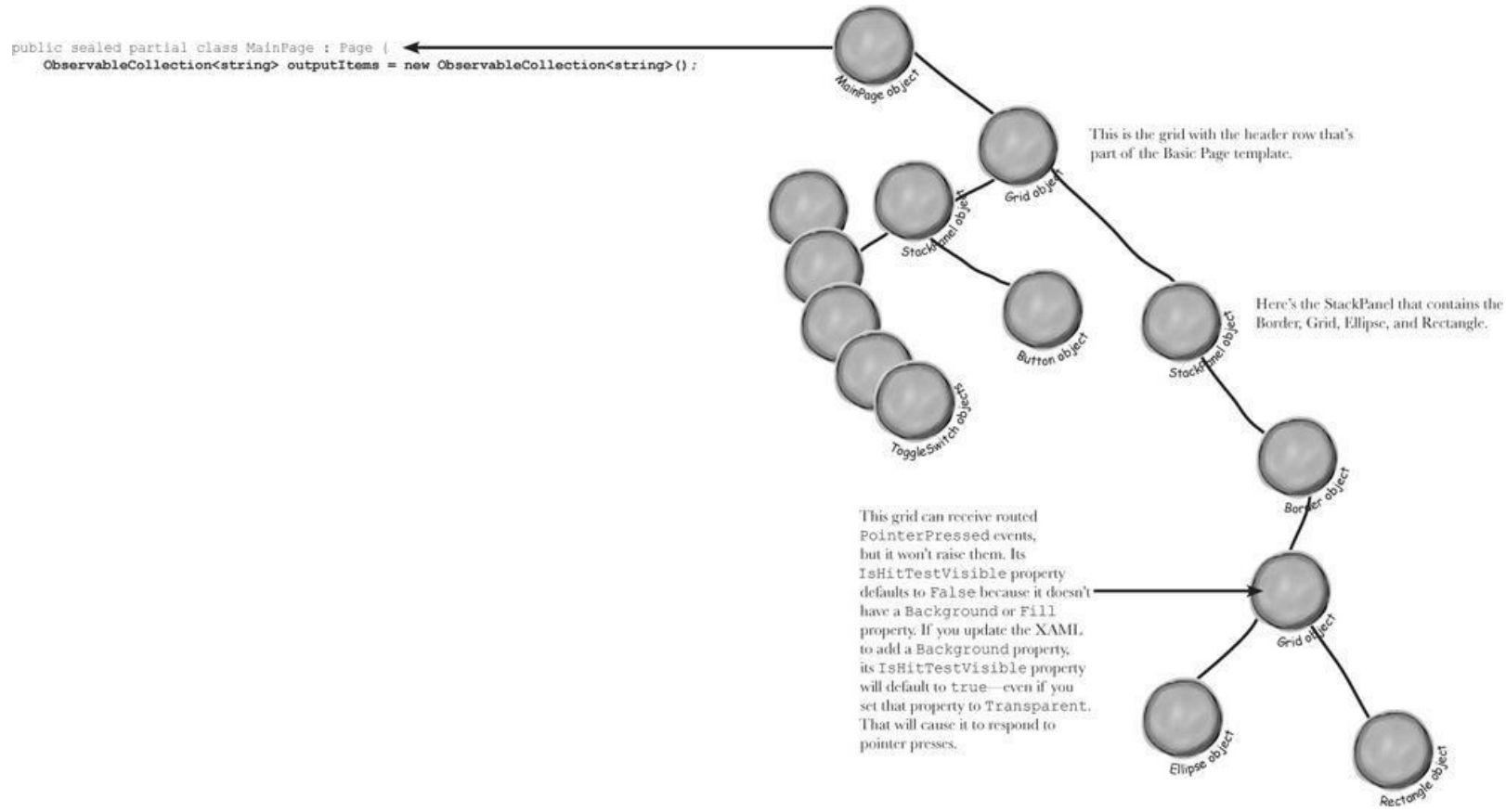
```
<Grid Grid.Row="1" Margin="120,0">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <StackPanel x:Name="panel" PointerPressed="StackPanel_PointerPressed">
        <Border BorderThickness="10" BorderBrush="Blue" Width="155" x:Name="border"
                Margin="20" PointerPressed="Border_PointerPressed">
            <Grid x:Name="grid" PointerPressed="Grid_PointerPressed">
                <Ellipse Fill="Red"  Width="100" Height="100"
                         PointerPressed="Ellipse_PointerPressed"/>
                <Rectangle Fill="Gray" Width="50" Height="50"
                         PointerPressed="Rectangle_PointerPressed" x:Name="grayRectangle"/>
            </Grid>
        </Border>
        <ListBox BorderThickness="1" Width="300" Height="250" x:Name="output" Margin="0,0,20,0"/>
    </StackPanel>

    <StackPanel Grid.Column="1">
        <ToggleSwitch Header="Border sets handled" x:Name="borderSetsHandled"/>
        <ToggleSwitch Header="Grid sets handled" x:Name="gridSetsHandled" />
        <ToggleSwitch Header="Ellipse sets handled" x:Name="ellipseSetsHandled"/>
        <ToggleSwitch Header="Rectangle sets handled" x:Name="rectangleSetsHandled"/>
        <Button Content="Update Rectangle IsHitTestVisible"
                Click="UpdateHitTestButton" Margin="0,20,20,0"/>
        <ToggleSwitch IsOn="True" Header="New IsHitTestVisible value"
                x:Name="newHitTestVisibleValue" />
    </StackPanel>
</Grid>
```

Routed events bubble up the object tree.

IsOn defaults to False. This switch has it set to True because controls always have IsHitTestVisible set to true by default.

Flip the page to finish the app

YOU'LL NEED THIS OBSERVABLECOLLECTION TO DISPLAY OUTPUT IN THE LISTBOX.



Modify the MainPage contsructor to set the output ListBox's `ItemsSource` to this new ObservableCollection. This will let the app display output to the user by clearing and adding items to `outputItems`.

```
public MainPage() {
    this.InitializeComponent();
    this.navigationHelper = new NavigationHelper(this);
    this.navigationHelper.LoadState += navigationHelper_LoadState;
    this.navigationHelper.SaveState += navigationHelper_SaveState;
    output.ItemsSource = outputItems;
}
```

Here's the rest of the code-behind. Each control's `PointerPressed` event handler clears the output if it's the original source, then it adds a string to the output. If its "handled" toggle switch is on, it uses `e.Handled` to handle the event.

```
private void Ellipse_PointerPressed(object sender, PointerRoutedEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The ellipse was pressed");
    if (ellipseSetsHandled.IsOn) e.Handled = true;
}
private void Rectangle_PointerPressed(object sender, PointerRoutedEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The rectangle was pressed");
    if (rectangleSetsHandled.IsOn) e.Handled = true;
}
private void Grid_PointerPressed(object sender, PointerRoutedEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The grid was pressed");
    if (gridSetsHandled.IsOn) e.Handled = true;
}
private void Border_PointerPressed(object sender, PointerRoutedEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The border was pressed");
    if (borderSetsHandled.IsOn) e.Handled = true;
}
private void StackPanel_PointerPressed(object sender, PointerRoutedEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The panel was pressed");
}
private void UpdateHitTestButton(object sender, RoutedEventArgs e) {
    grayRectangle.IsHitTestVisible = newHitTestVisibleValue.IsOn;
}
```

Some routed event handlers will receive a subclass of RoutedEventArgs, such as PointerRoutedEventArgs for the PointerPressed event.

The Click event handler for the button uses the IsOn property of the toggle switch to turn IsHitTestVisible on or off for the Rectangle control.

HERE'S THE OBJECT GRAPH FOR YOUR MAIN PAGE.

The `MainPage` class is at the root of the object tree. *MainPage.xaml* and *MainPage.xaml.cs* **define a subclass of the `Page` class called `MainPage`**. When you use the Basic Page template, it includes code-behind in MainPage.xaml to help with navigation (like you saw in Chapter 14). You'll add your own ObservableCollection to the MainPage to manage the output and use it as the `ItemsSource` for the ListBox that displays the output.

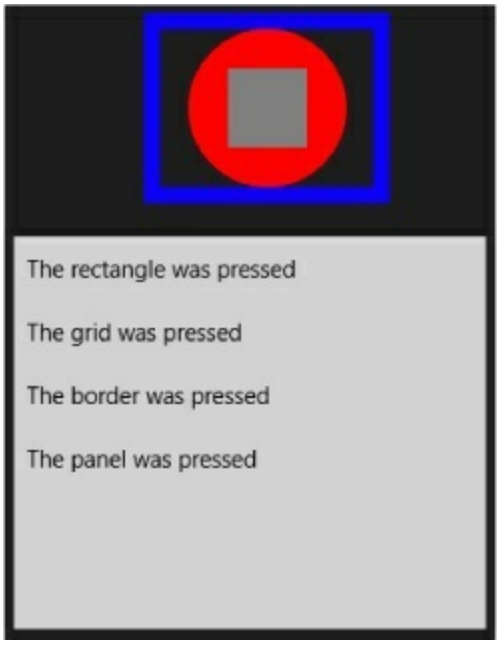
Flip the page to use your new app to explore routed events

RUN THE APP AND CLICK OR TAP THE GRAY RECTANGLE.

You should see the output in the screenshot to the right.

You can see exactly what's going on by putting a breakpoint on the first line of `Rectangle_PointerPressed()`, the Rectangle control's `PointerPressed` event handler:

```
private void Rectangle_PointerPressed(object sender, PointerRoutedEventArgs e)
{
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The rectangle was pressed");
    if (rectangleSetsHandled.IsOn) e.Handled = true;
}
```

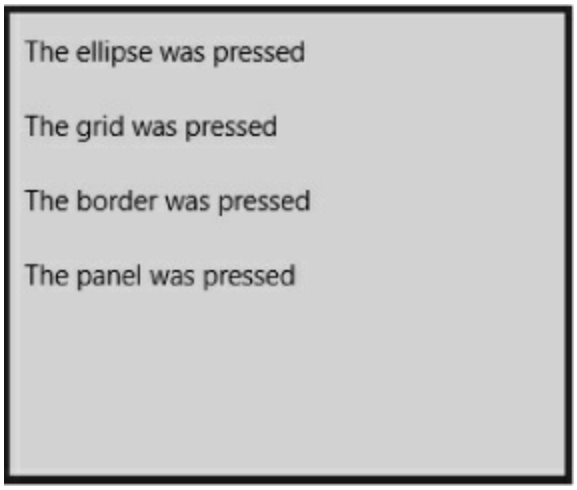


Click the gray rectangle again — this time the breakpoint should fire. Use Step Over (F10) to **step through the code line by line**. First you'll see the `if` block execute to clear the `outputItems ObservableCollection` that's bound to the ListBox. This happens because `sender` and `e.OriginalSource` reference the same Rectangle control, which is only true inside the event handler method for the control that originated the event (in this case, the control that you clicked or tapped), so `sender == e.OriginalSource` is true.

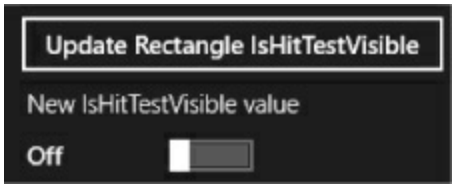
When you get to the end of the method, **keep stepping through the program**. The event will bubble up through the object tree, first running the Rectangle's event handler, then the Grid's event handler, then the Border's, and then the Panel's, and finally it runs an event handler method that's part of `LayoutAwarePage` — this is outside of your code and not part of the routed event, so it will always run. Since none of those controls are the original source for the event, none of their senders will be the same as `e.OriginalSource`, so none of them clear the output.

TURN ISHITTESTVISIBLE OFF, PRESS THE "UPDATE" BUTTON, AND THEN CLICK OR TAP THE RECTANGLE.

← You should see this output.



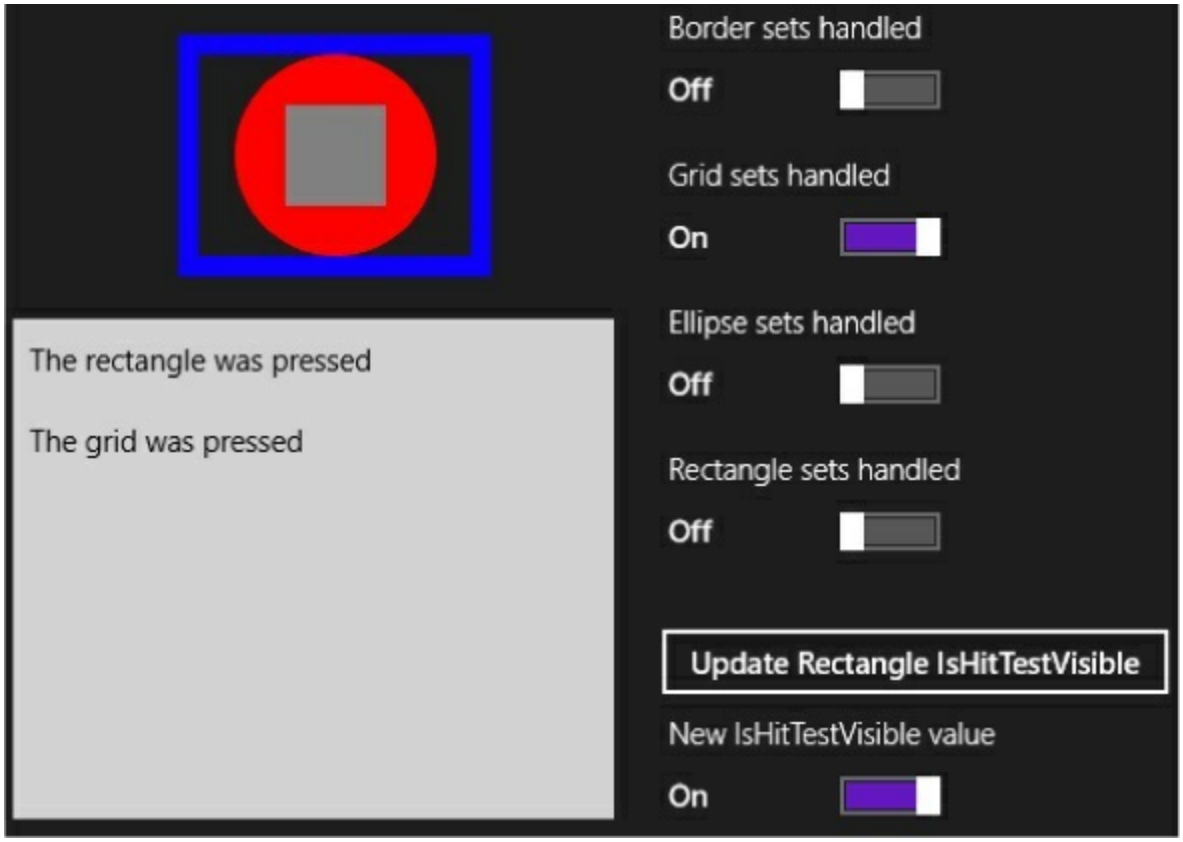
Wait a minute! You pressed the Rectangle, but the Ellipse control's `PointerPressed` event handler fired. What's going on?



When you pressed the button, its `Click` event handler updated the Rectangle control's `IsHitTestVisible` property to `false`, which made it "invisible" to pointer presses, clicks, and other pointer events. So when you tapped the rectangle, your tap passed right through it to the topmost control underneath it on the page that has `IsHitTestVisible` set to `true` and has a `Background` property that's set to a color or `Transparent`. In this case, it finds the Ellipse control and fires its `PointerPressed` event.

TOGGLE THE "GRID SETS HANDLED" SWITCH ON AND CLICK OR TAP THE GRAY RECTANGLE.

You should see this output. →



So why did only two lines get added to the output ListBox? **Step through the code again** to see what's going on. This time, `gridSetsHandled.IsOn` was true because you toggled the `gridSetsHandled` to `On`, so the last line in **the Grid's event handler set `e.IsHandled` to `true`**. As soon as a routed event handler method does that, **the event stops bubbling up**. As soon as the Grid's event handler completes, the app sees that the event has been handled, so it doesn't call the Border or Panel's event handler method, and instead skips to the event handler method in `LayoutAwarePage` that's outside of the code you added.

> **A routed event first fires the event handler for the control that originated the event, and then bubbles up through the control hierarchy until it hits the top — or an event handler sets e.Handled to true.**

USE THE APP TO EXPERIMENT WITH ROUTED EVENTS.

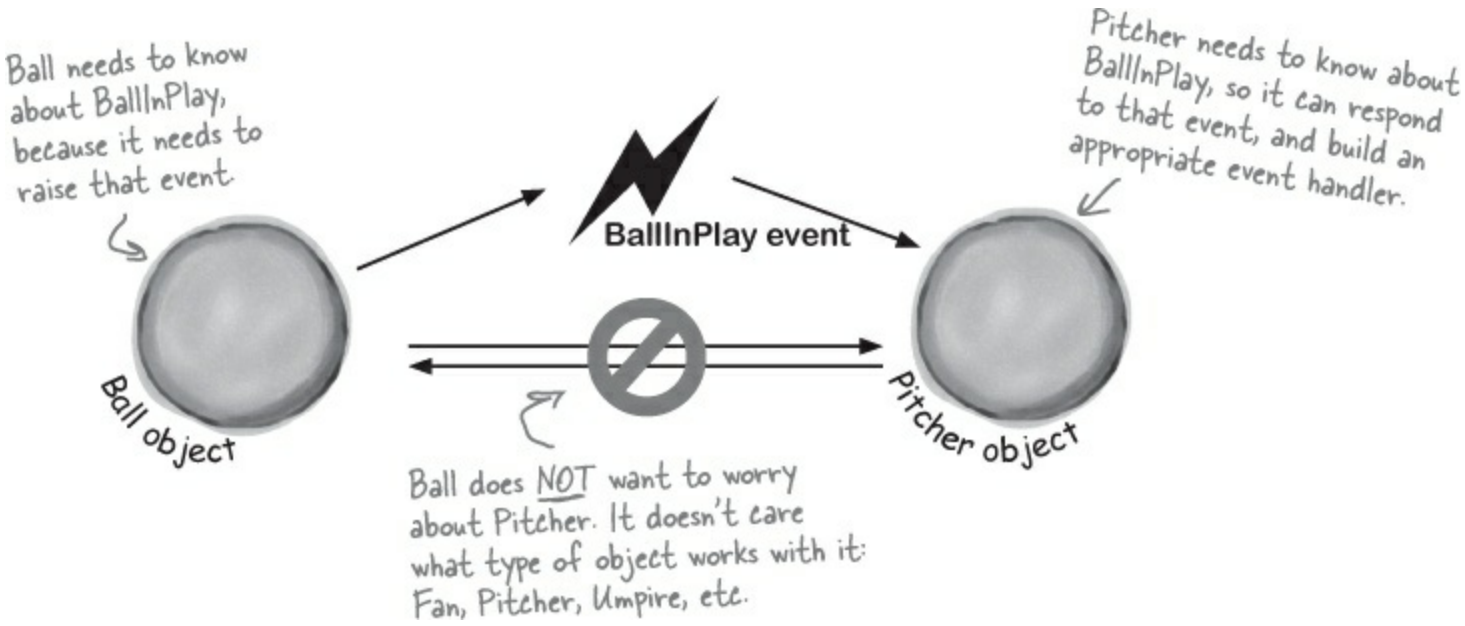
Here are a few things to try:

- Click on the gray rectangle and the red ellipse and watch the output to see how the events bubble up.
- Turn on each of the toggle switches, starting at the top, to cause the event handlers to set `e.Handled` to `true`. Watch the events stop bubbling when they're handled.
- Set breakpoints and debug through all of the event handler methods.
- Try setting a breakpoint in the Ellipse's event handler method, then turn the gray rectangle's `IsHitTestVisible` property on and off by toggling the bottom switch and pressing the button. Step through the code for the rectangle when `IsHitTestVisible` is set to `false`.
- Stop the program and add a `Background` property to the Grid to make it visible to pointer hits.

Connecting event senders with event listeners

One of the trickiest things about events is that the **sender** of the event has to know what kind of event to send — including the arguments to pass to the event. And the **listener** of the event has to know about the return type and the arguments its handler methods must use.

But — and here's the tricky part — you can't tie the sender and receiver *together*. You want the sender to send the event and *not worry about who receives it*. And the receiver cares about the event, *not the object that raised the event*. So both sender and receiver focus on the event, not each other.



“My people will get in touch with your people.”

You know what this code does:

```
Ball currentBall;
```

It creates a **reference variable** that can point to any `Ball` object. It’s not tied to a single `Ball`. Instead, it can point to any ball object — or it can be `null`, and not point to anything at all.

An event needs a similar kind of reference — except instead of pointing to an object, it needs one that **points to a method**. Every event needs to keep track of a list of methods that are subscribed to it. You’ve already seen that they can be in other classes, and they can even be private. So how does it keep track of all of the event handler methods that it needs to call? It uses something called a **delegate**.

NOTE

del-e-gate, noun.

a person sent or authorized to represent others. *The president sent a **delegate** to the summit.*

A delegate STANDS IN for an actual method

One of the most useful aspects of events is that when an event fires, it **has no idea** whose event handler methods it's calling. Anyone who happens to subscribe to an event gets his event handler called. So how does the event manage that?

It uses a C# type called a **delegate**. A delegate is a special kind of reference type that lets you **refer to a method inside a class**...and delegates are the basis for events.

You've actually already been using delegates throughout this chapter! When you created the `BallInPlay` event, you used `EventHandler`. Well, an `EventHandler` is just a delegate. If you right-click on `EventHandler` in the IDE and select Go To Definition, this is what you'll see (try it yourself):

When you create a delegate, all you need to do is specify the signature of methods that it can point to.

So this delegate can be used to reference any method that takes an object and an EventArgs and has no return value.

```
public delegate void EventHandler(object sender, EventArgs e);
```

This specifies the return value of the delegate's signature—which means an EventHandler can only point to methods with void return values.

The name of this delegate is EventHandler.

A delegate adds a new type to your project

DO THIS

When you add a delegate to your project, you're adding a **delegate type**. And when you use it to create a field or variable, you're creating an **instance** of that delegate type. **So create a new Console Application project**. Then add a new class file to the project called *ConvertsIntToString.cs*. But instead of putting a class inside it, add a single line:

```
delegate string ConvertsIntToString(int i);
```

ConvertsIntToString is a delegate type that you've added to your project. Now you can use it to declare variables. This is just like how you can use a class or interface as a type to define variables.

Next, add a method called `HiThere()` to your `Program` class:

```
private static string HiThere(int i)
{
    return "Hi there! #" + (i * 100);
}
```

This method's signature matches ConvertsIntToString.

Finally, fill in the `Main()` method:

someMethod is a variable whose type is ConvertsIntToString. It's a lot like a reference variable, except instead of putting a label on an object on the heap you're putting a label on a method.

```
static void Main(string[] args)
{
    ConvertsIntToString someMethod = new ConvertsIntToString(HiThere);
    string message = someMethod(5);
    Console.WriteLine(message);
    Console.ReadKey();
}
```

You can set someMethod just like any other variable. When you call it like a method, it calls whatever method it happens to point to.

The `someMethod` variable is pointing to the `HiThere()` method. When your program calls `someMethod(5)`, it calls `HiThere()` and passes it the argument 5, which causes it to return the string value "`Hi there! #500`" — exactly as if it were called directly. Take a minute and step through the program in the debugger to see exactly what's going on.

Delegates in action

There's nothing mysterious about delegates — in fact, they don't take much code at all to use. Let's use them to help a restaurant owner sort out his top chef's secret ingredients.

DO THIS

- **❶ Create a new Windows Forms Application project and add a delegate.**
  Delegates usually appear outside of any other classes, so add a new class file to your project and call it *GetSecretIngredient.cs*. It will have exactly one line of code in it:

Delete the class declaration from the new class file and replace it with this line of code.

```
delegate string GetSecretIngredient(int amount);
```

  (Make sure you delete the class declaration entirely.) This delegate can be used to create a variable that can point to any method that takes one `int` parameter and returns a string.

- **❷ Add a class for the first chef, Suzanne.**
  Suzanne.cs will hold a class that keeps track of the first chef's secret ingredient. It has a private method called `SuzannesSecretIngredient()` with a signature that matches `GetSecretIngredient`. But it also has a read-only property — and check out that property's type. It returns a `GetSecretIngredient`. So other objects can use that property to get a reference to her `SuzannesSecretIngredient()` method — the property can return a delegate reference to it, even though it's private.

Suzanne's secret ingredient method takes an int called amount and returns a string that describes her secret ingredient.

```
class Suzanne {
  public GetSecretIngredient MySecretIngredientMethod {
    get {
      return new GetSecretIngredient(SuzannesSecretIngredient);
    }
  }
  private string SuzannesSecretIngredient(int amount) {
    return amount.ToString() + " ounces of cloves";
  }
}
```

- **❸ Then add a class for the second chef, Amy.**
  Amy's method works a lot like Suzanne's:

The AmysSecretIngredientMethod property returns a new instance of the GetSecretIngredient delegate that's pointing to her secret ingredient method.

```
class Amy {
    public GetSecretIngredient AmysSecretIngredientMethod {
        get {
            return new GetSecretIngredient(AmysSecretIngredient);
        }
    }
    private string AmysSecretIngredient(int amount) {
        if (amount < 10)
            return amount.ToString()
                    + " cans of sardines -- you need more!";
        else
            return amount.ToString() + " cans of sardines";
    }
}
```

Amy's secret ingredient method also takes an int called amount and returns a string, but it returns a different string from Suzanne's.

- **❹ Build this form.**
  Here's the code for the form:

```
GetSecretIngredient ingredientMethod = null;
Suzanne suzanne = new Suzanne();
Amy amy = new Amy();

private void useIngredient_Click(object sender, EventArgs e) {
    if (ingredientMethod != null)
        MessageBox.Show("I'll add " + ingredientMethod((int)amount.Value));
    else
        MessageBox.Show("I don't have a secret ingredient!");
}

private void getSuzanne_Click(object sender, EventArgs e) {
    ingredientMethod = new GetSecretIngredient(suzanne.MySecretIngredientMethod);
}

private void getAmy_Click(object sender, EventArgs e) {
    ingredientMethod = new GetSecretIngredient(amy.AmysSecretIngredientMethod);
}
```

Make sure your NumericUpDown control is named "amount".

> **You can use implicit conversion with delegates, just like you did with events earlier in the chapter. Try replacing this:** `= new GetSecretIngredient(suzanne.MySecretIngredientMethod);` **with this:** `= suzanne.MySecretIngredientMethod;`

- **❺ Use the debugger to explore how delegates work.**
  You've got a great tool — the IDE's debugger — that can really help you get a handle on how delegates work:

  Start by running your program. First click the "Get the ingredient" button — it should pop up a message box that says, "I don't have a secret ingredient!"
  Click the "Get Suzanne's delegate" button — that takes the form's `ingredientMethod` field (which is a `GetSecretIngredient` delegate) — and sets it equal to whatever Suzanne's `MySecretIngredientMethod` property returns. That property returns a new

instance of the `GetSecretIngredient` type that's pointing to the `SuzannesSecretIngredient()` method.
Click the "Get the ingredient" button again. Now that the form's `ingredientMethod` field is pointing to `SuzannesSecretIngredient()`, it calls that, passing it the value in the `numericUpDown` control (make sure it's named **amount**) and showing it in a message box.
Click the "Get Amy's delegate" button. It uses the `Amy.AmysSecretIngredientMethod` property to set the form's `ingredientMethod` field to point to the `AmysSecretIngredient()` method.
Click the "Get the ingredient" button one more time. Now it calls Amy's method.
Now **use the debugger** to see exactly what's going on. Place a breakpoint on the first line of each of the three methods in the form. Then **restart the program** (which resets the `ingredientMethod` so that it's equal to null), and start over with the above five steps. Use the Step Into (F11) feature of the debugger to step through every line of code. Watch what happens when you click "Get the ingredient." It steps right into the `Suzanne` and `Amy` classes, depending on which method the `ingredientMethod` field is pointing to.

POOL PUZZLE

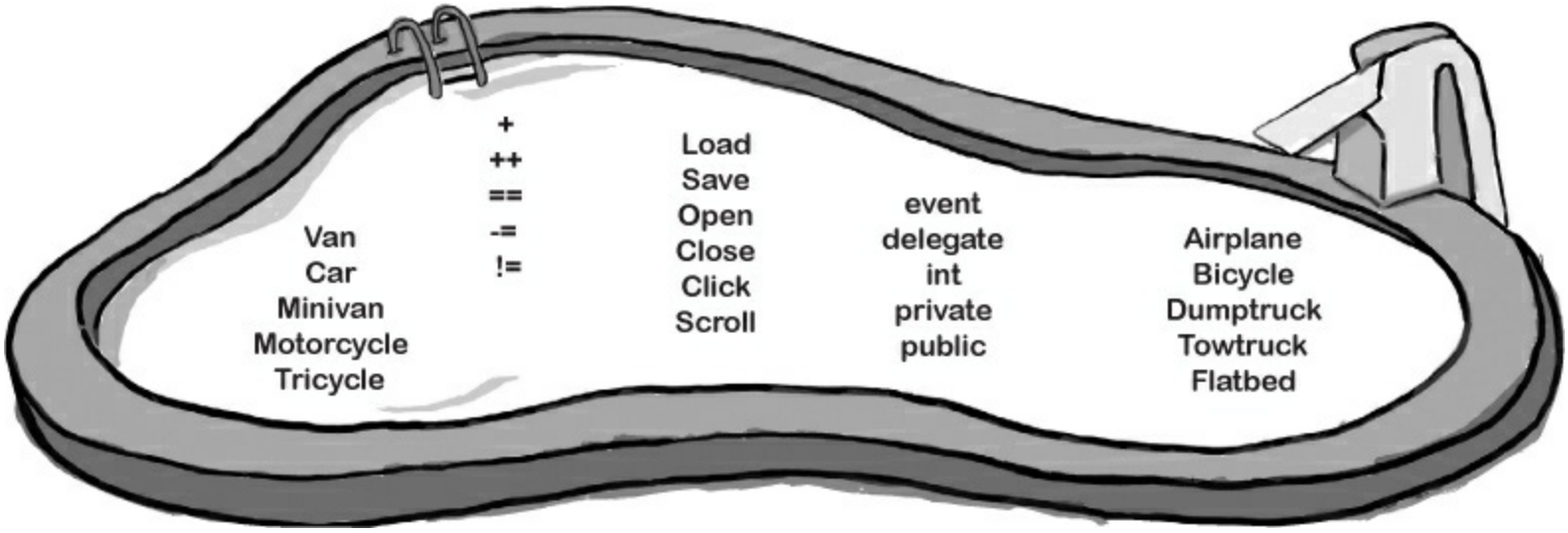
Your ***job*** is to take snippets from the pool and place them into the blank lines in the code. You can use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to complete the code for a form that writes this output to the console when its **`button1`** button is clicked.

```
public Form1() {
    InitializeComponent();
    this._______ += new EventHandler(Minivan);
    this._______ += new EventHandler(_____________);
}
void Towtruck(object sender, EventArgs e) {
    Console.Write("is coming ");
}
void Motorcycle(object sender, EventArgs e) {
    button1._______ += new EventHandler(_____________);
}
void Bicycle(object sender, EventArgs e) {
    Console.WriteLine("to get you!");
}
void _____________(object sender, EventArgs e) {
    button1._______ += new EventHandler(Dumptruck);
    button1._______ += new EventHandler(_____________);
}
void _____________(object sender, EventArgs e) {
    Console.Write("Fingers ");
}
```

Output

```
Fingers is coming to get you!
```

Note: Each thing from the pool can be used more than once.

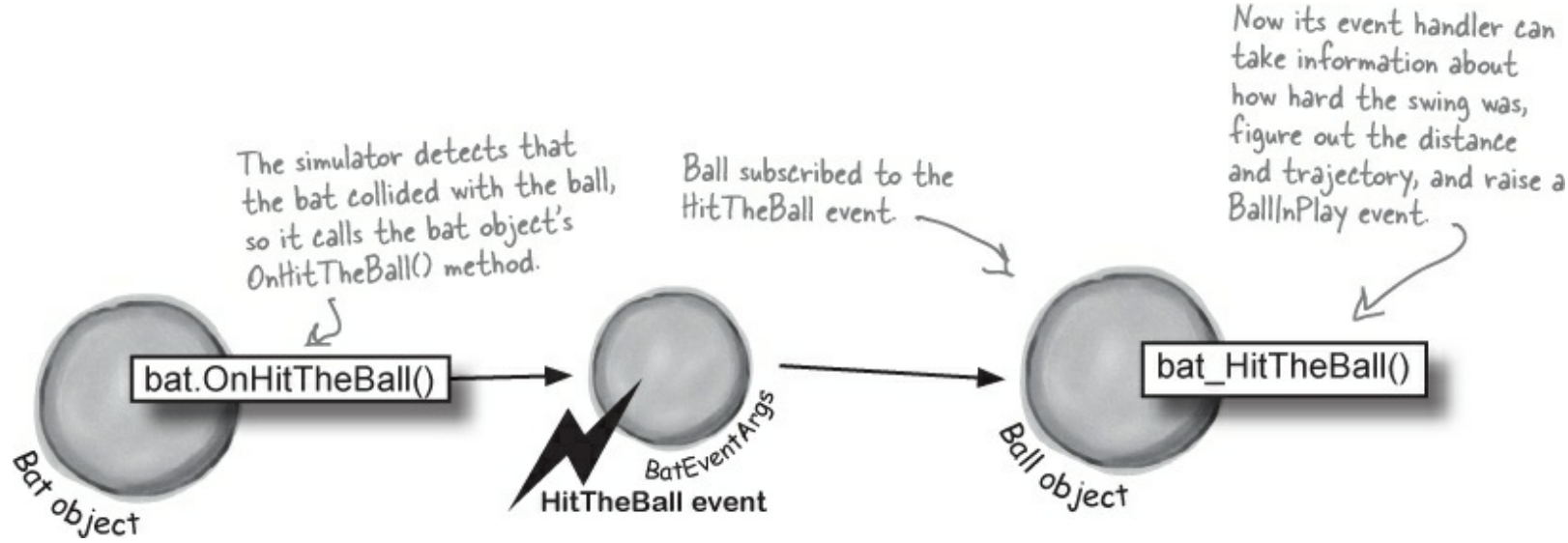


Solution in Add process lifetime management to Jimmy's comics

An object can subscribe to an event...

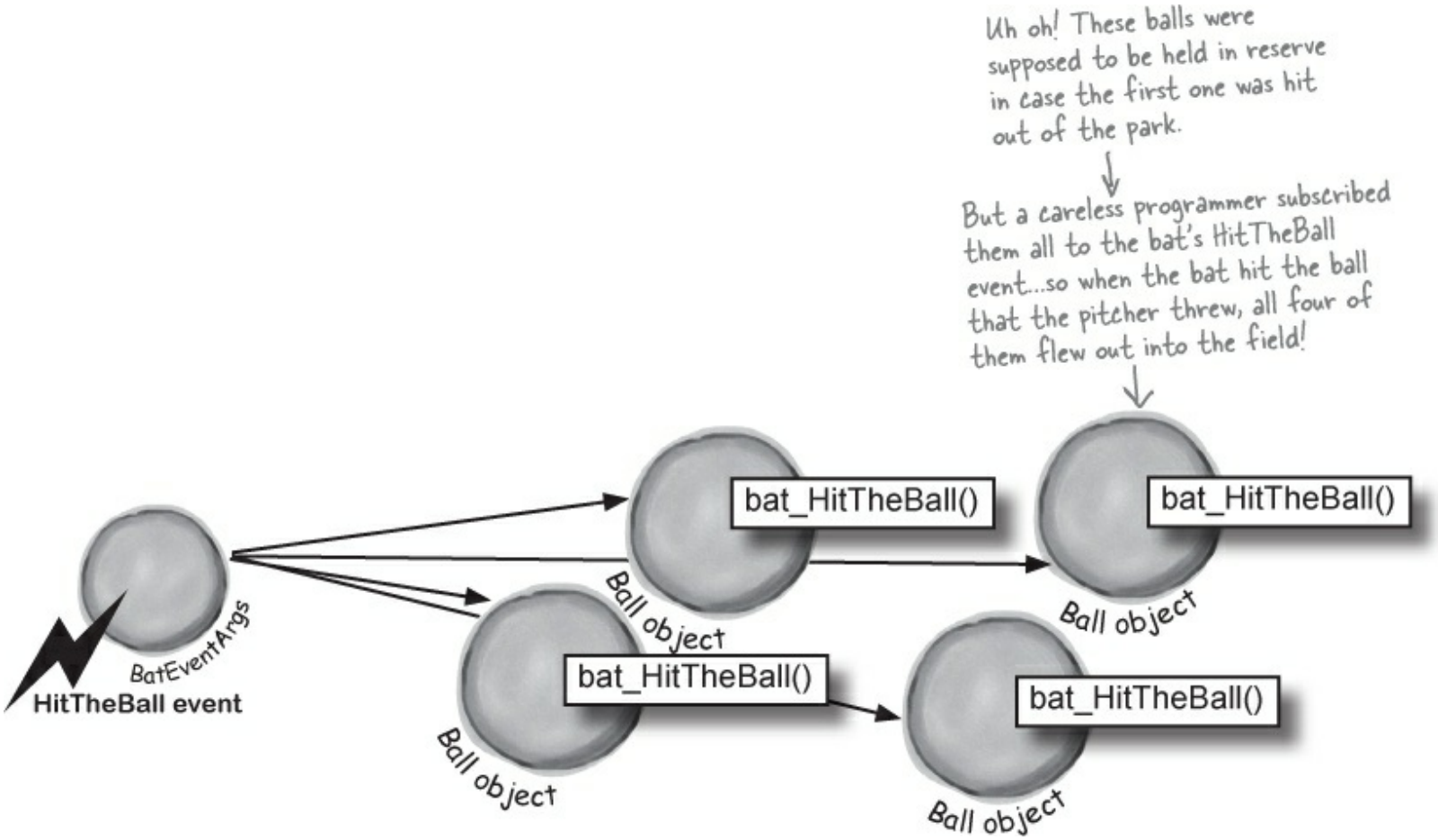
Suppose we add a new class to our simulator, a `Bat` class, and that class adds a `HitTheBall` event into the mix. Here's how it works: if the simulator detects that the player hit the ball, it calls the `Bat` object's `OnHitTheBall()` method, which raises a `HitTheBall` event.

So now we can add a `bat_HitTheBall()` method to the `Ball` class that subscribes to the `Bat` object's `HitTheBall` event. Then, when the ball gets hit, its own event handler calls its `OnBallInPlay()` method to raise its own event, `BallInPlay`, and the chain reaction begins. Fielders field, fans scream, umpires yell...we've got a ball game.



...but that's not always a good thing!

There's only ever going to be one ball in play at any time. But if the `Bat` object uses an event to announce to the ball that it's been hit, then any `Ball` object can subscribe to it. And that means we've set ourselves up for a nasty little bug — what happens if a programmer accidentally adds three more `Ball` objects? Then the batter will swing, hit, and **four different balls will fly** out into the field!

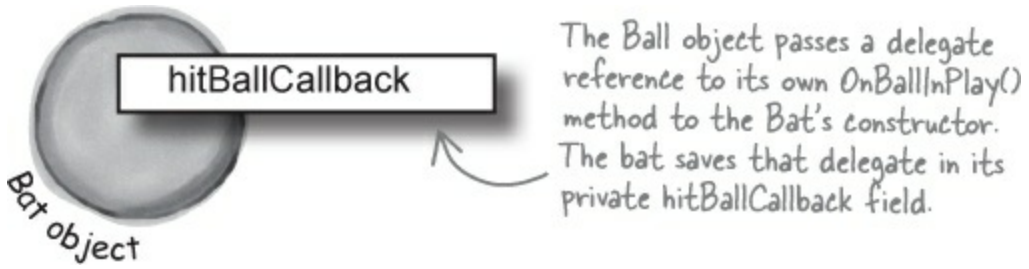


Use a callback to control who's listening

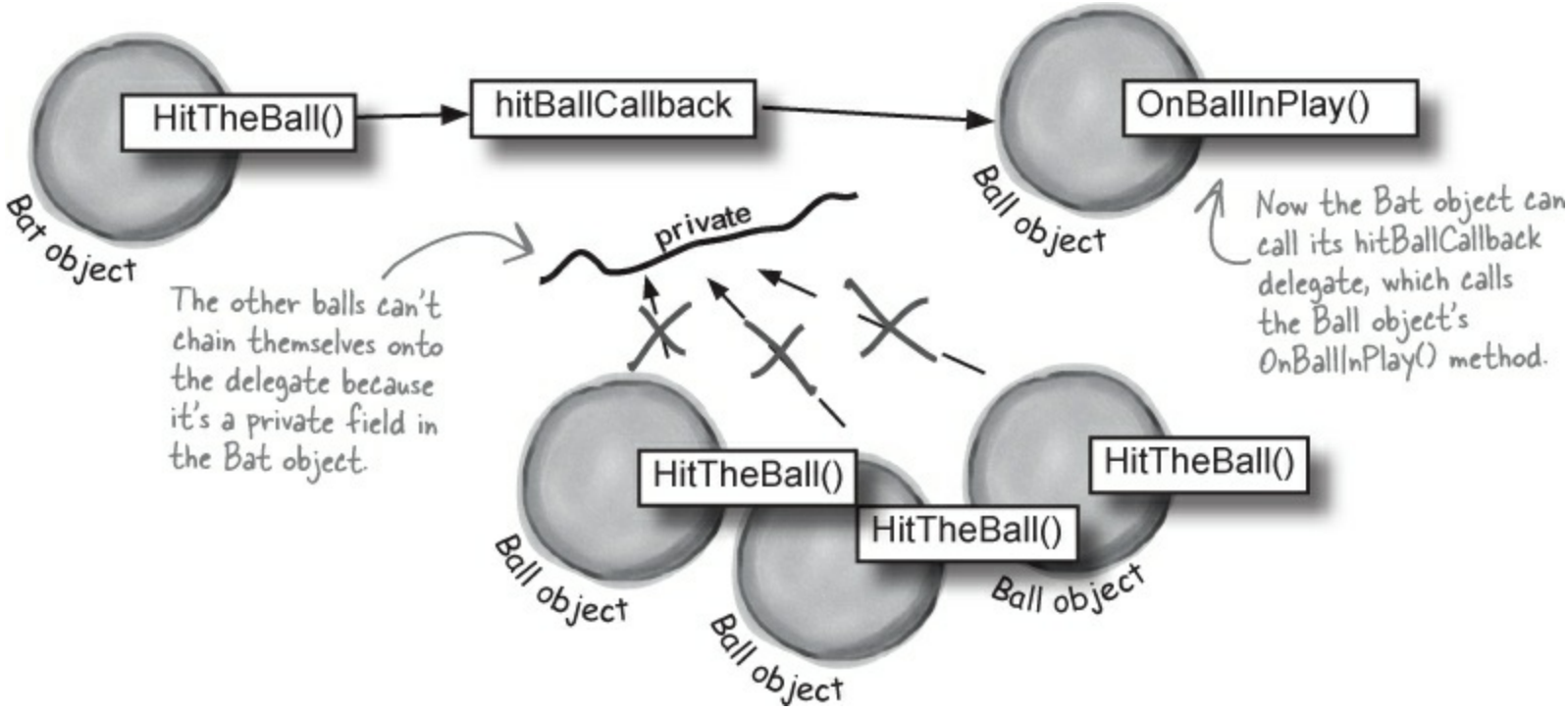
Our system of events only works if we've got one `Ball` and one `Bat`. If you've got several `Ball` objects, and they all subscribe to the public event `HitTheBall`, then they'll all go flying when the event is raised. But that doesn't make any sense...it's really only one `Ball` object that got hit. We need to let the one ball that's being pitched hook itself up to the bat, but we need to do it in a way that doesn't allow any other balls to hook themselves up.

That's where a **callback** comes in handy. It's a technique that you can use with delegates. Instead of exposing an event that anyone can subscribe to, an object uses a method (often a constructor) that takes a delegate as an argument and holds onto that delegate in a private field. We'll use a callback to make sure that the `Bat` notifies exactly one `Ball`:

- **❶ The `Bat` will keep its delegate field private.**
  The easiest way to keep the wrong `Ball` objects from chaining themselves onto the `Bat`'s delegate is for the bat to make it private. That way, it has control over which `Ball` object's method gets called.
- **❷ The `Bat`'s constructor takes a delegate that points to a method in the ball.**
  When the ball is in play, it creates the new instance of the bat, and it passes the `Bat` object a pointer to its `OnBallInPlay()` method. This is called a **callback method** because the `Bat` is using it to call back to the object that instantiated it.



- **❸ When the bat hits the ball, it calls the callback method.**
  But since the bat kept its delegate private, it can be 100% sure that no other ball has been hit. That solves the problem!



FIVE MINUTE MYSTERY

The Case of the Golden Crustacean

Henry "Flatfoot" Hodgkins is a `TreasureHunter`. He's hot on the trail of one of the most prized possessions in the rare and unusual aquatic-themed jewelry markets: a jade-encrusted translucent gold crab. But so are lots of other `TreasureHunter`s. They all got a reference to the same crab in their constructor, but Henry wants to claim the prize ***first***.

In a stolen set of class diagrams, Henry discovers that the `GoldenCrab` class raises a `RunForCover` event every time anyone gets close to it. Even better, the event includes `NewLocationArgs`, which detail where the crab is moving to. But none of the other treasure hunters know about the event, so Henry figures he can cash in.

Henry adds code to his constructor to register his `treasure_RunForCover( )` method as an event handler for the `RunForCover` event on the crab reference he's got. Then, he sends a lowly underling after the crab, knowing it will run away, hide, and raise the `RunForCover` event — giving Henry's `treasure_RunForCover( )` method all the information he needs.

Everything goes according to plan, until Henry gets the new location and rushes to grab the crab. He's stunned to see three other `TreasureHunter`s already there, fighting over the crab.

How did the other treasure hunters beat Henry to the crab?

Answers in Five Minute Mystery Solved.

POOL PUZZLE SOLUTION

The constructor chains two event handlers onto the load events. They get fired off as soon as the form is loaded.

```
public Form1() {
    InitializeComponent();
    this.Load += new EventHandler(Minivan);
    this.Load += new EventHandler(Motorcycle);
}
void Towtruck(object sender, EventArgs e) {
    Console.Write("is coming ");
}
void Motorcycle(object sender, EventArgs e) {

    button1.Click += new EventHandler(Bicycle);
}
void Bicycle(object sender, EventArgs e) {
    Console.WriteLine("to get you!");
}
void Minivan(object sender, EventArgs e) {
    button1.Click += new EventHandler(Dumptruck);
    button1.Click += new EventHandler(Towtruck);
}
void Dumptruck(object sender, EventArgs e) {
    Console.Write("Fingers ");
}
```

The two Load event handlers hook up three separate event handlers to the button's Click event handler.

When the button is clicked, it calls the three event handlers that are chained to it.

Remember, in a WinForms app the Console.WriteLine() writes to the Output window in the IDE. You'll have to open it up from the View menu if it's not already open.

A callback is just a way to use delegates

A callback is a **different way of using a delegate**. It's not a new keyword or operator. It just describes a **pattern** — a way that you use delegates with your classes so that one object can tell another object, "Notify me when this happens — if that's OK with you!"

DO THIS

- **❶ Define another delegate in your baseball project.**
  Since the `Bat` will have a private delegate field that points to the `Ball` object's `OnBallInPlay()` method, we'll need a delegate that matches its signature:

```
delegate void BatCallback(BallEventArgs e);
```

The Bat object's callback will point to a Ball object's OnBallInPlay() method, so the callback's delegate needs to match the signature of OnBallInPlay()—so it needs to take a BallEventArgs parameter and have a void return value.

NOTE

Delegates don't always need to live in their own files. Try putting this one in the same file as Bat. Make sure it's inside the namespace but outside the Bat class.

- **❷ Add the `Bat` class to the project.**
  The `Bat` class is simple. It's got a `HitTheBall()` method that the simulator will call every time a ball is hit. That `HitTheBall()` method uses the `hitBallCallback` delegate to call the ball's `OnBallInPlay()` method (or whatever method is passed into its constructor).

```
class Bat {
    private BatCallback hitBallCallback;
    public Bat(BatCallback callbackDelegate) {
        this.hitBallCallback = new BatCallback(callbackDelegate);
    }
    public void HitTheBall(BallEventArgs e) {
        if (hitBallCallback != null)
            hitBallCallback(e);
    }
}
```

Make sure you check every delegate to make sure it's not null—otherwise, it could throw a null reference exception.

We used = instead of += because in this case, we only want one bat to listen to any one ball, so this delegate only gets set once. But there's nothing stopping you from writing a callback that uses += to call back to multiple methods. The point of the callback is that the object doing the calling is in control of who's listening. In an event, other objects demand to be notified by adding event handlers. In a callback, other objects simply turn over their delegates and politely ask to be notified.

- **❸ We'll need to hook the bat up to a ball.**
  So how does the `Bat`'s constructor get a reference to a particular ball's `OnBallInPlay()` method? Easy — just call that `Ball` object's `GetNewBat()` method, which you'll have to add to `Ball`:

```
public Bat GetNewBat() {
    return new Bat(new BatCallback(OnBallInPlay));
}
```

We set the callback in the Bat object's constructor. But in some cases, it makes more sense to set up the callback method using a public method or property's set accessor.

The Ball's new GetNewBat() method creates a new Bat object, and it uses the BatCallBack delegate to pass a reference to its own OnBallInPlay() method to the new bat. That's the callback method the bat will use when it hits the ball.

- **❹ Now we can encapsulate the `Ball` class a little better.**
  It's unusual for one of the `On...` methods that raise an event to be `public`. So let's follow that pattern with our ball, too, by making its `OnBallInPlay()` method `protected`:

```
protected void OnBallInPlay(BallEventArgs e) {
    EventHandler<BallEventArgs> ballInPlay = BallInPlay;
    if (ballInPlay != null)
        ballInPlay(this, e);
}
```

This is a **really standard pattern** that you'll see over and over again when you work with .NET classes. When a .NET class has an event that gets fired, you'll almost always find a protected method that starts with "On".

  Try running your app again — you'll get an error because OnBallInPlay() is inaccessible.

- **❺ All that's left to do is fixing the `BaseballSimulator` class.**
  `BaseballSimulator` can't call the `Ball` object's `OnBallInPlay()` method anymore — which is exactly what we wanted (and why the IDE now shows an error). Instead, it needs to ask the `Ball` for a new bat in order to hit the ball. And when it does, the Ball object will make sure that its `OnBallInPlay()` method is hooked up to the bat's callback.

```
public void PlayBall() {
    Bat bat = ball.GetNewBat();
    BallEventArgs ballEventArgs =
                    new BallEventArgs(Trajectory, Distance);
    bat.HitTheBall(ballEventArgs);
}
```

If the BaseballSimulator wants to hit a Ball object, it needs to get a new Bat object from that ball. The ball will make sure that the callback is hooked up to the bat. Now when it calls the bat's HitTheBall() method, it calls the ball's OnBallInPlay() method, which fires its BallInPlay event.

But don't take our word for it—pop it open in the debugger!

  Now **run the program** — it should work exactly like it did before. But it's now **protected** from any problems that would be caused by more than one ball listening for the same event.

BULLET POINTS

- When you add a delegate to your project, you're **creating a new type** that stores references to methods.
- Events use delegates to notify objects that actions have occurred.
- Objects subscribe to an object's event if they need to react to something that happened in that object.
- An `EventHandler` is a kind of delegate that's really common when you work with events.
- You can chain several event handlers onto one event. That's why you use += to assign a handler to an event.
- Always check that an event or delegate is not null before you use it to avoid a `NullReferenceException`.
- All of the controls in the toolbox use events to make things happen in your programs.
- When one object passes a reference to a method to another object so it — and only it — can return information, it's called a callback.
- Events let any method subscribe to your object's events anonymously, while callbacks let your objects exercise more control over which delegates they accept.
- Both callbacks and events use delegates to reference and call methods in other objects.
- The debugger is a really useful tool to help you understand how events, delegates, and callbacks work. Take advantage of it!

MessageDialog uses the callback pattern

When you create a `UICommand` for a `MessageDialog`, you can give it a callback using the `UICommandInvokedHandler` delegate. You can also pass it an optional identifier object, and the label and identifier are accessible through the `IUICommand` delegate parameter.

```
MessageDialog dialog = new MessageDialog("Here's a dialog.");
dialog.Commands.Add(new UICommand("My label", MyUiCommandCallback, "My identifier"));
await dialog.ShowAsync();
```

This can be any object, not just a string.

Try adding these lines to an app (don't forget to add `using Windows.UI.Popups;`). You can use the Generate Method Stub IDE command to generate a stub for the callback method.

THERE ARE NO DUMB QUESTIONS

Q: **Q: How are callbacks different from events?**

A: **A:** Events and delegates are part of .NET. They're a way for one object to announce to other objects that something specific has happened. When one object publishes an event, any number of other objects can subscribe to it without the publishing object knowing or caring. When an object fires off an event, if anyone happens to have subscribed to it, then it calls each of their event handlers.
Callbacks are not part of .NET at all — instead, *callback* is just a name for the way we use delegates (or events — there's nothing stopping you from using a private event to build a callback). A callback is just a relationship between two classes where one object requests that it be notified. Compare this to an event, where one object **demands** that it be notified of that event.

Q: **Q: So a callback isn't an actual type in .NET?**

A: **A:** No, it isn't. A callback is a ***pattern*** — it's just a novel way of using the existing types, keywords, and tools that C# comes with. Go back and take another look at the callback code you just wrote for the bat and ball. Did you see any new keywords that we haven't used before? Nope! But it does use a delegate, which **is** a .NET type.
It turns out that there are a lot of patterns that you can use. In fact, there's a whole area of programming called ***design patterns***. A lot of problems that you'll run into have been solved before, and the ones that pop up over and over again have their own design patterns that you can benefit from.

Q: **Q: So callbacks are just private events?**

A: **A:** Not quite. It seems easy to think about it that way, but private events are a different beast altogether. Remember what the `private` access modifier really means? When you mark a class member `private`, only instances of that same class can access it. So if you mark an event `private`, then other instances of the same class can subscribe to it. That's different from a callback, because it still involves one or more objects anonymously subscribing to an event.

Q: **Q: But it looks just like an event, except with the event keyword, right?**

A: **A:** The reason a callback looks so much like an event is that they both use **delegates**. And it makes sense that they both use delegates, because that's C#'s tool for letting one object pass another object a reference to one of its methods.
But the big difference between normal events and callbacks is that an event is a way for a class to publish to the world that some specific thing has happened. A callback, on the other hand, is never published. It's private, and the method that's doing the calling keeps tight control over who it's calling.

> **Check out "Head First Design Patterns" at the Head First Labs website. It's a great way to learn about different patterns that you can apply to your own programs. The first one you'll learn about is called the `Observer` (or Publisher-Subscriber) pattern, and it'll look really familiar to you. One object publishes information, and other objects subscribe to it. Events are the C# way of implementing the Observer pattern.**

You'll often see delegates used with anonymous methods and lambda expressions. Flip to leftover #9 in the appendix to learn more about them.

FIVE MINUTE MYSTERY SOLVED

The Case of the Golden Crustacean

How did the other treasure hunters beat Henry to the crab?

The crux of the mystery lies in how the treasure hunter seeks his quarry. But first we'll need to see exactly what Henry found in the stolen diagrams.

In a stolen set of class diagrams, Henry discovers that the GoldenCrab class raises a RunForCover event every time anyone gets close to it. Even better, the event includes NewLocationArgs, which detail where the crab is moving to. But none of the other treasure hunters know about the event, so Henry figures he can cash in.

```
class GoldenCrab {
    public delegate void Escape(object sender, NewLocationArgs e);
    public event Escape RunForCover;
    public void SomeonesNearby() {
        Escape runForCover = RunForCover;
        if (runForCover != null)
            runForCover(this, new NewLocationArgs("Under the rock");
    }
}
class NewLocationArgs {
    public NewLocationArgs(HidingPlace newLocation) {
        this.newLocation = newLocation;
    }
    private HidingPlace newLocation;
    public HidingPlace NewLocation { get { return newLocation; } }
}
```

Any time someone comes close to the golden crab, its SomeonesNearby() method fires off a RunForCover event, and it finds a place to hide.

So how did Henry take advantage of his newfound insider information?

Henry adds code to his constructor to register his treasure_RunForCover() method as an event handler for the RunForCover event on the crab reference he's got. Then, he sends a lowly underling after the crab, knowing it will run away, hide, and raise the RunForCover event — giving Henry's treasure_RunForCover() method all the information he needs.

```
class TreasureHunter {
    public TreasureHunter(GoldenCrab treasure) {
        treasure.RunForCover += treasure_RunForCover;
    }
    void treasure_RunForCover(object sender, NewLocationArgs e) {
        MoveHere(e.NewLocation);
    }
    void MoveHere(HidingPlace location) {
        // ... code to move to a new location ...
    }
}
```

Henry thought he was being clever by altering his class's constructor to add an event handler that calls his MoveHere() method every time the crab raises its RunForCover event. But he forgot that the other treasure hunters inherit from the same class, and his clever code adds their event handlers to the chain, too!

And that explains why Henry's plan backfired. When he added the event handler to the `TreasureHunter` constructor, he was inadvertently ***doing the same thing for all of the treasure hunters!*** And that meant that every treasure hunter's event handler got chained onto the same `RunForCover` event. So when the Golden Crustacean ran for cover, everyone was notified about the event. And all of that would have been fine if Henry were the first one to get the message. But Henry had no way of knowing when the other treasure hunters would have been called — if they subscribed before he did, they'd get the event first.

Use delegates to use the Windows settings charm

Go to the Windows 8 Start Page and tap the Internet Explorer icon, then open up the charms and tap Settings. Internet Explorer told Windows 8 to add options like Internet Options and About to the Settings charm menu. But when you click the IE About option, it looks very different from the About page for the Maps, Mail, or Windows Store apps. That's because it's up to each app — and, in fact, each page — to tell Windows about its Settings charm options and register a callback that gets called when the user chooses the option. C# Windows Store apps use delegates to do this. Let's use the IDE to explore how this works and **add an About command to the Settings charm for Jimmy's app**.

Open up *MainPage.xaml.cs* and add these two `using` statements and this code to the code-behind:

```
using Windows.UI.ApplicationSettings;
using Windows.UI.Popups;

public sealed partial class MainPage : Page {
    static bool aboutCommandAdded = false;
...
    public MainPage() {
        this.InitializeComponent();
        this.navigationHelper = new NavigationHelper(this);
        this.navigationHelper.LoadState += navigationHelper_LoadState;
        this.navigationHelper.SaveState += navigationHelper_SaveState;

        if (!aboutCommandAdded) {
            SettingsPane.GetForCurrentView().CommandsRequested += MainPage_CommandsRequested;
            aboutCommandAdded = true;
        }
    }
```

We're using a static bool field to make sure the command is only added once. Try removing the line that sets aboutCommandAdded to true—you'll see multiple "About" commands added to the Settings charm.

`SettingsPane` is a static class that lets your app add or remove commands to the Settings charm. It's in the `Windows.UI.ApplicationSettings` namespace.

When you type += the IDE will prompt you to automatically create the event handler method stub. Here's what should go into that event handler. It uses a delegate called `UICommandInvokedHandler`, so add a method called `AboutInvokedHandler()`. That's the method that will get called by the new About setting.

```
void MainPage_CommandsRequested(SettingsPane sender, SettingsPaneCommandsRequestedEventArgs args) {
    UICommandInvokedHandler invokedHandler =
                                new UICommandInvokedHandler(AboutInvokedHandler);
    SettingsCommand aboutCommand = new SettingsCommand("About", "About Jimmy's Comics",
                                            invokedHandler);
    args.Request.ApplicationCommands.Add(aboutCommand);
}

async void AboutInvokedHandler(IUICommand command) {
    await new MessageDialog("An app to help Jimmy manage his comic collection",
                "Jimmy's Comics").ShowAsync();
}
```

Now run your app. Open up the charms, tap Settings, and then choose the About menu option. Your app will call `AboutInvokedHandler` and display the `MessageDialog`.

When you go to the main page, tap the Settings charm, the app now shows an "About option" that pops up a MessageDialog when you choose it.

You can use the Windows key (⊞) to access the charms and app bar.

- Bring up the charms with ⊞ + C
- Pop up the Settings charm with ⊞ + I
- Display the App Bar with ⊞ + Z

Now let's use the IDE to explore how this works. Stop the program and use Go To Definition to get the **definition of `SettingsCommand`** from metadata. It should look like this when you collapse the comments:

```
...public sealed class SettingsCommand : IUICommand
{
    ...public SettingsCommand(object settingsCommandId, string label, UICommandInvokedHandler handler);

    ...public static SettingsCommand AccountsCommand { get; }
    ...public object Id { get; set; }
    ...public UICommandInvokedHandler Invoked { get; set; }
    ...public string Label { get; set; }
}
```

Now use Go To Definition again to see the definition of `UICommandInvokedHandler`:

```
...public delegate void UICommandInvokedHandler(IUICommand command);
```

Walk through the various objects so you can see exactly how this works:

- The `SettingsPane.GetForCurrentView()` method returns an object that has a `CommandsRequested` event. Go back to your code and then go to the definition of `CommandsRequested` to see the event definition.
- The event handler has a `SettingsPaneCommandsRequestedEventArgs` argument. Go into its definition to see the `Request` object that's used in the third line of your event handler.
- The `Request` object has one property: a collection called `ApplicationCommands` that contains `SettingsCommand` objects.
- Go back to your event handler again, because now you can see what it does. When the user taps the Settings charm, the settings pane fires its `CommandsRequested` event to ask apps for commands and callbacks. You hooked a listener up to this event, and had that listener return a `SettingsCommand` that defined the About option, with a delegate that pointed to a method to pop up a `MessageDialog`. When you tap About, the settings pane uses that delegate to call back to `AboutInvokedHandler()`.
- Still not 100% clear? Don't worry. Use the navigation buttons in the toolbar to navigate back and forth through the definitions. Try putting a breakpoint in the constructor and the two methods. Sometimes you need to flip back and forth through the definitions before it all "clicks" in your brain.

You can use the Windows Key + I shortcut to pop up the Settings charm.

Apps can interact with the Search and Share charms, too! Flip to leftover #1 in the appendix to find out where to learn more about it.

Chapter 16. Architecting Apps with the mvvm Pattern: Great apps on the inside and outside

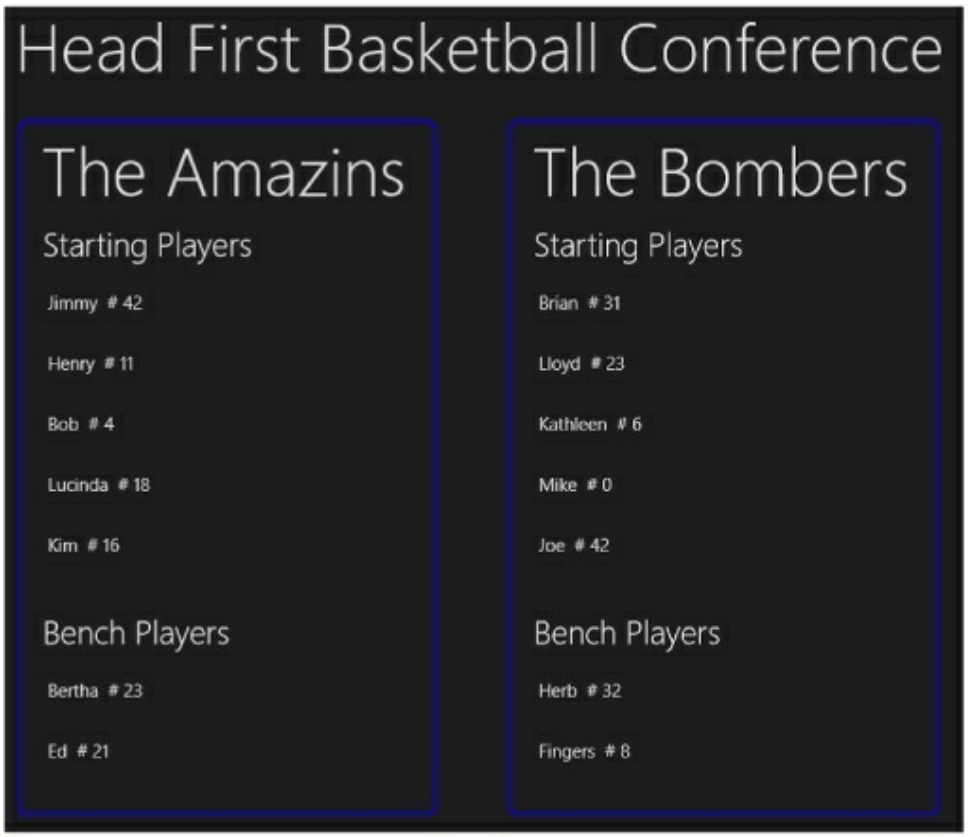
Your apps need to be more than just visually stunning.

When you think of *design*, what comes to mind? An example of great building architecture? A beautifully-laid-out page? A product that's as aesthetically pleasing as it is well engineered? Those same principles apply to your apps. In this chapter you'll learn about **the Model-View-ViewModel pattern** and how you can use it to build well-architected, loosely coupled apps. Along the way you'll learn about **animation** and **control templates** for your apps' visual design, how to use **converters** to make data binding easier, and how to pull it all together to ***lay a solid C# foundation*** to build any app you want.

The Head First Basketball Conference needs an app

Jimmy and Brian are the captains of the two top teams in the Head First Basketball Conference, Objectville's amateur basketball league. They've got some great players, and those players deserve a great app to keep track of who's starting and who's on the bench.

Each team has starting players and bench players, and each player has a name and a number.



There are four different ListView controls on this page, so each one needs its own ObservableCollection to bind its ItemsSource to.

But can they agree on how to build it?

Uh oh — Brian and Jimmy have different ideas about how to build this app, and the argument's starting to get a little heated. It sounds like Brian really wants it to be easy to manage the data that's displayed on the page, while Jimmy cares a lot about simplifying the data binding. This may make for a great off-court rivalry, but it's not going to make it any easier to build the app!

Jimmy: Hold on there, cowboy. Sounds a little short-sighted.

Brian: I'm sure you just don't understand what I'm telling you, so I'll talk real slow and spell it out for you. We'll start with a simple `Player` class that has properties for the name, number, and whether the player's a starter.

Jimmy: Yes, I *understand* what you're saying. But you're not listening to me. You're thinking about how to model the data.

Brian: Clearly. That's where everything starts.

Jimmy: That makes it convenient to create the data.

Brian: You're getting it —

Jimmy: I'm not done. What about the rest of the app? We've got ListView and TextBlock controls that need to display the data. If we don't have collections for the controls to bind to, they won't work.

Brian: Um...

Jimmy: Exactly. So we may need to make a couple of, ah, tactical decisions in our object model.

Brian: You mean, we need to compromise by creating a lousy object model that's hard to work with, because we need something to bind to.

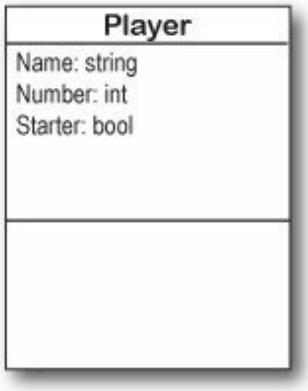
Jimmy: Unless you've got a better idea.

BRAIN POWER

How can you create classes that are easy to bind to, but still have an object model that makes it easy to work with the data?

Do you design for binding or for working with data?

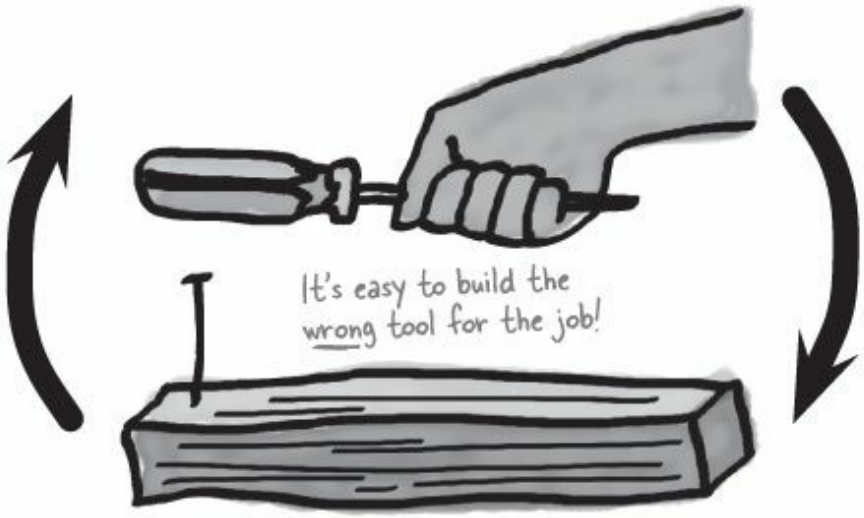
You already know how important it is to build an object model that makes your data easy to work with. But what if you need to do **two different things** with those objects? That's one of the most common problems that you face as an app designer. Your objects need to have public properties and `ObservableCollections` to bind to your XAML controls. But sometimes that makes your data harder to work with, because it forces you to build an unintuitive object model that's difficult to work with.



It's hard to optimize your classes to make it easy to slice and dice your data with LINQ queries...

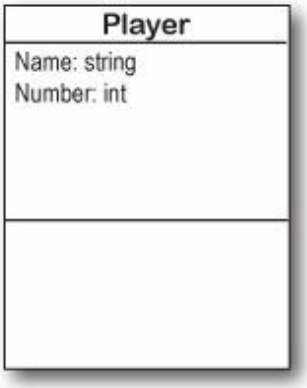
```
var benchPlayers =
    from player in _roster.Players
    where player.Starter == false
    select player;
```

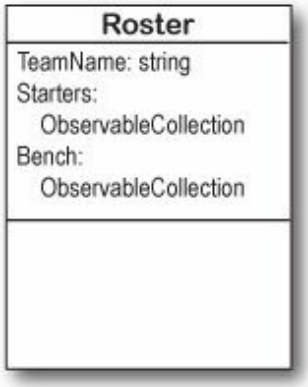
If you model your data like this, it limits your ability to build the pages that you want.

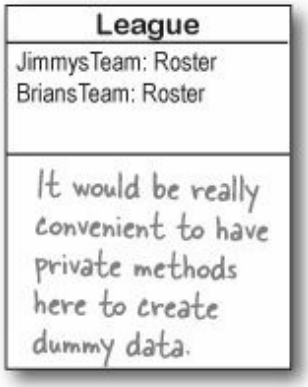


If you model your data like this, it's easier to build your pages but harder to write code to query and manage the data.

...if you also need to be able to bind that data to the XAML controls on your app's pages.







MVVM lets you design for binding and data

Almost all apps that have a large or complex enough object model face the problem of having to either compromise the class design or compromise the objects available for binding. Luckily, there's a design pattern that app developers use to solve this problem. It's called **Model-View-ViewModel** (or **MVVM**), and it works by splitting your app into three layers: the ***Model*** to hold the data and state of the app, the ***View*** that contains the pages and controls that the user interacts with, and the ***ViewModel*** that converts the data in the Model into objects that can be bound and listens for events in the View that the model needs to know about.

> **NOTE**
>
> MVVM is a pattern that uses the existing tools you already have, just like the callback and Observer patterns in the last chapter.

Any object that the user directly interacts with goes in the View.

That includes pages, buttons, text, grids, StackPanels, ListViews, and any other controls that can be laid out using XAML. The controls are bound to objects in the ViewModel, and the controls' event handlers call methods in the ViewModel objects.

The ViewModel has the properties that can bind to the controls in the View.

The properties in the view get their data from the objects in the Model, convert that data into a form that the View's controls can understand, and notify the View when the data changes.

All of the objects that hold the state of the app live in the Model.

This is where your app keeps its data. The ViewModel calls the properties and methods in the Model. If there are objects that change over the course of the app's lifetime, or if data needs to be saved or loaded from files, those things go here.

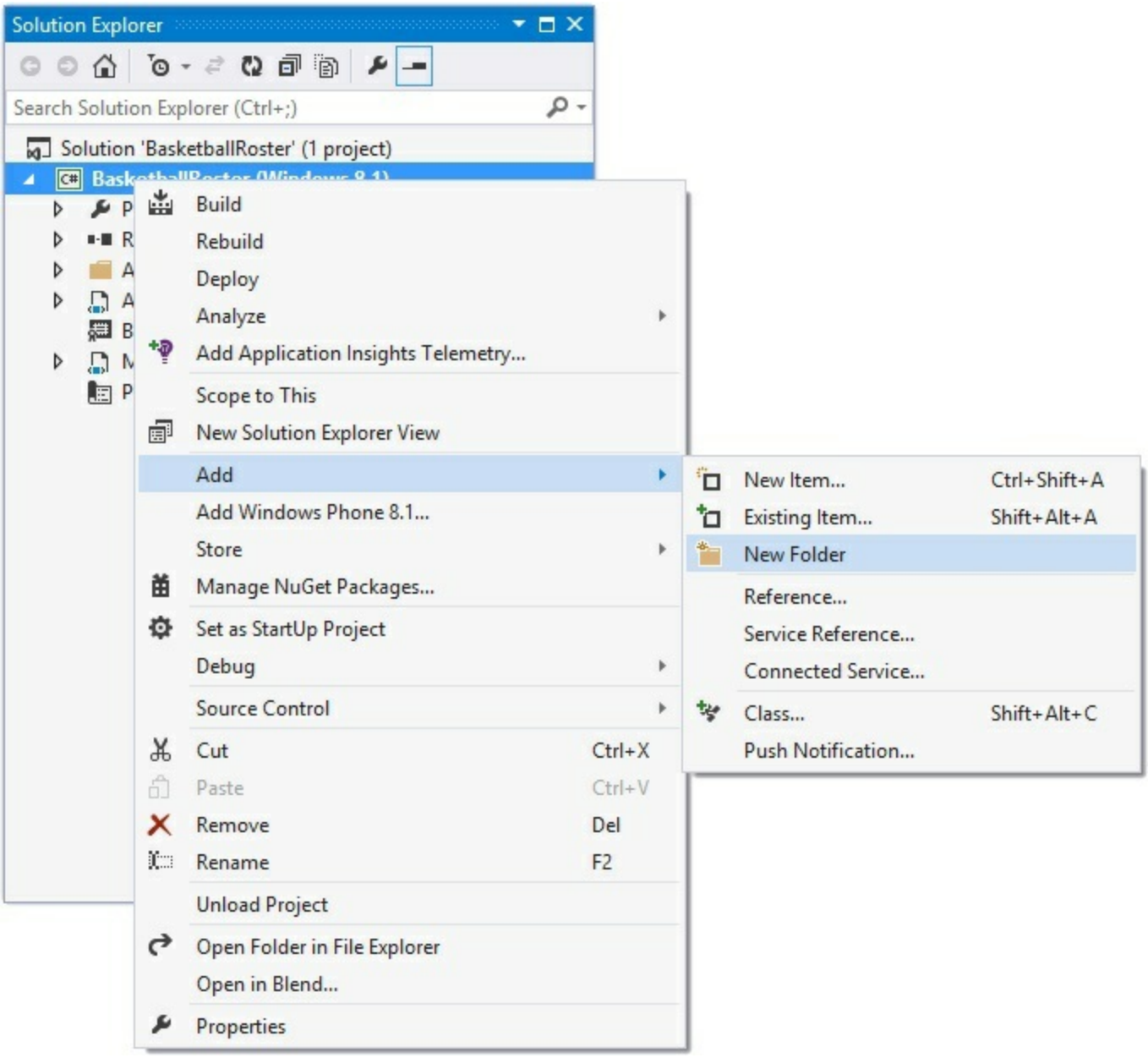
> **The ViewModel is like the plumbing that connects the objects in the View to the objects in the Model, using tools you already know how to work with.**

Use the MVVM pattern to start building the basketball roster app

Create a new Windows Store app and **make sure it's called BasketballRoster** (because we'll be using the namespace `BasketballRoster` in the code, and this will make sure your code matches what's on the next few pages).

DO THIS

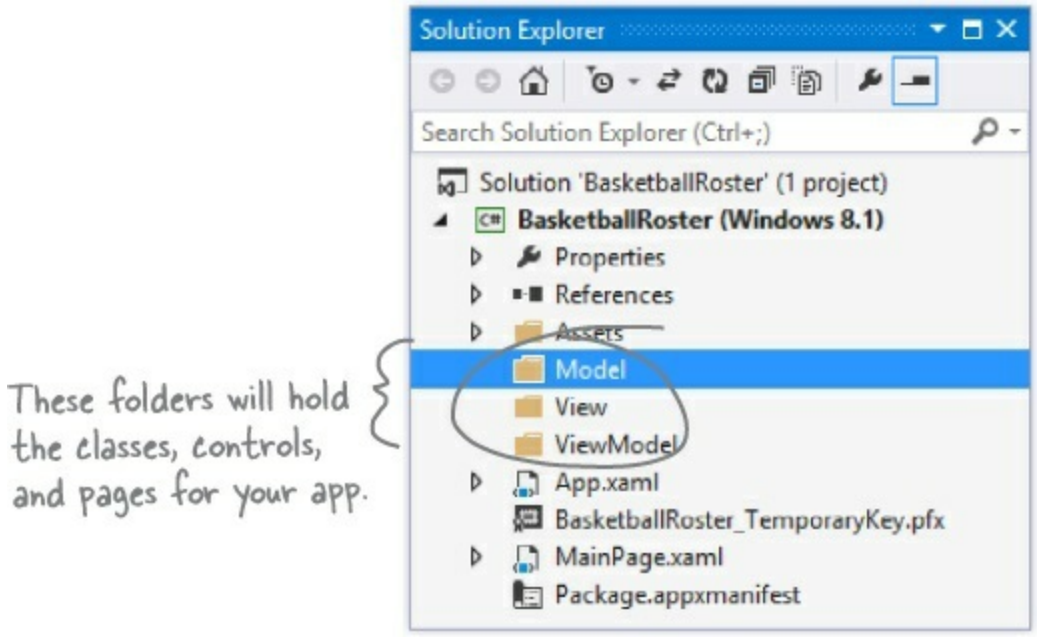
- **❶ CREATE THE MODEL, VIEW, AND VIEWMODEL FOLDERS IN THE PROJECT.**
  Right-click on the project in the Solution Explorer and choose New Folder from the Add menu:



Don't replace *MainPage.xaml* with a Basic Page yet. You'll do that in step #4.

> **When you use the Solution Explorer to add a new folder to your project, the IDE creates a new namespace based on the folder name. This causes the Add→Class... menu option to create classes with that namespace.So if you add a class to the Model folder, the IDE will add `BasketballRoster.Model` to the `namespace` line at the top of the class file.**

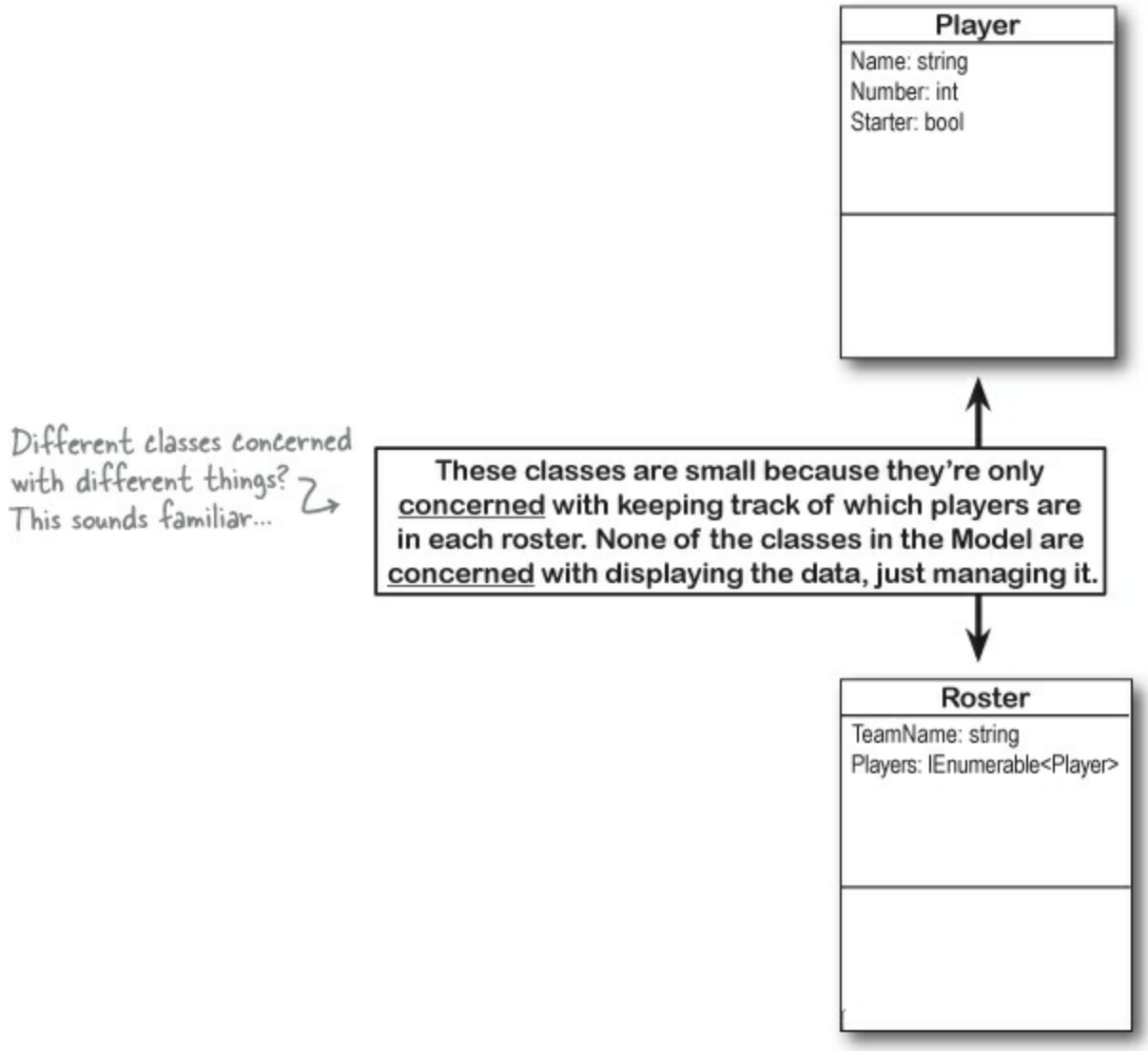
Add a *Model* folder. Then do it two more times to add the *View* and *ViewModel* folders, so your project looks like this:



- **❷ START BUILDING THE MODEL BY ADDING THE PLAYER CLASS.**
  Right-click on the *Model* folder and **add a class called `Player`**. When you add a class into a folder, the IDE updates the namespace to add the folder name to the end. Here's the `Player` class:

When you add a class file into a folder, the IDE adds the folder name to the namespace.

```
namespace BasketballRoster.Model {
    class Player {
        public string Name { get; private set; }
        public int Number { get;  private set; }
        public bool Starter { get; private set; }

        public Player(string name, int number, bool starter) {
            Name = name;
            Number = number;
            Starter = starter;
        }
    }
}
```



- **❸ FINISH THE MODEL BY ADDING THE ROSTER CLASS**
  Next, **add the `Roster` class to the *Model* folder**. Here's the code for it.

```
namespace BasketballRoster.Model {
    class Roster {
        public string TeamName { get; private set; }

        private readonly List<Player> _players = new List<Player>();
        public IEnumerable<Player> Players {
            get { return new List<Player>(_players); }
        }

        public Roster(string teamName, IEnumerable<Player> players)
            TeamName = teamName;
            _players.AddRange(players);
        }
    }
}
```

The _ tells you that this field is private.

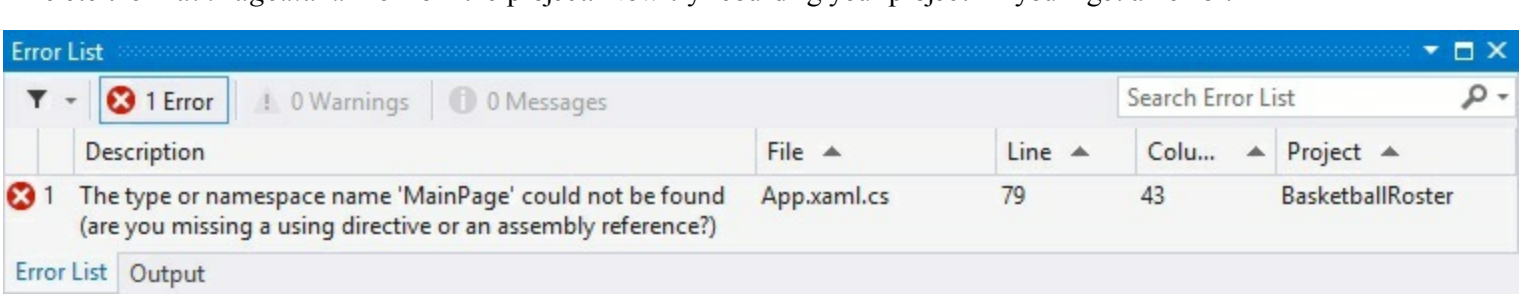
We added an underscore to the beginning of the name of the `_players` field. Adding an underscore to the beginning of private fields is a very common naming convention. We're going to use it throughout this chapter so you can get used to seeing it.

Your *Model* folder should now look like this:

We'll add the view on the next page →

- **❹ ADD THE MAIN PAGE TO THE VIEW FOLDER.**
  Right-click on the *View* folder and **add a new Basic Page called** *LeaguePage.xaml*. You'll be prompted to add missing pages and will need to rebuild the solution, just like when you replace *MainPage.xaml* with a new Basic Page. Edit the XAML and give the page the title "Head First Basketball Conference" by changing the `AppName` static resource (as usual). We're not going to use *MainPage.xaml*, so you'll delete it in the next step.

- **❺ DELETE THE MAIN PAGE AND REPLACE IT WITH YOUR NEW LEAGUEPAGE.XAML PAGE.**
  Delete the *MainPage.xaml* file from the project. Now try rebuilding your project — you'll get an error:



Double-click on the error to jump to the line that broke when you deleted *MainPage.xaml*:

```
// When the navigation stack isn't restored navigate to the first page,
// configuring the new page by passing required information as a navigation
// parameter
rootFrame.Navigate(typeof(MainPage), e.Arguments);
```

Wait a minute, you know what that code does! You modified it when you built the app for Jimmy. It's looking for a `MainPage` class to navigate to when the app launches, but you just deleted the XAML file that defines that class. No problem! Just specify the class that you want to launch:

```
// When the navigation stack isn't restored navigate to the first page,
// configuring the new page by passing required information as a navigation
// parameter
rootFrame.Navigate(typeof(LeaguePage), e.Arguments);
```

Hmm, that's strange. You added the `LeaguePage` to the project, but it's not being recognized. That's because you added it to a folder, so **the IDE added it to the View namespace**. So all you need to do is specify the namespace when you refer to the class:

```
// When the navigation stack isn't restored navigate to the first page,
// configuring the new page by passing required information as a navigation
// parameter
rootFrame.Navigate(typeof(View.LeaguePage), e.Arguments);
```

Now try rebuilding your app. It compiles! You can run it to see your new main page show up.

User controls let you create your own controls

Take a look at the basketball roster program that you're building. Each team gets an identical set of controls: a TextBlock, another TextBlock, a ListView, another TextBlock, and another ListView, all wrapped up by a StackPanel inside a Border. Do we really need to add two identical sets of controls to the page? What if we want to add a third and fourth team — that's going to mean a whole lot of duplication. And that's where **user controls** come in. A user control is a class that you can use to create your own controls. You use XAML and code-behind to build a user control, just like you do when you build a page. Let's get started and add a user control to your BasketballRoster project.

- **❶ Add a new user control to your View folder.**
  Right-click on the *View* folder and add a new item. Choose from the dialog and call it *RosterControl.xaml*.
- **❷ Look at the code-behind for the new user control.**
  Open up *RosterControl.xaml.cs*. Your new control extends the `UserControl` base class. Any code-behind that defines the user control's behavior goes here.

```
namespace BasketballRoster.View
{
    public sealed partial class RosterControl : UserControl
    {
        public RosterControl()
        {
            this.InitializeComponent();
        }
    }
}
```

- **❸ Look at the XAML for the new user control.**
  The IDE added a user control with an empty `<Grid>`. Your XAML will go here.

UserControl is a base class that gives you a way to encapsulate controls that are related to each other, and lets you build logic that defines the behavior of the control.

Before you flip the page, see if you can figure out what XAML should go into the new `RosterControl` by looking at the screenshot of the program that you're building.

- It will have a `<StackPanel>` to stack up the controls that live inside a blue `<Border>`. Can you figure out which property gives a Border control rounded corners?
- It has two ListView controls that display data for players, so it also needs a `<UserControl.Resources>` section that contains a `DataTemplate`. We called it `PlayerItemTemplate`.
- Bind the ListView items to properties called `Starters` and `Bench`, and the top TextBlock to a property called `TeamName`.
- The Border control lives inside a `<Grid>` with a single row that has `Height="Auto"` to keep it from expanding past the bottom of the ListView controls to fill up the entire page.

"TEACH A MAN TO FISH..."
We're nearing the end of the book, so we want to challenge you with problems that are similar to ones you'll face in the real world. A good programmer takes a lot of educated guesses, so we're giving you barely enough information about how a `UserControl` works. You don't even have binding set up, so you won't see data in the designer! How much of the XAML can you build before you flip the page to see the code for `RosterControl`?

- **❹ Finish the RosterControl XAML.**
  Here's the code for the `RosterControl` user control that you added to the *View* folder. Did you notice how we gave you properties for binding, but no data context? That should make sense. The two controls on the page show different data, so the page will set different data contexts for each of them.

```
<UserControl
    x:Class="BasketballRoster.View.RosterControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BasketballRoster.View"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300"
    d:DesignWidth="400">

    <UserControl.Resources>
        <DataTemplate x:Key="PlayerItemTemplate">
            <TextBlock Style="{StaticResource CaptionTextBlockStyle}">
                <Run Text="{Binding Name}"/>
                <Run Text=" #"/>
                <Run Text="{Binding Number}"/>
            </TextBlock>
        </DataTemplate>
    </UserControl.Resources>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <Border BorderThickness="2" BorderBrush="Blue" CornerRadius="6" Margin="0,0,40,0">
            <StackPanel Margin="20">
                <TextBlock Text="{Binding TeamName}"
                           Style="{StaticResource HeaderTextBlockStyle}"/>
                <TextBlock Text="Starting Players"
                           Style="{StaticResource SubheaderTextBlockStyle}" Margin="0,20,0,0"/>
                <ListView ItemsSource="{Binding Starters}"
                          ItemTemplate="{StaticResource PlayerItemTemplate}" Margin="0,20,0,0"/>
                <TextBlock Text="Bench Players"
                           Style="{StaticResource SubheaderTextBlockStyle}" Margin="0,20,0,0"/>
                <ListView ItemsSource="{Binding Bench}"
                          ItemTemplate="{StaticResource PlayerItemTemplate}" Margin="0,20,0,0"/>
            </StackPanel>
        </Border>
    </Grid>
</UserControl>
```

You already know that controls change size based on their Height and Width properties. You can change these numbers to alter how the control is displayed in the IDE's Designer window when you're modifying it.

Here's the template for the items in the ListView controls. Each line has a single TextBlock with three runs to display the player's name and number.

You can use the CornerRadius property to give a Border rounded corners.

Both ListView controls use the same template defined as a static resource.

EXERCISE

Build the ViewModel for the BasketballRoster app by looking at the data in the Model and the bindings in the View, and figuring out what "plumbing" the app needs to connect them together.

- **❶ UPDATE LEAGUEPAGE.XAML TO ADD THE ROSTER CONTROLS.**
  First add these `xmlns` properties to the page so it recognizes the new namespaces:

```
xmlns:view="using:BasketballRoster.View"
xmlns:viewmodel="using:BasketballRoster.ViewModel"
```

  Then add an instance of `LeagueViewModel` as a static resource:

```
<Page.Resources>
    <viewmodel:LeagueViewModel x:Name="LeagueViewModel"/>
    <x:String x:Key="AppName">Head First Basketball Conference</x:String>
</Page.Resources>
```

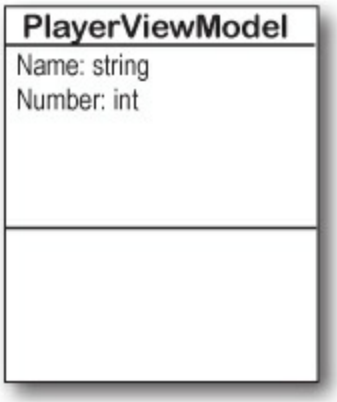
  Now you can add a StackPanel with two `RosterControls` to the page:

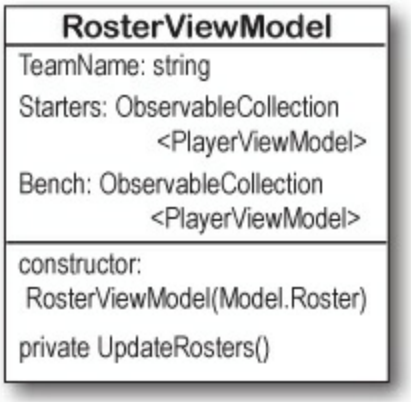
```
<StackPanel Orientation="Horizontal" Margin="120,0,0,0" Grid.Row="1"
DataContext="{StaticResource ResourceKey=LeagueViewModel}" >
    <view:RosterControl DataContext="{Binding JimmysTeam}" Margin="0,0,20,0"/>
    <view:RosterControl DataContext="{Binding BriansTeam}" Margin="0,0,20,0"/>
</StackPanel>
```

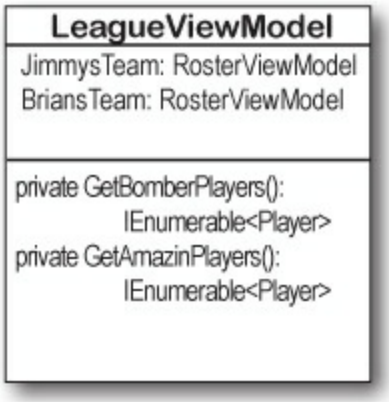
- **❷ CREATE THE VIEWMODEL CLASSES.**

> **NOTE**
>
> Make sure you created the classes and pages in the right folders; otherwise, the namespaces won't match the code in the solution.

  Create these three classes in the *ViewModel* folder.







- **❸ MAKE THE VIEWMODEL CLASSES WORK.**

  The `PlayerViewModel` class is a simple data object with two read-only properties.
  The `LeagueViewModel` class has two private methods to create dummy data for the page. It creates `Model.Roster` objects for each team that get passed to the RosterViewModel constructor.
  The `RosterViewModel` class has a constructor that takes a `Model.Roster` object. It sets the `TeamName` property, and then it calls its private `UpdateRosters()` method, which uses LINQ queries to extract the starting and bench players and update the `Starters`

and `Bench` properties. Add **`using Model;`** to the top of the classes so you can use objects in the `Model` namespace.

NOTE

— Flip a few pages back for a hint about the LINQ query...

If the IDE gives you an error message in the XAML designer that `LeagueViewModel` does not exist in the `ViewModel` namespace, but you're 100% certain you added it correctly, try right-clicking on the BasketballRoster project and choosing Unload Project, and then right-click again and choose Reload Project to reload it.

EXERCISE SOLUTION

The ViewModel for the BasketballRoster app has three classes: `LeagueViewModel`, `PlayerViewModel`, and `RosterViewModel`. They all live in the *ViewModel* folder.

If you left out the `using Model;` line then you'd have to use `Model.Roster` instead of `Roster` everywhere.

```
namespace BasketballRoster.ViewModel {
    using Model;
    using System.Collections.ObjectModel;

    class LeagueViewModel {
        public RosterViewModel BriansTeam { get; private set; }
        public RosterViewModel JimmysTeam { get; private set; }

        public LeagueViewModel() {
            Roster briansRoster = new Roster("The Bombers", GetBomberPlayers());
            BriansTeam = new RosterViewModel(briansRoster);

            Roster jimmysRoster= new Roster("The Amazins", GetAmazinPlayers());
            JimmysTeam = new RosterViewModel(jimmysRoster);
        }

        private IEnumerable<Player> GetBomberPlayers() {
            List<Player> bomberPlayers = new List<Player>() {
                new Player("Brian", 31, true),
                new Player("Lloyd", 23, true),
                new Player("Kathleen",6, true),
                new Player("Mike", 0, true),
                new Player("Joe", 42, true),
                new Player("Herb",32, false),
                new Player("Fingers",8, false),
            };
            return bomberPlayers;
        }

        private IEnumerable<Player> GetAmazinPlayers()  {
            List<Player> amazinPlayers = new List<Player>() {
                new Player("Jimmy",42, true),
                new Player("Henry",11, true),
                new Player("Bob",4, true),
                new Player("Lucinda", 18, true),
                new Player("Kim", 16, true),
                new Player("Bertha", 23, false),
                new Player("Ed",21,  false),
            };
            return amazinPlayers;
        }
    }
}
```

LeagueViewModel exposes RosterViewModel objects that a RosterControl can use as its data context. It creates the Roster model object for the RosterViewModel to use.

This private method generates dummy data for the Bombers by creating a new List of Player objects.

Dummy data typically goes in the ViewModel because the state of an MVVM application is managed using instances of the Model classes that are encapsulated inside the ViewModel objects.

You use classes from the View to store your data, which is why this method returns Player objects and not PlayerViewModel objects.

Here's the PlayerViewModel. It's just a simple data object with properties for the data template to bind to.

```
namespace BasketballRoster.ViewModel {
    class PlayerViewModel {
        public string Name { get; private set; }
        public int Number { get; private set; }

        public PlayerViewModel(string name, int number) {
            Name = name;
            Number = number;
        }
    }
}
```

```
namespace BasketballRoster.ViewModel {
    using Model;
    using System.Collections.ObjectModel;
    using System.ComponentModel;

    class RosterViewModel {
        public ObservableCollection<PlayerViewModel> Starters { get; private set; }
        public ObservableCollection<PlayerViewModel> Bench { get; private set; }

        private Roster _roster;

        private string _teamName;
        public string TeamName {
            get { return _teamName; }
            set {
                _teamName = value;
            }
        }

        public RosterViewModel(Roster roster) {
            _roster = roster;

            Starters = new ObservableCollection<PlayerViewModel>();
            Bench = new ObservableCollection<PlayerViewModel>();

            TeamName = _roster.TeamName;

            UpdateRosters();
        }

        private void UpdateRosters() {
            var startingPlayers =
                from player in _roster.Players
                where player.Starter
                select player;

            foreach (Player player in startingPlayers)
                Starters.Add(new PlayerViewModel(player.Name, player.Number));

            var benchPlayers =
                from player in _roster.Players
                where player.Starter == false
                select player;

            foreach (Player player in benchPlayers)
                Bench.Add(new PlayerViewModel(player.Name, player.Number));
        }
    }
}
```

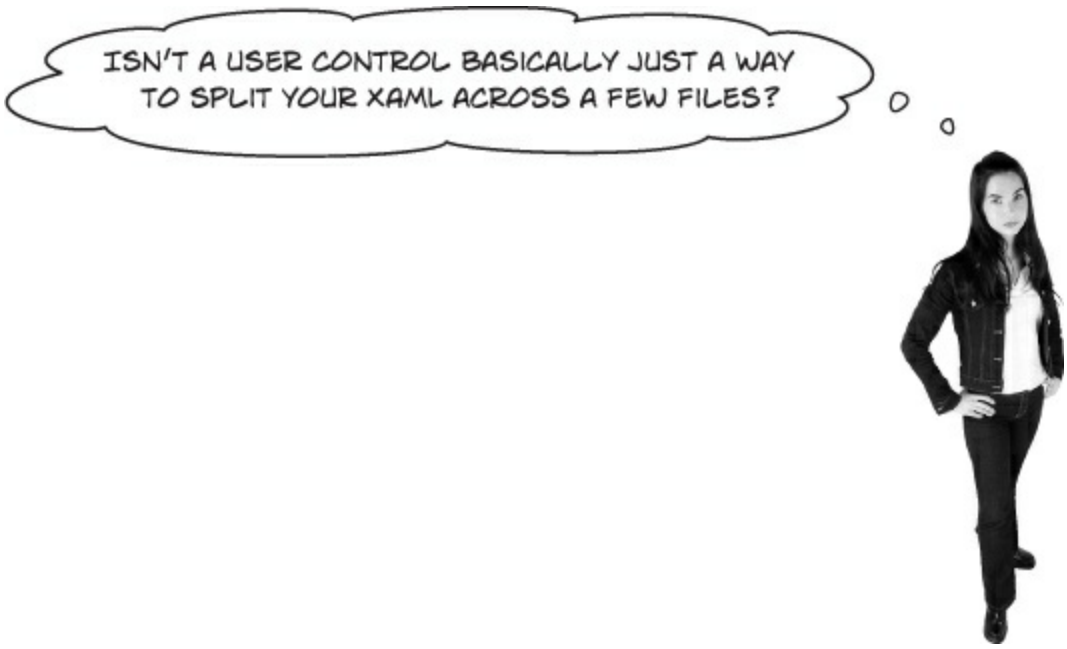
This is where the app stores its state—in Roster objects encapsulated inside the ViewModel. The rest of the class translates the Model data into properties that the View can bind to.

Whenever the TeamName property changes, the RosterViewModel fires off a PropertyChanged event so any object bound to it will get updated.

This LINQ query finds all the starting players and adds them to the Starters ObservableCollection property.

Here's a similar LINQ query to find the bench players.

In a typical MVVM app, only classes in the ViewModel implement `INotifyPropertyChanged`. That's because the ViewModel contains the only objects that XAML controls are bound to. In this project, however, we didn't need to implement INotifyPropertyChanged because the bound properties are updated in the constructor. If you wanted to modify the project to let the Brian and Jimmy change their team names, you'd need to fire a PropertyChanged event in the TeamName set accessor.



User controls are fully functional controls that you build.

And like every other control, a user control is an object — in this case, an object that extends the `UserControl` base class, which gives you familiar properties like `Height` and `Visibility`, and routed events like `Tapped` and `PointerEntered`. You can also add your own properties, and you can use the other XAML controls to make very intricate, even visually stunning, user interfaces. But most importantly, a user control lets you **encapsulate** those other controls into a single XAML control that you can reuse.

That's right! The Model, View, and ViewModel divide up the concerns of the program.

One of the most challenging parts of designing a large, robust app is choosing which objects do what. There are an almost infinite number of ways to design your app. That's great, because it means that C# gives you flexible tools to work with. But it's also a challenge, because today's decisions can make tomorrow's changes very difficult to manage. MVVM helps you separate the concerns about the data in your app from the concerns about its UI. This makes it easier to design your app by helping

you figure out exactly where data goes and where UI elements go, and by giving you patterns to help connect them together.

NOTE

When a change to one class requires changes to two more, which then require more changes to additional classes, there's a name for that. Programmers call it "shotgun surgery," and it's very frustrating — especially when you're in a hurry. Separation of concerns is a great way to prevent problems like that, and MVVM is a very useful tool to help you separate some important things that almost every app is concerned with.

THERE ARE NO DUMB QUESTIONS

Q: **Q: So what's stopping me from putting controls in the ViewModel or `ObservableCollections` in the Model?**

A: **A:** Nothing at all — except that once you do, you're no longer using the MVVM pattern. Classes like controls and pages are concerned with displaying the data. If you put them in the View, that makes it easier for you to manage your codebase as your app grows larger. When you trust the MVVM pattern today, your life is better tomorrow because your code is easier to manage.

Q: **Q: I still don't get what *state* means.**

A: **A:** When people talk about *state* they mean the objects in memory that determine how your app functions: the text in a text editor, the location of the enemies and player and the score in a video game, the values of the cells in a spreadsheet. This is actually a tough concept to wrap your brain around, because it's sometimes difficult to say "this object is part of the state" and "that object isn't." One of the goals of the next project in this chapter is to help you get a practical, realistic handle on what *state* really means.

Q: **Q: Why do I need `using Model;` at the top of my ViewModel classes?**

A: **A:** When you created classes in the *Model* folder, the IDE automatically created them in the `BasketballRoster.Model` namespace. The dot in the middle of that namespace means that Model is underneath BasketballRoster. Any other class in a namespace under BasketballRoster can access classes in Model by either adding `Model.` to the beginning or adding a using line. Outside the `BasketballRoster` namespace, classes will need to add `using BasketballRoster.Model;` instead.

Q: **Q: I keep seeing a triangle with an exclamation point on my page. What's that about?**

A: **A:** The IDE's XAML designer is a pretty sophisticated piece of machinery. It works so well that we sometimes forget just how much work it has to do to display a page and update it as we modify the XAML. Now that the BasketballRoster program is finished, the designer shows you the dummy data for both teams. But wait a minute — isn't that dummy data created in private methods in the ViewModel? That means the designer must be running those methods every time it updates the page. So in order for it to be able to run properly, those methods have to be compiled. If you modify the controls that are on the page, then the latest C# code hasn't been compiled yet, so the designer is telling you that the page that it's displaying may be out of date. Rebuild the code and the exclamation points usually disappear.

Q: **Q: The BasketballRoster app I just built only has dummy data that's created when it starts up. What if I want to add a feature to modify the data in the Model — how would that work?**

A: **A:** Let's say you wanted to modify your BasketballRoster program to let Jimmy and Brian trade players. You already know that the ListView controls in the View are bound to `ObservableCollection` objects, so the ViewModel communicates with the View using `PropertyChanged` and `CollectionChanged` events. And you can have the Model communicate with the ViewModel in exactly the same way. You could add an event, `RosterUpdated`, to the `Roster` object. The `RosterViewModel` would listen to that event, and its event handler would refresh the `Starters` and `Bench` collections, which would then fire off `CollectionChanged` events, which would update the ListView controls.

Events are a good way for the Model to communicate to the rest of the app because the Model doesn't need to know if any other classes are listening to the event. It can go about its business managing the state, and let some other class worry about getting input and updating the user interface because it's ***decoupled*** from the classes in the ViewModel and the View.

When you trust the MVVM pattern today, your project will be better tomorrow, because your app's code will be easier to manage.

The Model-View-ViewModel design pattern is actually adapted from another pattern called Model-View-Controller. You can learn all about the MVC pattern in the GDI+ PDF, which you can download from the Head First Labs website.

FIRESIDE CHATS

Tonight's talk: **A Model and a ViewModel have a heated debate over the critical issue of the day, "Who's needed more?"**

Model	**ViewModel**
I'm not quite sure why we're even having this discussion. Where would you be without me? I've got the data; I've got the important logic that determines how the app works. Without me, you'd have nothing to do.	
	There you go again, thinking that you're the center of the universe.
Well, as far as you're concerned, I may as well be.	
	Ha! What would happen if I decided to stay home?
You wouldn't dare.	
	Try me! Without me, you'd be useless. The View would have no idea how to talk to you. The controls would be empty, and the user would be left in the dark.
Now you know why I only speak to you through events. You're just so annoying!	
	You know what? Let's talk about that for a minute. Why is it that you can't even let me see your internals? You only expose methods and properties to me, and you'll only ever send me messages through event arguments.
Of course I do! If I didn't encapsulate my data, who knows what damage you might cause?	
	It sounds like ***someone*** has trust issues.
Absolutely! I don't trust anything except my own private methods to manage my data; otherwise, the whole state of our app could go haywire. But I'm not the only one who plays this game! Why don't you ever let me talk to the View? He seems like a good guy.	
	You barely even speak the same language as the View! I've never seen you fire a `PropertyChanged` event — in fact, I don't think any of your objects even implement `INotifyPropertyChanged`.
How dare you! Raise `PropertyChanged` events? No self-respecting Model has ever raised a `PropertyChanged` event! I'm insulted you'd even suggest I'm concerned with anything but data. What kind of layer do you think I am?	

The ref needs a stopwatch

Jimmy and Brian had to call off their last game because the referee forgot his stopwatch. Can we use the MVVM pattern to build a stopwatch app for them?

How's the ref going to enforce the three-second rule without it?

MVVM means thinking about the state of the app

MVVM apps use the Model and View to separate the state from the user interface. So when you start building an MVVM app, the first thing you usually do is think about exactly what it means to manage the state of the app. Once you've got the state under control in your brain, you can start building the Model, which will use fields and properties to keep track of the state — or everything the app needs to keep track of to do its job. Most apps need to modify the state as well, so the Model exposes public methods that change the state. The rest of the app needs to be able to see the current state, so the Model provides public properties.

So what does it mean to manage the state of a stopwatch?

The stopwatch knows whether or not it's running.

You can see at a glance whether or not the hands are moving, so the stopwatch Model needs to have a way to tell whether or not it's running.

The elapsed time is always available.

Whether it's the hands on an analog stopwatch or numbers on a digital one, you can always see the elapsed time.

The lap time can be set and viewed.

Most stopwatches have a lap time function that lets you save the current time without stopping the clock. Analog stopwatches use an extra set of hands to show the lap time, while digital stopwatches usually have a separate lap time readout.

The stopwatch can stop, start, and reset.

The app will need to provide a way to start the stopwatch, stop it, and reset the time, which means the Model will need to give the rest of the app a way to do this.

The Model keeps track of the state of the app: what the app knows right now. It provides actions that modify the app's state and properties to let the rest of the app see the current state.

Start building the stopwatch app's Model

Now that we know what it means to define the state of a stopwatch, we have enough information to start to build out the Model layer of the stopwatch app. Create a new Windows Store app. **Name the app *Stopwatch*** so your namespaces match the code on the next few pages. Then **create the *Model*, *View*, and *ViewModel* folders**. Add the `StopwatchModel` class to the *Model* folder:

DO THIS

```
class StopwatchModel {
    private DateTime? _started;

    private TimeSpan? _previousElapsedTime;

    public bool Running {
        get { return _started.HasValue; }
    }

    public TimeSpan? Elapsed {
        get {
            if (_started.HasValue) {
                if (_previousElapsedTime.HasValue)
                    return CalculateTimeElapsedSinceStarted() + _previousElapsedTime;
                else
                    return CalculateTimeElapsedSinceStarted();
            }
            else
                return _previousElapsedTime;
        }
    }

    private TimeSpan CalculateTimeElapsedSinceStarted() {
        return DateTime.Now - _started.Value;
    }

    public void Start() {
        _started = DateTime.Now;
        if (!_previousElapsedTime.HasValue)
            _previousElapsedTime = new TimeSpan(0);
    }

    public void Stop() {
        if (_started.HasValue) {
            _previousElapsedTime += DateTime.Now - _started.Value;
            _started = null;
        }
    }

    public void Reset() {
        _previousElapsedTime = null;
        _started = null;
    }

    public StopwatchModel() {
        Reset();
    }
}
```

Make sure you create the Stopwatch class in the Model folder. We'll leave out the extra namespace { } lines because you know what they look like.

These two private fields hold the state of the stopwatch. They're both nullable.

We could use an extra Boolean field to keep track of whether or not the stopwatch is running. But that field would only be true if the _started field has a value, so why not just use _started.HasValue instead?

This read-only property uses the two private fields to calculate the elapsed time. Look closely at it. Can you figure out how it works?

Here's a hint: when you add or subtract DateTime or TimeSpan values, you always get a TimeSpan.

The rest of the app needs to be able to start and stop the stopwatch, so the Model provides methods to do that.

Resetting the state means setting its fields to null.

This initializes each new instance of a StopwatchModel as reset and stopped.

TIMESPAN AND DATETIME STRUCTS

There are two very useful structs for managing time in an app. You've already worked with DateTime, which stores a date. TimeSpan represents an interval of time. The interval is stored in ticks (a tick is one ten-millionth of a second, or 10,000 ticks per millisecond), so the TimeSpan has methods to convert it to seconds, milliseconds, days, etc.

Events alert the rest of the app to state changes

The stopwatch needs to track the lap time, so it needs to store that time as part of the state. It also needs a method to get the lap time. But what happens if we want the rest of the app to do a few things when the lap time is triggered? The ViewModel may want to turn on an indicator or show a quick animation. The Model will often **use an event to tell the rest of the app about important state changes**. So let's add an event to the Model that gets fired whenever the lap time is updated. Start by adding the `LapEventArgs` to the *Model* folder:

Here's the LapEventArgs class. Make sure to add it to the Model folder so it ends up in the correct namespace.

```
class LapEventArgs : EventArgs {

    public TimeSpan? LapTime { get; private set; }

    public LapEventArgs(TimeSpan? lapTime) {

        LapTime = lapTime;

    }

}
```

When the lap time is updated, the app needs to know the time elapsed, so it has a TimeSpan property to store that.

Modify your `StopwatchModel` class to add a `Lap()` method that sets the `LapTime` property and fires a `LapTimeUpdated` event.

```
public void Reset() {
    _previousElapsedTime = null;
    _started = null;
    LapTime = null;
}

public TimeSpan? LapTime { get; private set; }

public void Lap() {
    LapTime = Elapsed;
    OnLapTimeUpdated(LapTime);
}

public event EventHandler<LapEventArgs> LapTimeUpdated;

private void OnLapTimeUpdated(TimeSpan? lapTime) {
    EventHandler<LapEventArgs> lapTimeUpdated = LapTimeUpdated;
    if (lapTimeUpdated != null) {
        lapTimeUpdated(this, new LapEventArgs(lapTime));
    }
}
```

Make sure the LapTime property is reset when the rest of the stopwatch state gets reset.

The Lap() method updates the property and fires the event.

An automatic property will be just fine. We don't need a private backing field because there aren't any calculations that need to be encapsulated.

This is just the usual code to fire an event.

The Model can fire an event to tell the rest of the app about important state changes without any references to classes outside the Model. It's easier to build because it's decoupled from the rest of the MVVM layers.

A nice side effect of decoupled layers is that your project can build as soon as the Model is complete.

If the IDE tells you that `StopwatchViewModel` does not exist in the `ViewModel` namespace but you're 100% sure that you put it there, try unloading and reloading the project.

Build the view for a simple stopwatch

Here's the XAML for a simple stopwatch control. **Add a user control to the *View* folder called *BasicStopwatch.xaml*** and add this code. The control has TextBlock controls to display the elapsed and lap times, and buttons to start, stop, reset, and take the lap time.

```
<UserControl
    x:Class="Stopwatch.View.BasicStopwatch"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Stopwatch.View"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300"
    d:DesignWidth="400"
    xmlns:viewmodel="using:Stopwatch.ViewModel">

    <UserControl.Resources>
        <viewmodel:StopwatchViewModel x:Name="viewModel"/>
    </UserControl.Resources>

    <Grid DataContext="{StaticResource ResourceKey=viewModel}">
        <StackPanel>
            <TextBlock>
                <Run>Elapsed time: </Run>
                <Run Text="{Binding Hours}"/>
                <Run>:</Run>
                <Run Text="{Binding Minutes}"/>
                <Run>:</Run>
                <Run Text="{Binding Seconds}"/>
            </TextBlock>
            <TextBlock>
                <Run>Lap time: </Run>
                <Run Text="{Binding LapHours}"/>
                <Run>:</Run>
                <Run Text="{Binding LapMinutes}"/>
                <Run>:</Run>
                <Run Text="{Binding LapSeconds}"/>
            </TextBlock>
            <StackPanel Orientation="Horizontal">
                <Button Click="StartButton_Click">Start</Button>
                <Button Click="StopButton_Click">Stop</Button>
                <Button Click="ResetButton_Click">Reset</Button>
                <Button Click="LapButton_Click">Lap</Button>
            </StackPanel>
        </StackPanel>
    </Grid>
</UserControl>
```

This control is in the View folder under your project's main namespace.

You'll need this xmlns property to add the namespace. We called our project Stopwatch, so the ViewModel namespace is Stopwatch.ViewModel.

This user control stores an instance of the ViewModel as a static resource and uses it as its data context. It doesn't need its container to set a data context. It keeps track of its own state.

This TextBlock is bound to properties in the ViewModel that return the elapsed time.

This TextBlock is bound to properties that expose the lap time.

The ViewModel must be firing off PropertyChanged events to keep these values up to date.

You'll need to add `Click` event handlers to the control and a `StopwatchViewModel` class to the `ViewModel` namespace for this to compile.

NOTE

Here's a hint: use a DispatcherTimer to constantly check the Model and update the properties.

The code for the ViewModel is on the next page. How much of the ViewModel code can you build just from the View and Model code before you flip the page? Add a BasicStopwatch control to the main page *(for now)* and see how far you can get.

But be really careful and don't assume the IDE is necessarily wrong. Sometimes an error in the XAML for one page (like a broken `xmlns` property) can cause all of the designers to break.

Add the stopwatch ViewModel

Here's the ViewModel for the stopwatch. Make sure it goes in the `ViewModel` namespace.

```
class StopwatchViewModel : INotifyPropertyChanged {
    private StopwatchModel _stopwatchModel = new StopwatchModel();

    private DispatcherTimer _timer = new DispatcherTimer();

    public bool Running { get { return _stopwatchModel.Running; } }

    public StopwatchViewModel() {
        _timer.Interval = TimeSpan.FromMilliseconds(50);
        _timer.Tick += TimerTick;
        _timer.Start();
        Start();

        _stopwatchModel.LapTimeUpdated += LapTimeUpdatedEventHandler;
    }

    public void Start() {
        _stopwatchModel.Start();
    }

    public void Stop() {
        _stopwatchModel.Stop();
    }

    public void Lap() {
        _stopwatchModel.Lap();
    }

    public void Reset() {
        bool running = Running;
        _stopwatchModel.Reset();
        if (running)
            _stopwatchModel.Start();
        OnPropertyChanged("LapHours");
        OnPropertyChanged("LapMinutes");
        OnPropertyChanged("LapSeconds");
    }

    int _lastHours;
    int _lastMinutes;
    decimal _lastSeconds;
    void TimerTick(object sender, object e) {
        if (_lastHours != Hours) {
            _lastHours = Hours;
            OnPropertyChanged("Hours");
        }
        if (_lastMinutes != Minutes) {
            _lastMinutes = Minutes;
            OnPropertyChanged("Minutes");
        }
        if (_lastSeconds != Seconds) {
            _lastSeconds = Seconds;
            OnPropertyChanged("Seconds");
        }
    }

    public int Hours {
        get { return _stopwatchModel.Elapsed.HasValue ? _stopwatchModel.Elapsed.Value.Hours : 0; }
    }
```

The Running property checks the Model to see if the stopwatch is running.

You'll need these using statements for the class to compile.

```
using Model;
using System.ComponentModel;
using Windows.UI.Xaml;
```

The Start(), Stop(), and Lap() methods just pass through to methods on the model.

If you don't want your stopwatch to start immediately when you run the app, don't have the ViewModel call Start() from the constructor. That will prevent it from telling the Model to start the clock. But you still want the ViewModel to start its timer. Can you figure out why?

The Reset() method first calls the Model's Reset() method, then calls its Start() method if the stopwatch was already running.

Since the Model's Reset() method resets the lap time, the ViewModel needs to fire PropertyChanged events.

Every time the DispatcherTimer ticks, the ViewModel checks to see if the hours, minutes, and seconds have changed. If they have, it fires off the right PropertyChanged event so the View can update itself.

This ? : syntax lets you fit a conditional on a single line, and it works just like an `if` statement. Flip to leftover #2 in the appendix to learn more about how it works.

```
    public int Minutes {
        get { return _stopwatchModel.Elapsed.HasValue ? _stopwatchModel.Elapsed.Value.Minutes : 0; }
    }

    public decimal Seconds {
        get {
            if (_stopwatchModel.Elapsed.HasValue) {
                return (decimal)_stopwatchModel.Elapsed.Value.Seconds
                    + (_stopwatchModel.Elapsed.Value.Milliseconds * .001M);
            }
            else
                return 0.0M;
        }
    }

    public int LapHours {
        get { return _stopwatchModel.LapTime.HasValue ? _stopwatchModel.LapTime.Value.Hours : 0; }
    }

    public int LapMinutes {
        get { return _stopwatchModel.LapTime.HasValue ? _stopwatchModel.LapTime.Value.Minutes : 0; }
    }

    public decimal LapSeconds {
        get {
            if (_stopwatchModel.LapTime.HasValue) {
                return (decimal)_stopwatchModel.LapTime.Value.Seconds
                    + (_stopwatchModel.LapTime.Value.Milliseconds * .001M);
            }
            else
                return 0.0M;
        }
    }

    int _lastLapHours;
    int _lastLapMinutes;
    decimal _lastLapSeconds;
    private void LapTimeUpdatedEventHandler(object sender, LapEventArgs e) {
        if (_lastLapHours != LapHours) {
            _lastLapHours = LapHours;
            OnPropertyChanged("LapHours");
        }
        if (_lastLapMinutes != LapMinutes) {
            _lastLapMinutes = LapMinutes;
            OnPropertyChanged("LapMinutes");
        }
        if (_lastLapSeconds != LapSeconds) {
            _lastLapSeconds = LapSeconds;
            OnPropertyChanged("LapSeconds");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string propertyName) {
        PropertyChangedEventHandler propertyChanged = PropertyChanged;
        if (propertyChanged != null)
            propertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Elapsed.Value returns a TimeSpan, and its Minutes property returns an int.

The Seconds property returns the seconds plus hundredths of a second as a decimal. Set a breakpoint and use the debugger to explore how it works.

These properties work just like the elapsed time properties, except they're based on LapTime instead of Elapsed.

Here's the event handler for the Model's LapTimeUpdated event. It works just like the DispatcherTimer's event handler by checking the lap time properties and raising PropertyChanged events for only the ones that have changed.

Here's the familiar code for PropertyChanged.

Finish the stopwatch app

There are just a few more loose ends to tie together. Your BasicStopwatch user control doesn't have event handlers, so you need to add them. And then you just need to add the control to your main page.

- ❶ First, **go back to *BasicStopwatch.xaml.cs*** and add these event handlers to the code-behind:

```
private void StartButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Start();
}
private void StopButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Stop();
}
private void ResetButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Reset();
}
private void LapButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Lap();
}
```

The buttons in the view just call methods in the ViewModel. This is a pretty typical pattern for the View.

- ❷ Next, **delete the *MainPage.xaml* file and *replace it* with a Basic Page**, just like you've done in your other projects (don't forget to rebuild the solution).
- ❸ Open the new *MainPage.xaml* and **add the XML namespace** to the top-level tag:

```
xmlns:view="using:Stopwatch.View"
```

- ❹ Modify the `AppName` resource in *MainPage.xaml* to **set the page name**.
- ❺ **Add a BasicStopwatch control to the XAML code** in *MainPage.xaml*:

> **NOTE**
>
> All of the behavior is in the user control, so there's no code-behind for the main page.

```
<view:BasicStopwatch Grid.Row="1" Margin="120,0"/>
```

Your app should now run. Click the Start, Stop, Reset, and Lap buttons to see your stopwatch work. And now you can use the **frame rate counter** to check your app's performance. The numbers in the upper left-hand corner show you the app's frames per second, and the CPU usage for your app's UI (the numbers in the upper right are for the entire system). Try **modifying the StopwatchViewModel constructor** to to change the Timer interval to `TimeSpan.FromMilliseconds(100)`. What happens to the frame rate? What happens if you set it to tick every 10 milliseconds?

Using a pattern like MVVM means making decisions.

MVVM is a pattern, which means there are conventions, but not hard-and-fast rules that can be checked with a compiler. And it's a ***flexible*** pattern, which means that there are a lot of different ways that you can implement it. Throughout the examples in this chapter, we'll show you some of the more common things that you'll see in an app with an MVVM architecture. And where we do vary things, we'll explain why we made those decisions. The goal is to show you how much flexibility is

— *and isn't* — in the MVVM pattern, so that you can **make good decisions** when you build your own apps.

HERE ARE A FEW RULES THAT WE'RE FOLLOWING WHEN BUILDING OUR MVVM APPS:

- The Model, ViewModel, and View classes live in **separate namespaces**.
- Controls and pages in the View can **keep references to the ViewModel**, so they can call its methods and bind to its properties with one- or two-way binding.
- Objects in the ViewModel **don't** store any references to objects in the View.
- If the ViewModel has information to pass to the View, it uses `PropertyChanged` **and** `CollectionChanged` **events** so the bindings can update automatically.
- ViewModel objects have **references to Model objects**, and can call their methods, as well as get and set their properties.
- If the Model has information to pass to the ViewModel, it can **raise an event**.
- Objects in the Model **don't** have references to objects in the ViewModel.
- The Model **must be well encapsulated** so that it only depends on other objects in the Model. If you delete all of the other code in the program, everything in the Model folder should still compile.
- `DispatcherTimers` and asynchronous code typically go in the ViewModel and not the Model. Code related to timing usually drives ***how*** the state of the app changes but is ***not actually part*** of the state of the app most of the time.

Converters automatically convert values for binding

Anyone with a digital clock knows that it typically shows the minutes with a leading zero. Our stopwatch should also show the minutes with two digits. And it should also show the seconds with two digits, and round to the nearest hundredth of a second. We *could* modify the ViewModel to expose string values that are formatted properly, but that would mean that we'd need to keep adding more and more properties each time we wanted to reformat the same data. That's where **value converters** come in very handy. A value converter is an object that the XAML binding uses to modify data before it's passed to the control. You can build a value converter by implementing the `IValueConverter` interface (which is in the `Windows.UI.Xaml.Data` namespace). Add a value converter to your stopwatch now.

- **❶ Add the TimeNumberFormatConverter class to the ViewModel folder.**
  Add **`using Windows.UI.Xaml.Data;`** to the top of the class, then have it implement the **`IValueConverter`** interface. Use the IDE to automatically implement the interface. This will add two method stubs for the `Convert()` and `ConvertBack()` methods.
- **❷ Implement the Convert() method in the value converter.**
  The `Convert()` method takes several parameters — we'll use two of them. The **`value`** parameter is the raw value that's passed into the binding, and **`parameter`** lets you specify a parameter in XAML.

```
using Windows.UI.Xaml.Data;

class TimeNumberFormatConverter : IValueConverter {
    public object Convert(object value, Type targetType,
                          object parameter, string language) {
        if (value is decimal)
            return ((decimal)value).ToString("00.00");
        else if (value is int) {
            if (parameter == null)
                return ((int)value).ToString("d1");
            else
                return ((int)value).ToString(parameter.ToString());
        }
        return value;
    }

    public object ConvertBack(object value, Type targetType,
                              object parameter, string language) {
        throw new NotImplementedException();
    }
}
```

This converter knows how to convert decimal and int values. For int values, you can optionally pass in a parameter.

The ConvertBack() method is used for two-way binding. We're not using that in this project, so you can leave the method stub as is.

Is it a good idea to leave this `NotImplementedException` in your code? For this project, this is code that is never supposed to be run. If it does get run, is it better to fail silently, so the user never sees it? Or is it better to throw an exception so that you can track down the problem? Which of those gives you a more robust app? There's not necessarily one right answer.

- **❸ Add the converter to your stopwatch control as a static resource.**
  It should go right below the `ViewModel` object:

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Name="viewModel"/>
    <viewmodel:TimeNumberFormatConverter x:Name="timeNumberFormatConverter"/>
</UserControl.Resources>
```

The designer may make you rebuild the solution after you add this line. In rare cases, you might even need to unload and reload the project.

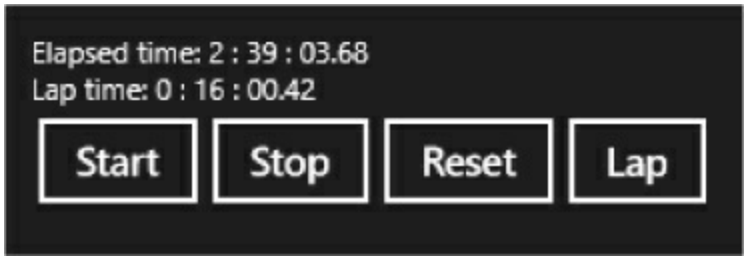
- **❹ Update the XAML code to use the value converter.**
  Modify the `{Binding}` markup by adding the `Converter=` to it in each of the `<Run>` tags.

```
<TextBlock>
    <Run>Elapsed time: </Run>
    <Run Text="{Binding Hours,
        Converter={StaticResource timeNumberFormatConverter}}"/>
    <Run>:</Run>
    <Run Text="{Binding Minutes,
        Converter={StaticResource timeNumberFormatConverter}, ConverterParameter=d2}"/>
    <Run>:</Run>
    <Run Text="{Binding Seconds,
        Converter={StaticResource timeNumberFormatConverter}}"/>
</TextBlock>
<TextBlock>
    <Run>Lap time: </Run>
    <Run Text="{Binding LapHours,
        Converter={StaticResource timeNumberFormatConverter}}"/>
    <Run>:</Run>
    <Run Text="{Binding LapMinutes,
        Converter={StaticResource timeNumberFormatConverter}, ConverterParameter=d2}"/>
    <Run>:</Run>
    <Run Text="{Binding LapSeconds,
        Converter={StaticResource timeNumberFormatConverter}}"/>
</TextBlock>
```

If there's no parameter specified, don't forget the extra closing bracket }}.

Use the ConverterParameter syntax to pass a parameter into the converter.

Now the stopwatch runs the values through the converter before passing them into the TextBlock controls, and the numbers are formatted correctly on the page.



Converters can work with many different types

TextBlock and TextBox controls work with text, so binding strings or numbers to the `Text` property makes sense. But there are many other properties, and you can bind to those as well. If your ViewModel has a Boolean property, it can be bound to any `true`/`false` property. You can even bind properties that use enums — the `IsVisible` property uses the `Visibility` enum, which means you can also write value converters for it. Let's add Boolean and `Visibility` binding and conversion to the stopwatch.

Here are two converters that will come in handy.

Sometimes you want to bind Boolean properties like `IsEnabled` so that a control is enabled if the bound property is `false`. We'll add a new converter called `BooleanNotConverter`, which uses the `!` operator to invert a Boolean target property.

```
IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
```

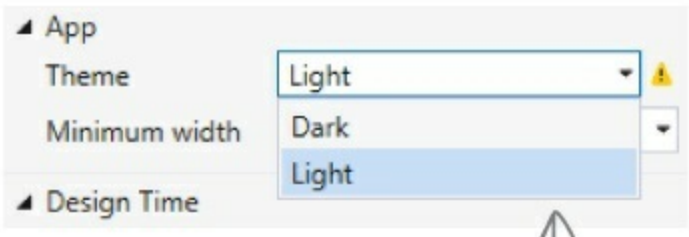
You'll often want to have controls show or hide themselves based on a boolean property in the data context. You can only bind the Visibility property of a control to a target property that's of the type `Visibility` (meaning it returns values like `Visibility.Collapsed`). We'll add a converter called `BooleanVisibilityConverter` that will let us bind a control's `Visibility` property to a Boolean target property to make it visible or invisible.

```
Visibility="{Binding Running, Converter={StaticResource visibilityConverter}}"
```

- **❶ MODIFY THE VIEWMODEL'S TICK EVENT HANDLER.**
  Modify the `DispatcherTimer`'s `Tick` event handler to raise a `PropertyChanged` event if the value of the `Running` property has changed:

```
int _lastHours;
int _lastMinutes;
decimal _lastSeconds;
bool _lastRunning;
void TimerTick(object sender, object e) {
    if (_lastRunning != Running) {
        _lastRunning = Running;
        OnPropertyChanged("Running");
    }
    if (_lastHours != Hours) {
        _lastHours = Hours;
        OnPropertyChanged("Hours");
    }
    if (_lastMinutes != Minutes) {
        _lastMinutes = Minutes;
        OnPropertyChanged("Minutes");
    }
    if (_lastSeconds != Seconds) {
        _lastSeconds = Seconds;
        OnPropertyChanged("Seconds");
    }
}
```

We added the Running check to the timer. Would it make more sense to have the Model fire an event instead?



If you have trouble seeing your user control in the designer, try switching to the Light app theme in the Device window.

- **❷ ADD A CONVERTER THAT INVERTS BOOLEAN VALUES.**
  Here's a value converter that converts `true` to `false` and vice versa. You can use it with Boolean properties on your controls like `IsEnabled`.

```
using Windows.UI.Xaml.Data;

class BooleanNotConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter, string language) {
        if ((value is bool) && ((bool)value) == false)
            return true;
        else
            return false;
    }
    public object ConvertBack(object value, Type targetType, object parameter, string language) {
        throw new NotImplementedException();
    }
}
```

- **❸ ADD A CONVERTER THAT CONVERTS BOOLEANS TO VISIBILITY ENUMS.** You've already seen how you can make a control visible or invisible by setting its `Visibility` property to `Visible` or `Collapsed`. These values come from an enum in the *Windows.UI.Xaml* namespace called `Visibility`. Here's a converter that converts Boolean values to Visibility values:

```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Data;

class BooleanVisibilityConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter, string language) {
        if ((value is bool) && ((bool)value) == true)
            return Visibility.Visible;
        else
            return Visibility.Collapsed;
    }
    public object ConvertBack(object value, Type targetType, object parameter, string language) {
        throw new NotImplementedException();
    }
}
```

- **❹ MODIFY YOUR BASIC STOPWATCH CONTROL TO USE THE CONVERTERS.** Modify *BasicStopwatch.xaml* to add instances of these converters as static resources:

```
<viewmodel:BooleanVisibilityConverter x:Key="visibilityConverter"/>
<viewmodel:BooleanNotConverter x:Key="notConverter"/>
```

Now you can bind the controls' `IsEnabled` and `Visibility` properties to the ViewModel's `Running` property:

```
<StackPanel Orientation="Horizontal">
    <Button IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
            Click="StartButton_Click">Start</Button>
    <Button IsEnabled="{Binding Running}" Click="StopButton_Click">Stop</Button>
    <Button Click="ResetButton_Click">Reset</Button>
    <Button IsEnabled="{Binding Running}" Click="LapButton_Click">Lap</Button>
</StackPanel>
<TextBlock Text="Stopwatch is running"
           Visibility="{Binding Running, Converter={StaticResource visibilityConverter}}"/>
```

This enables the Start button only if the stopwatch is not running.

This causes a TextBlock to become visible when the stopwatch is running.

Styles set properties on multiple controls

When you build out the View layer of your app, you're typically writing mostly XAML code. Those XAML controls are just objects, so it's definitely *possible* to build the entire View using nothing but C# code, but XAML is really optimized to make that job a lot easier.

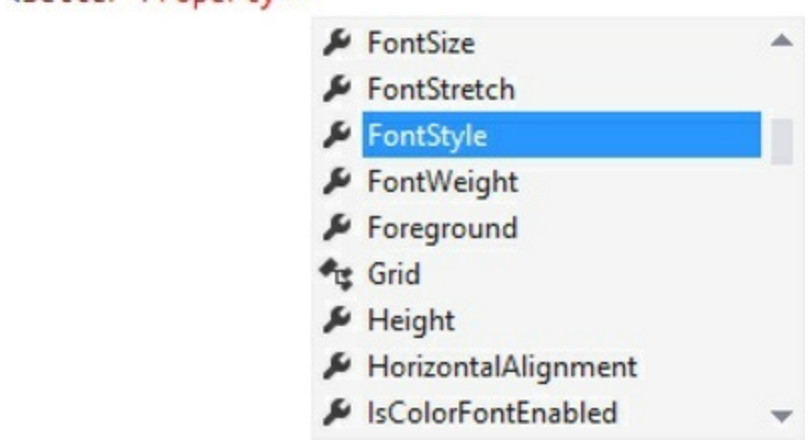
You create a style using the `<Style>` tag. Inside the style, you can use `<Setter>` tags to set properties on any controls that the style is applied to. Let's take a closer look at how this works, using a simple example of **a style that alters all TextBlock controls**.

DO THIS!

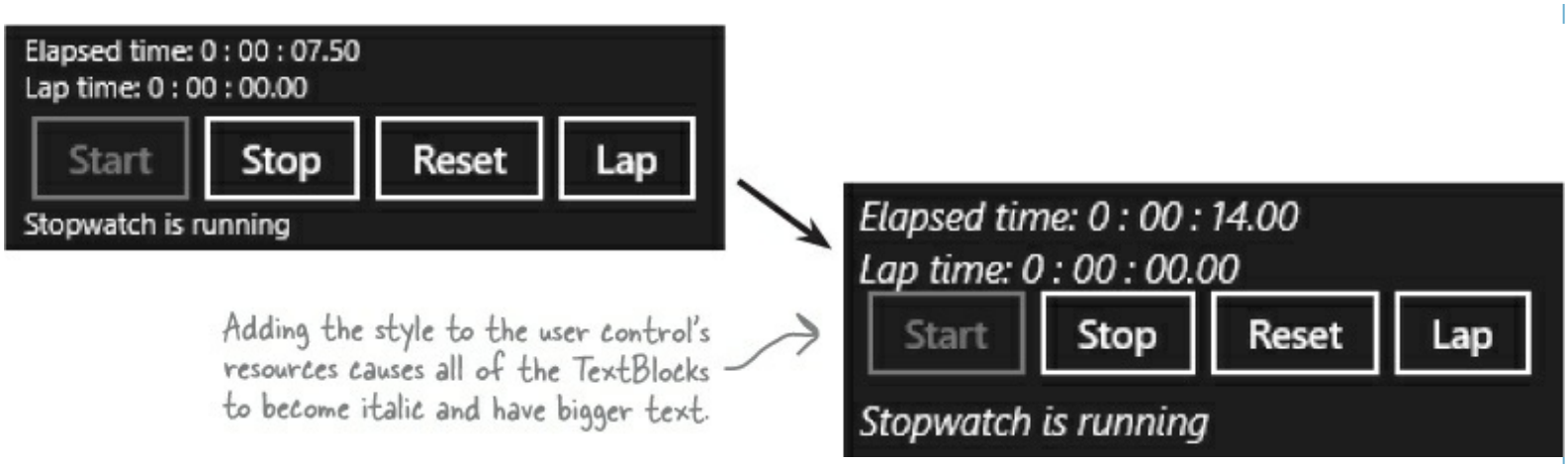
- ❶ Start by modifying the `<UserControl.Resources>` in *BasicStopwatch.xaml* to add the `<Style>` tag. You'll use a `<Setter>` to set the FontStyle to Italic, and set the style's `BasedOn` property to base the style on `SubtitleTextBlockStyle`.

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Name="viewModel"/>
    <viewmodel:TimeNumberFormatConverter x:Name="timeNumberFormatConverter"/>
    <viewmodel:BooleanVisibilityConverter x:Key="visibilityConverter"/>
    <viewmodel:BooleanNotConverter x:Key="notConverter"/>
    <Style TargetType="TextBlock" BasedOn="{StaticResource SubtitleTextBlockStyle}">
        <Setter Property="FontStyle" Value="Italic"/>
    </Style>
</UserControl.Resources>
```

Watch the IntelliSense windows that pop up as you enter the XAML. The TargetType for the style is TextBlock, which means this style applies to TextBlock controls — so when you start to enter the Property in the Setter, the IDE shows you TextBlock properties:



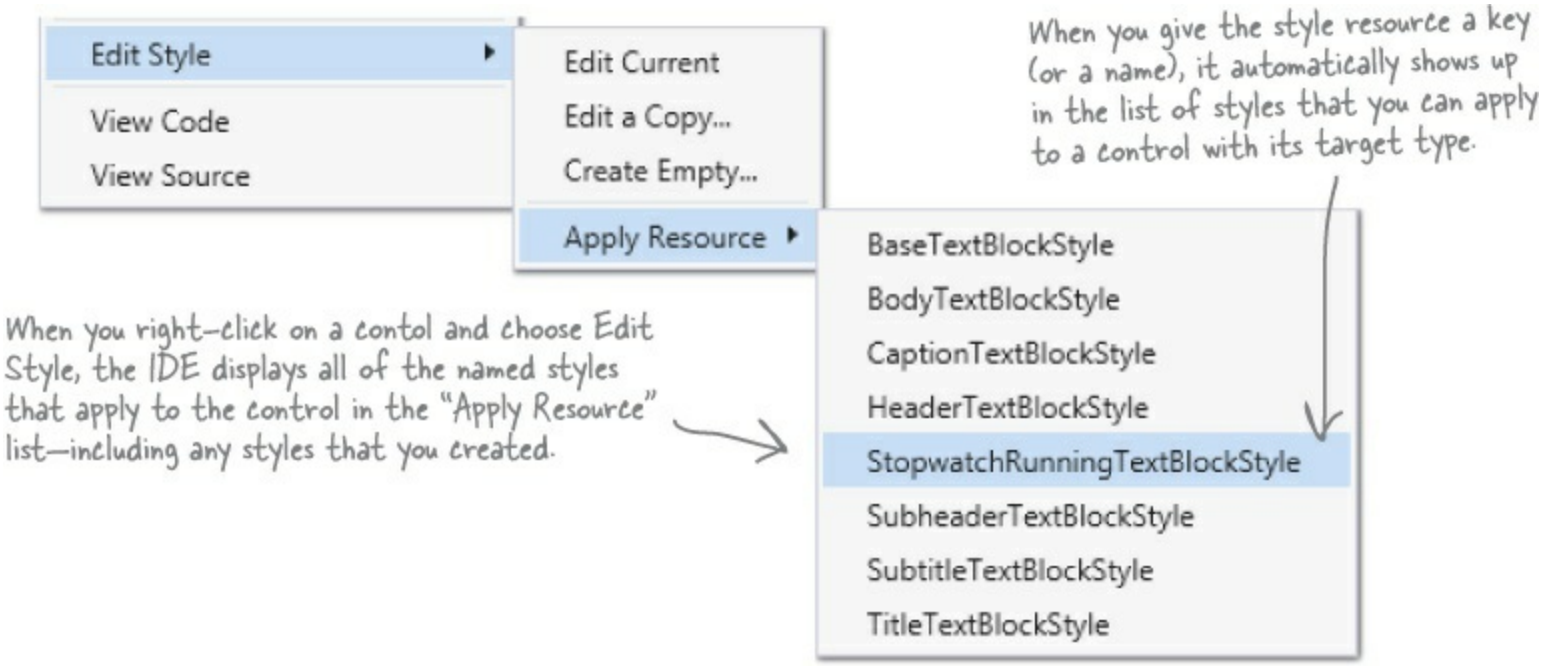
As soon as you finish adding the `<Style>` to the user control resources, you'll see a change in the designer. The style **automatically gets applied to every TextBlock in the user control**.



- ❷ But what if we ***don't want to alter every TextBlock control***? What if we only want to alter specific ones? No problem. We just need to an `x:Key` property to the Style:

```
<Style TargetType="TextBlock" BasedOn="{StaticResource SubtitleTextBlockStyle}"
                              x:Key="StopwatchRunningTextBlockStyle">
<Setter Property="FontStyle" Value="Italic"/>
</Style>
```

As soon as you make the change, the TextBlock controls go back to the way they were. Right-click on the "Stopwatch is running" TextBlock and choose **Edit Style→Apply Resource** from the menu



Use the IDE to apply your new style resource to the "Stopwatch is running" label, just like you've done with the built-in styles. Here's what its XAML should look like after you apply the style:

```
<TextBlock Text="Stopwatch is running"
       Visibility="{Binding Running, Converter={StaticResource visibilityConverter}}"
       Style="{StaticResource StopwatchRunningTextBlockStyle}"/>
```

Now the style is applied only to the "Stopwatch is running" TextBlock on the bottom — the rest of the TextBlock controls have reverted back to how they looked before.

BRAIN POWER

Your new style is based on an existing style called SubtitleTextBlockStyle. Where do you think that style is defined?

Flip the page to keep going!

Use a resource dictionary to share resources between pages

When you add styles and other static resources to the `<Resources>` section of a page or user control, ***you're adding an object to a resoucre dictionary***. A ResourceDictionary works like a Dictionary<object, object>. The dictionary key is defined with `x:Key` (or `x:Name` if you want a variable name), and the value is the static resource object (like a Style, StopwatchViewModel, or BooleanNotConverter). You can see this for yourself: put a breakpoint in your code, run it, and add a watch for `this.Resources["viewModel"]`. Let's keep going with the basic stopwatch. Next, you'll **move your new style to a resource dictionary in the View folder**.

- ❸ Right-click on the View folder, choose **Add→New Item...**, and choose . Give it the name *StopwatchStyles.xaml*. **Cut** the style from the user control and **paste** it into the resource dictionary:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Stopwatch.View">

    <Style TargetType="TextBlock" BasedOn="{StaticResource SubtitleTextBlockStyle}"
                                  x:Key="StopwatchRunningTextBlockStyle">
        <Setter Property="FontStyle" Value="Italic"/>
    </Style>

</ResourceDictionary>
```

Move this style from the user control's resoucres to the resource dictionary.

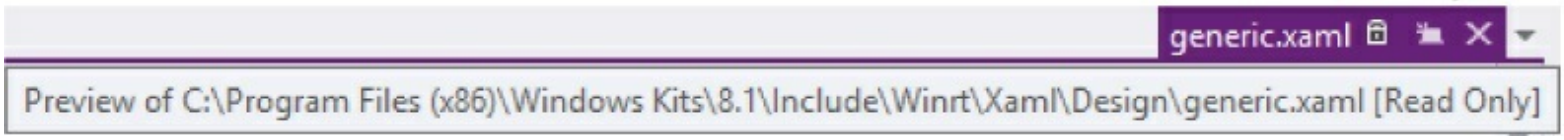
- ❹ Your app has its own resource dictionary, and any resources that your controls use come from other resource dictionaries that have been **merged** into it. Merge your newly added dictioanry into your app's resource dictionary — **open *App.xaml* and add a new `<Application.Resources>` section**:

```
<Application
    x:Class="Stopwatch.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Stopwatch">

    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="View/StopwatchStyles.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>

</Application>
```

This is the XAML tag for your app's resource dictionary. You can merge your new resource dictionary into it by adding a MergedDictionaries section.

- ❺ Let's do some more experimenting. Open *StopwatchStyles.xaml*, find the BasedOn property for your style, then **right-click on SubtitleTextBlockStyle and choose Go To Definition**. This will open up a file called *generic.xaml* and jump to the style definition for SubtitleTextBlockStyle. Scroll to the top of *generic.xaml* to find the opening tag — it's a <ResourceDictionary>, and all of its styles are merged into your app automatically.

Hover your cursor over the generic.xaml tab in the IDE to see a tooltip that shows the location of the file.



- ❻ Search inside generic.xaml for `<Style TargetType="Button">` to find the base style that's applied to all buttons. It's a large style, but it's easier to understand if you collapse the <ControlTemplate> tag:

Collapse this. →

```
<ControlTemplate TargetType="Button">
```

Select the entire style, copy it, and paste it into *StopwatchStyles.xaml* (right before the closing </ResourceDictionary> tag). Here's what you copied and pasted:

```
<Style TargetType="Button">
    <Setter Property="Background" Value="{ThemeResource ButtonBackgroundThemeBrush}" />
    <Setter Property="Foreground" Value="{ThemeResource ButtonForegroundThemeBrush}"/>
    <Setter Property="BorderBrush" Value="{ThemeResource ButtonBorderThemeBrush}" />
    <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}" />
    <Setter Property="Padding" Value="12,4,12,4" />
    <Setter Property="HorizontalAlignment" Value="Left" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}" />
    <Setter Property="FontWeight" Value="SemiBold" />
    <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}" />
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="Button"...>
        </Setter.Value>
    </Setter>
</Style>
```

Here's the collapsed control template tag. →

This <Setter> sets the Template property to a ControlTemplate. This defines the template for the control. When a button is drawn on the page, Windows looks in the control template to see what to draw, and it shows all of the controls contained in the template.

- ❼ Now that you've copied the button style into your resource dictionary, you can edit it. Add this setter:

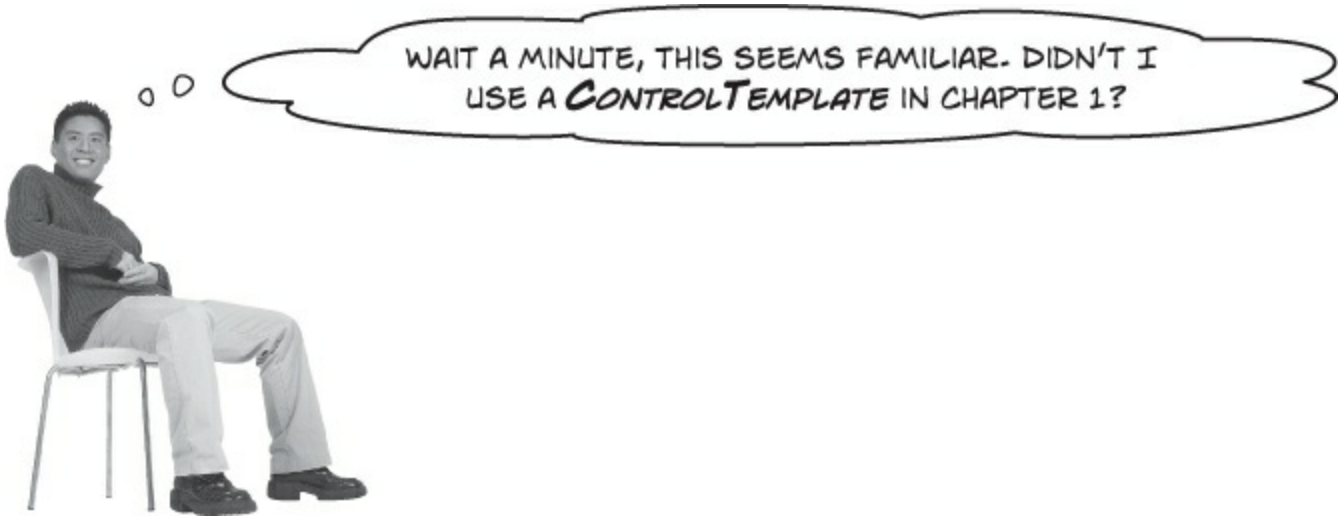
```
<Setter Property="Margin" Value="10"/>
```

Now go back to the app designer. All of your stopwatch buttons are now farther apart, because you've added a 10 pixel margin to each one. Your style now controls the buttons.

- ❽ Run your program and press Tab to switch the focus between the buttons.

A button shows that it's focsed by drawing a dashed rectangle around itself. That rectangle is part of the control template.

Expand the Control Template and locate the two `<Rectangle>` controls near the bottom — they're named `FocusVisualWhite` and `FocusVisualBlack`. Both controls have an `Opacity` property set to 0, which makes them invisible. When the button is focused, it sets that `Opacity` to 1, and the rectangle shows.



Yes! You used the IDE to create a template for the enemies.

Take a minute and **look back at that code** to see exactly how this worked. The IDE added the control template as a static resource named `EnemyTemplate`, and you were able to make the enemy control look like an alien by setting its `Template` property to point to the template. The IDE created the template with an `x:Key` (and not an `x:Name`), so your code used the `Resources` dictionary to look it up by name.

Visual states make controls respond to changes

When you hover over a button, it changes from being transparent to being opaque. When you tab to switch focus to the button, a dashed line appears around it. These things happen because **you changed the state of the button**. When you hover over it, that puts it into a state called `PointerOver`, and when you change focus it puts it into a focus state. There are lots of different states that a control can be in, and most controls don't need to respond to every state.

Controls and control templates use **visual state groups** to change the way the control looks and acts when it's in a specific state. Buttons have a visual state group called `CommonStates` that include a state called `Normal`, one called `PointerOver` (when the pointer is hovering over the button), one called `Pressed` (when the user is actually pressing the button), and one called `Disabled` (when the button is disabled). The button style you copied has a <VisualStateGroup> section that determines how the button's properties change when it's in one of those common states. Here's how it works:

```
<VisualStateGroup x:Name="CommonStates">
    <VisualState x:Name="Normal"/>
    <VisualState x:Name="PointerOver">
        <Storyboard>
            Animation for a property when the pointer is over the button
            Another animation for a different property for the same state
        </Storyboard>
    </VisualState>
    <VisualState x:Name="Pressed">
        <Storyboard>
            Animation for a property when the button is pressed
            Another animation for a different property for the same state
            And another—it can have many animations for properties
        </Storyboard>
    </VisualState>
    <VisualState x:Name="Disabled">
        <Storyboard>
            Animation for a property when the button is pressed
            Another animation for a different property for the same state
            etc.
        </Storyboard>
    </VisualState>
</VisualStateGroup>
```

The control doesn't need to do anything special when it's in the normal state.

Each state is handled with a `<VisualState>` tag, which contains a `Storyboard`. `Storyboards` control animations, and can use a timeline to determine when animations begin and end.

Each `Storyboard` uses animations to modify properties. When the control enters the state, the `Storyboard` starts the animations, which change the values of the properties.

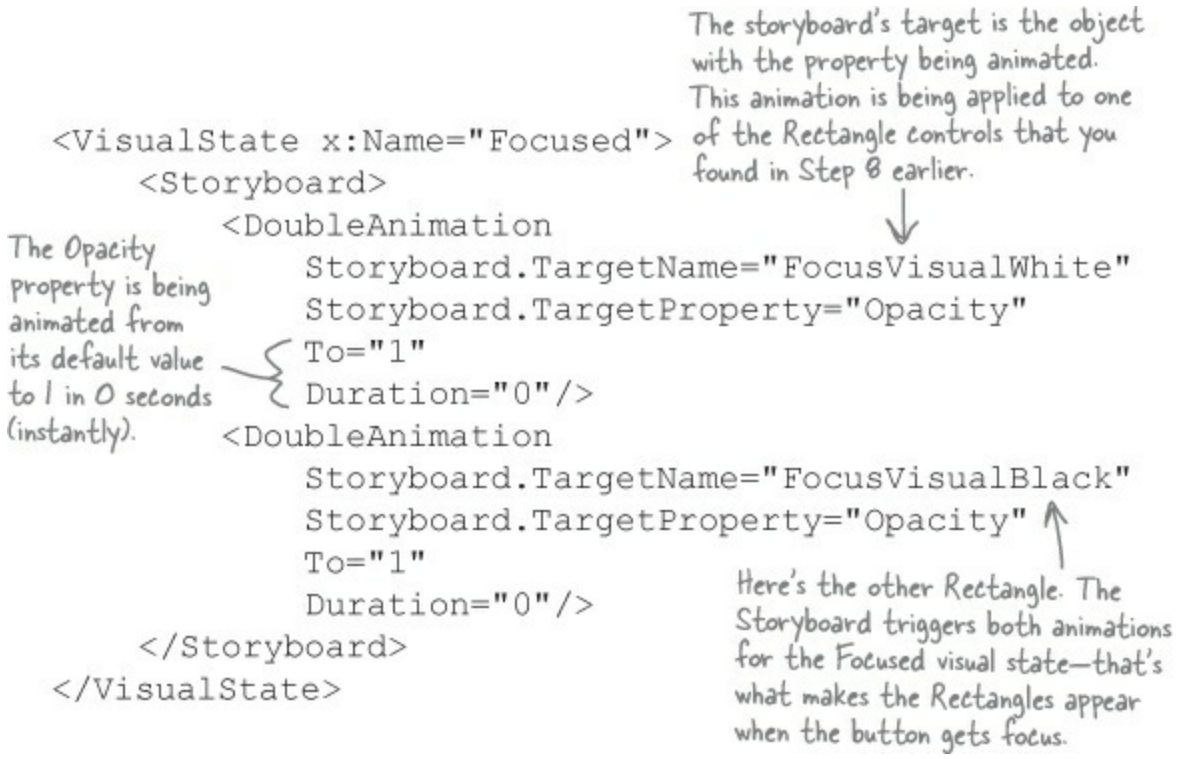
NOTE

Take a minute and look closely at the button style that you copied into your resource dictionary. All it has are <Setter> tags, including one that sets the Template property to a ControlTemplate. Try using the ⊟ button in the IDE to collapse the <VisualStateManager.VisualStateGroups> tag. Once you do that, you can see that the control template just has a Grid that contains a Border with a ContentPresenter (which displays the contents of the button) and two Rectangles. Add the visual state management, and you get a button!

Try changing the Border to give it rounded corners. What happens to your app's buttons?

Use DoubleAnimation to animate double values

When you set a numeric property like `Width` or `Height` on a control in XAML, that sets the value of a `double` property on the control object. `DoubleAnimation` gives you a way to **gradually change that double value from one value to another over a time period**, and is often used to modify the control when it enters a visual state. The control template the button style that you copied uses `DoubleAnimation` to animate the focused state by changing the opacity of the two rectangles around the edge of the control from 0 (clear) to 1 (opaque). The duration is zero, which means the animation happens instantly:



A Storyboard is a container that's used to organize and apply animateions. When the button gets focus, the visual state manager looks for a VisualState called "Focused" and executes its Storyboard to trigger the animations. When the button leaves the Focused state, the Storyboard resets, and all animations go back to their initial state, which in this case means the Opacity is set back to 0 (invisible).

Let's experiment with this animation to get a feel for how it works:

- ❶ Modify the `DoubleAnimation` tags in the `Focused` visual state to change it from a zero-duration animation to one that takes five seconds. Durations are always in the form ***`hours:minutes:seconds`***, so **change it to `Duration="0:0:5"`** (in both animations, so it works with light and dark themes).
- ❷ Start your program, then **use the Tab key** to change focus between the buttons. The dashed outline should now fade in **very slowly** over five seconds. Try the duration to half a second: **`Duration="0:0:0.5"`**
- ❸ Stop the program so we can modify the animation again. This time, **modify the Duration and add two more properties**. Setting AutoReverse to `true` reverses the animation after it's complete, and setting RepeatBehavior to `Forever` causes it to loop for as long as the target control is displayed:

```
Duration="0:0:0.5" AutoReverse="true" RepeatBehavior="Forever"
```

- ❹ Run your program again. Now the focus rectangle pulses by fading in for half a second, then fading out for half a second, and repeats as long as the button is in focus.

Use object animations to animate object values

While some properties on your controls use double values, others use objects. For example, when you set the `Foreground` property to Black, you're actually setting it to a `SolidColorBrush` object. You can see an example of this in the animation for your button style's `Pressed` visual state, which uses an `ObjectAnimationUsingKeyFrames` animation to change the color of the the Border control that makes up the rectangular button body when the button is pressed:

```
<VisualState x:Name="Pressed">
    <Storyboard>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="Border"
                                       Storyboard.TargetProperty="Background">
            <DiscreteObjectKeyFrame KeyTime="0"
                    Value="{ThemeResource ButtonPressedBackgroundThemeBrush}" />
        </ObjectAnimationUsingKeyFrames>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="ContentPresenter"
                                       Storyboard.TargetProperty="Foreground">
            <DiscreteObjectKeyFrame KeyTime="0"
                    Value="{ThemeResource ButtonPressedForegroundThemeBrush}" />
        </ObjectAnimationUsingKeyFrames>
    </Storyboard>
</VisualState>
```

Key frame animations work by creating **key frames**. A key frame is a discrete event that happens at a specific times during the animation. You can see how this works by **adding a third animation to the `Pressed` storyboard**. Add this right above the closing `</Storyboard>` tag:

```
<ObjectAnimationUsingKeyFrames RepeatBehavior="Forever"
         Storyboard.TargetName="Border" Storyboard.TargetProperty="BorderBrush">
    <DiscreteObjectKeyFrame KeyTime="0:0:0"
                            Value="{TemplateBinding BorderBrush}"/>
    <DiscreteObjectKeyFrame KeyTime="0:0:0.2" Value="White"/>
    <DiscreteObjectKeyFrame KeyTime="0:0:0.4"
                            Value="{TemplateBinding BorderBrush}"/>
    <DiscreteObjectKeyFrame KeyTime="0:0:0.6" Value="Blue"/>
    <DiscreteObjectKeyFrame KeyTime="0:0:0.8"
                            Value="{TemplateBinding BorderBrush}"/>
</ObjectAnimationUsingKeyFrames>
```

If you scroll down and look at the <Border> you'll see this value, which is set to the value of the Button's BorderBrush property.

Run your program again. Now when you press a button, the border around the button will flash different colors. Notice how the animation stops partway through if you stop pressing the button? That's because the state changed back to `Normal`, so the animation reset to its starting point.

Here's an example of how a control template for a checkbox uses visual states:
http://msdn.microsoft.com/library/windows/apps/hh465374.aspx

Build an analog stopwatch using the same ViewModel

The MVVM pattern **decouples** the View from the ViewModel, and the ViewModel from the Model. This is really useful if you need to make changes to one of the layers. Because of that decoupling, you can be very confident that the changes you make will not cause the "shotgun surgery" effect and ripple into the other layers. So did we do a good job decoupling the stopwatch program's View from its ViewModel? There's one way to be sure: let's build an entirely new View without changing the existing classes in the ViewModel. The only change you'll need in the C# code **is a new converter in the ViewModel** that converts minutes and seconds into angles.

NOTE

Remember how you used the data classes you built for Jimmy's Comics in Chapter 14 and reused them to create a Split App without making any changes? This is the same idea.

DO THIS!

- **❶ ADD A CONVERTER TO CONVERT TIME TO ANGLES.**
  Add the `AngleConverter` class to the *ViewModel* folder. You'll use it for the hands on the face.

```
using Windows.UI.Xaml.Data;

class AngleConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter, string language) {
        double parsedValue;
        if ((value != null)
            && double.TryParse(value.ToString(), out parsedValue)
            && (parameter != null))
            switch (parameter.ToString()) {
                case "Hours":
                    return parsedValue * 30;
                case "Minutes":
                case "Seconds":
                    return parsedValue * 6;
            }
        return 0;
    }

    public object ConvertBack(object value, Type targetType, object parameter, string language) {
        throw new NotImplementedException();
    }
}
```

An hour value ranges from 0 to 11, so to convert to an angle it's multiplied by 30.

Minutes and seconds range from 0 to 59, so the angle conversion means multiplying by 6.

- **❷ ADD THE NEW USERCONTROL.**
  Add a **new user control called `AnalogStopwatch.xaml` to the *View* folder** and add the `ViewModel` namespace to the `<UserControl>` tag:

```
d:DesignHeight="300"
d:DesignWidth="400"
xmlns:viewmodel="using:Stopwatch.ViewModel">
```

And add the ViewModel and two converters to the user control's static resources.

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Name="viewModel"/>
    <viewmodel:BooleanNotConverter x:Key="notConverter"/>
    <viewmodel:AngleConverter x:Key="angleConverter"/>
</UserControl.Resources>
```

- **❸ ADD THE FACE AND HANDS TO THE GRID.**
  Modify the `<Grid>` tag to add the stopwatch face, using four rectangles for hands.

```
<Grid x:Name="baseGrid" DataContext="{StaticResource ResourceKey=viewModel}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="400"/>
    </Grid.ColumnDefinitions>
    <Ellipse Width="300" Height="300" Stroke="Black" StrokeThickness="2">
        <Ellipse.Fill>
            <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                <LinearGradientBrush.RelativeTransform>
                    <CompositeTransform CenterY="0.5" CenterX="0.5" Rotation="45"/>
                </LinearGradientBrush.RelativeTransform>
                <GradientStop Color="#FFB03F3F"/>
                <GradientStop Color="#FFE4CECE" Offset="1"/>
            </LinearGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="2" Height="150" Fill="Black">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-60"/>
                <RotateTransform Angle="{Binding Seconds,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Seconds}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="4" Height="100" Fill="Black">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-50"/>
                <RotateTransform Angle="{Binding Minutes,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Minutes}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="1" Height="150" Fill="Yellow">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-60"/>
                <RotateTransform Angle="{Binding LapSeconds,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Seconds}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="2" Height="100" Fill="Yellow">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-50"/>
                <RotateTransform Angle="{Binding LapMinutes,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Minutes}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Ellipse Width="10" Height="10" Fill="Black"/>
</Grid>
```

Setting the column width keeps it from expanding to fill whatever container it's in.

This is the face of the stopwatch. It has a black outline and a grayish gradient background.

Here's the second hand. It's a long, thin Rectangle with a translate and rotate transform.

Every control can have one RenderTransform section.

Here's the minute hand.

There are two yellow hands for the lap time.

The TransformGroup tag lets you apply multiple transforms to the same control.

This draws an extra circle in the middle to cover up where the hands overlap. Since it's at the bottom of the Grid, it's drawn last and ends up on top.

The stopwatch face is filled with a gradient brush, just like the background you used in *Save the Humans*.

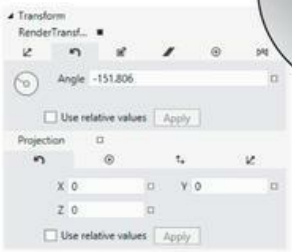
Each hand is transformed twice. It starts out centered in the face, so the first transform shifts it up so that it's in position to rotate.

```
<TranslateTransform Y="-60"/>
```

```
<RotateTransform Angle="{Binding Seconds,
    Converter={StaticResource ResourceKey=angleConverter},
    ConverterParameter=Seconds}"/>
```

The second transform rotates the hand to the correct angle. The `Angle` property of the rotation is bound to seconds or minutes in the ViewModel, and uses the angle converter to convert it to an angle.

Every control can have one `RenderTransform` element that changes how it's displayed. This can include rotating, moving to an offset, skewing, scaling its size up or down, and more.



You used transforms in *Save the Humans* to change the shape of the ellipses in the enemy to make it look like an alien.

Your stopwatch will start ticking as soon as you add the second hand, because it creates an instance of the ViewModel as a static resource to render the control in the designer. The designer may stop it updating, but you can restart it by switching away from the designer window and back again.

- **❹ ADD THE BUTTONS TO THE STOPWATCH.**
  If you haven't already, modify the button style in your resource dictionary to give the buttons rounded corners:

```
<Border x:Name="Border"
        Background="{TemplateBinding Background}"
        BorderBrush="{TemplateBinding BorderBrush}"
        BorderThickness="{TemplateBinding BorderThickness}"
        Margin="3"
        CornerRadius="15">
```

Setting the Border control's CornerRadius property to 15 will give the buttons rounded corners. Combine that with the flashing border when the button is pressed, and you've got a pretty interesting customized button.

We can use the same "Stopwatch is running" TextBlock, so we'll need to add the Visibility converter as a static resource:

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Name="viewModel"/>
    <viewmodel:BooleanNotConverter x:Key="notConverter"/>
    <viewmodel:AngleConverter x:Key="angleConverter"/>
```

```
    <viewmodel:BooleanVisibilityConverter x:Key="visibilityConverter"/>
</UserControl.Resources>
```

Now you can add the buttons. Since it's an analog stopwatch, let's use symbols instead of text by setting the font to Segoe UI Symbol and using the Unicode character codes:

We like the way the buttons look when they overlap the face, so we gave them a bottom vertical alignment and 30 pixel bottom margin.

```
<StackPanel Orientation="Horizontal" HorizontalAlignment="Center"
            VerticalAlignment="Bottom" Margin="0,0,0,30">
    <Button IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
    Click="StartButton_Click" FontFamily="Segoe UI Symbol">&#xE102;</Button>
    <Button IsEnabled="{Binding Running}"
            Click="StopButton_Click" FontFamily="Segoe UI Symbol">&#xE103;</Button>
    <Button Click="ResetButton_Click" FontFamily="Segoe UI Symbol">&#xE10E;</Button>
    <Button IsEnabled="{Binding Running}" FontFamily="Segoe UI Symbol"
            Click="LapButton_Click" >&#xE16D;</Button>
</StackPanel>
<TextBlock Text="Stopwatch is running" HorizontalAlignment="Left"
          VerticalAlignment="Bottom" Margin="10"
          Visibility="{Binding Running, Converter={StaticResource visibilityConverter}}"
          Style="{StaticResource StopwatchRunningTextBlockStyle}"/>
```

We used Character Map to find appropriate button symbols.

The UserControl is just a Grid with a single cell, so all of these controls are in the same cell. We'll use the alignment and margin properties to position the TextBlock.

- **❺ MODIFY THE CODE-BEHIND AND UPDATE THE MAIN PAGE.**
  You added buttons, but you still need to add their event handler methods. The code-behind for the buttons is the same as in the basic stopwatch:

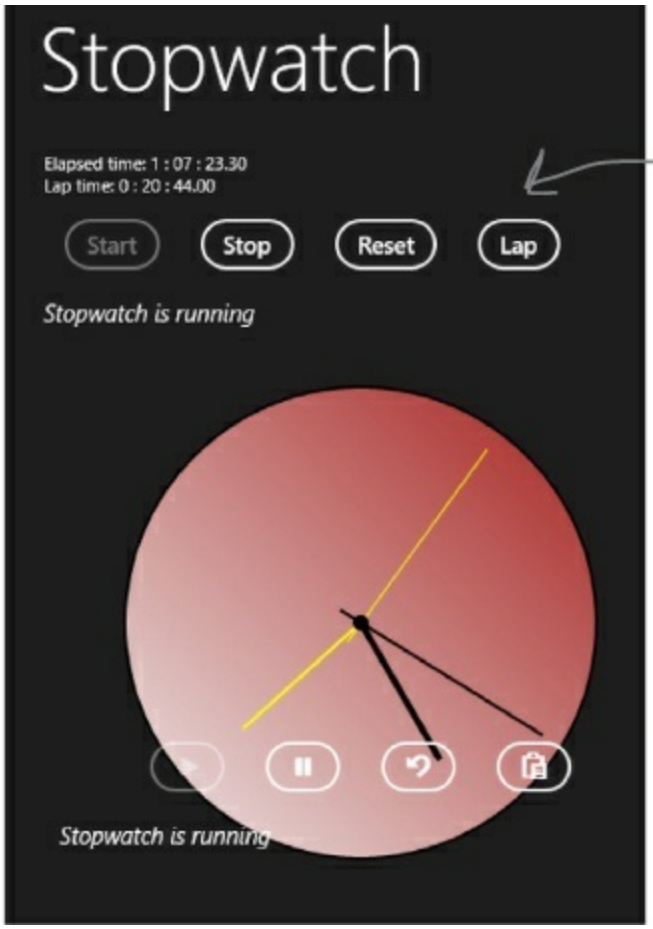
```
private void StartButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Start();
}
private void StopButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Stop();
}
private void ResetButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Reset();
}
private void LapButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Lap();
}
```

Now you just need to modify your *MainPage.xaml* to add an `AnalogStopwatch` control:

```
<StackPanel Orientation="Vertical" Grid.Row="1" Margin="120,0">
    <view:BasicStopwatch Margin="0,0,0,40" />
    <view:AnalogStopwatch/>
</StackPanel>
```

Run your app. Now you have two stopwatch controls on the page.

Each stopwatch keeps its own time, because each one has its own separate instance of the ViewModel as a static resource.



The animation for the Focused visual state causes the rectangle around the button in focus to pulsate.

The buttons are rounded, and have a flashing border while they're being pressed.

Are you wondering why you didn't add the ViewModel resource to your resource dictionary? Try changing the ViewModel to make the _stopwatchModel field static. What does this change about how the stopwatch app behaves? Can you figure out why that happens? This is exactly what would happen if you had a single static resource in the app's resource dictionary. But don't take our word for it — try it yourself and see what happens!

UI controls can be instantiated with C# code, too

You already know that your XAML code instantiates classes in the `Windows.UI` namespace, and you even used the Watch window in the IDE back in Chapter 10 to explore them. But what if you want to create controls from inside your code? Well, controls are just objects, so you can create them and work with them just like you would with any other object. Go ahead and **modify the code-behind to add markings to the face of your analog stopwatch**.

```
using Windows.UI;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Media;
public sealed partial class AnalogStopwatch : UserControl {
    public AnalogStopwatch() {
        this.InitializeComponent();
        AddMarkings();
    }

    private void AddMarkings() {
        for (int i = 0; i < 360; i += 3) {
            Rectangle rectangle = new Rectangle();
            rectangle.Width = (i % 30 == 0) ? 3 : 1;
            rectangle.Height = 15;
            rectangle.Fill = new SolidColorBrush(Colors.Black);
            rectangle.RenderTransformOrigin = new Point(0.5, 0.5);

            TransformGroup transforms = new TransformGroup();
            transforms.Children.Add(new TranslateTransform() { Y = -140 });
            transforms.Children.Add(new RotateTransform() { Angle = i });
            rectangle.RenderTransform = transforms;
            baseGrid.Children.Add(rectangle);
        }
    }
    // ... the button event handlers stay the same
```

You need the Windows.UI namespace for the Colors class, the Windows.UI.Xaml.Shapes namespace for the Rectangle and transforms, and the Windows.UI.Xaml.Media namespace for SolidColorBrush.

Modify the constructor to call a method that adds the markings.

This statement uses the % modulo operator to make the marks for the hours thicker than the ones for the minutes. `i % 30` returns 0 only if `i` is divisible by 30.

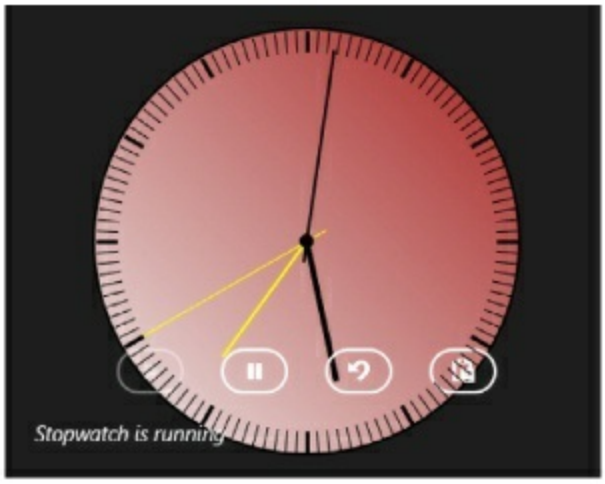
This creates instances of the same Rectangle object that you created with the <Rectangle> tag.

Flip back to the XAML for the hour and minute hands. This code sets up exactly the same transform, except instead of binding the Angle property it sets it to a value.

Controls like Grid, StackPanel, and Canvas have a `Children` collection with references to all of the other controls contained inside them. You can add controls to the grid with its `Add()` method, and remove all controls by calling its `Clear()` method. You add transforms to a `TransformGroup` the same way.

You used a `Binding` object to set up data binding in C# code back in Chapter 11. Can you figure out how to remove the XAML to create the Rectangle controls for the hour and minute hands and replace it with C# code to do the same thing?

Now that you added the markings to the stopwatch, the ref will make all the right calls.

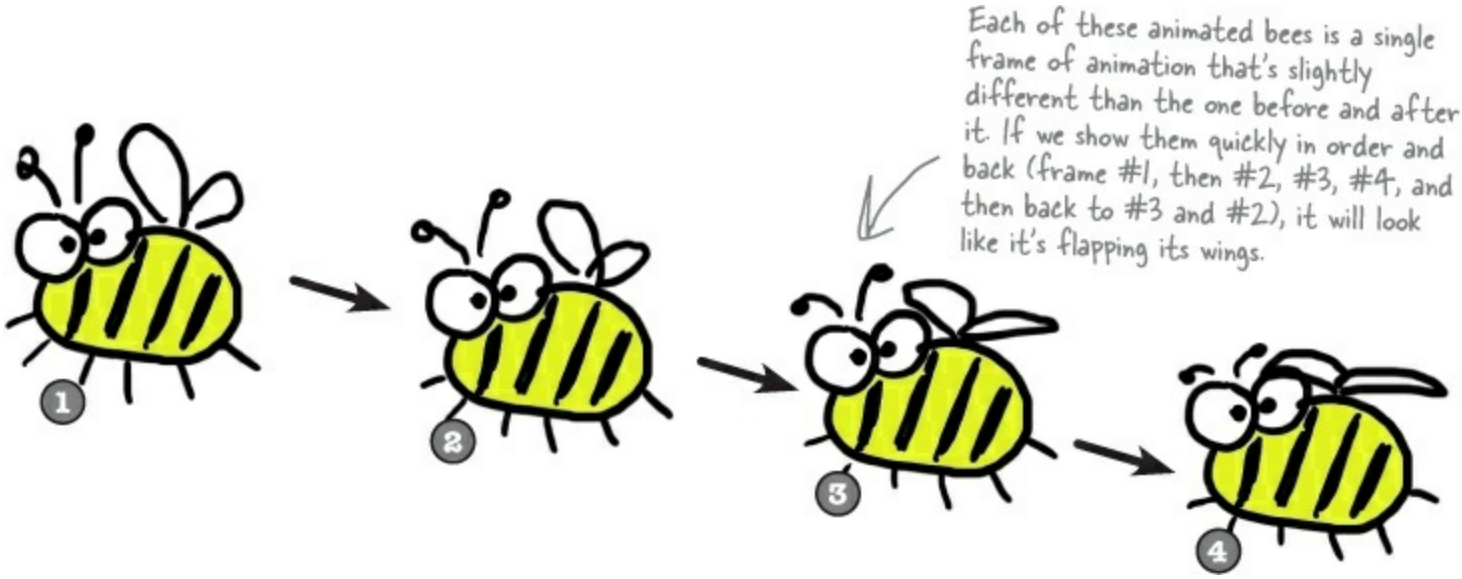


Which team will dominate the conference and win the Objectville Trophy? Nobody's sure. All we know is that Joe, Bob, and Ed will be betting on it!

C# can build “real” animations, too

In the C# and XAML world, *animation* can refer to any property that changes over a specific time period. But in the real world, it means drawings that move and change. So let’s build a simple program to do some “real” animation.

DO THIS!



Create a project and add the pictures

Let's get started with the project. **Create a new Windows Store project called `AnimatedBee`.** Download the four images (they're *.png* files) from the Head First Labs website. Then **add each one to the *Assets* folder**. You'll also need to create *View*, *Model*, and *ViewModel* folders. Right-click on the *Assets* folder and choose **Add→Existing Item...** just like you did in Chapter 14.

Download the images for this chapter from the your class website.

Keep an open mind about animation.

Watch carefully when you bring up the Windows Start page, open an About window, hover over a button, or do any number of things in Windows apps. Animations are everywhere, and once you start looking for them, you'll keep seeing them.

Create a user control to animate a picture

Let's encapsulate all of the frame-by-frame animation code. **Add a user control called *AnimatedImage* to your View folder**. It has very little XAML — all of the intelligence is in the code-behind. Here's everything inside the `<UserControl>` tag in the XAML:

```
<Grid>
    <Image x:Name="image" Stretch="Fill"/>
</Grid>
```

The work is done in the code-behind. Notice its overloaded constructor that calls the `StartAnimation()` method, which **creates storyboard and key frame animation objects** to animate the `Source` property of the Image control.

```
using Windows.UI.Xaml.Media.Animation;
using Windows.UI.Xaml.Media.Imaging;

public sealed partial class AnimatedImage : UserControl {
    public AnimatedImage() {
        this.InitializeComponent();
    }

    public AnimatedImage(IEnumerable<string> imageNames, TimeSpan interval)
        : this()
    {
        StartAnimation(imageNames, interval);
    }

    public void StartAnimation(IEnumerable<string> imageNames, TimeSpan interval) {
        Storyboard storyboard = new Storyboard();
        ObjectAnimationUsingKeyFrames animation = new ObjectAnimationUsingKeyFrames();
        Storyboard.SetTarget(animation, image);
        Storyboard.SetTargetProperty(animation, "Source");

        TimeSpan currentInterval = TimeSpan.FromMilliseconds(0);
        foreach (string imageName in imageNames) {
            ObjectKeyFrame keyFrame = new DiscreteObjectKeyFrame();
            keyFrame.Value = CreateImageFromAssets(imageName);
            keyFrame.KeyTime = currentInterval;
            animation.KeyFrames.Add(keyFrame);
            currentInterval = currentInterval.Add(interval);
        }
        storyboard.RepeatBehavior = RepeatBehavior.Forever;
        storyboard.AutoReverse = true;
        storyboard.Children.Add(animation);
        storyboard.Begin();
    }

    private static BitmapImage CreateImageFromAssets(string imageFilename) {
        return new BitmapImage(new Uri("ms-appx:///Assets/" + imageFilename));
    }
}
```

BitmapImage is in the Media.Imaging namespace. Storyboard and the other animation classes are in the Media.Animation namespace.

Every control must have a parameterless constructor if you want to create an instance of the control using XAML. You can still add overloaded constructors, but that's only useful if you're writing code to create the control.

The static `SetTarget()` and `SetTargetProperty()` methods from the `Storyboard` class set the target object being animated (`"image"`) and the property that will change (`"Source"`).

Once the `Storyboard` object is set up and animations have been added to its `Children` collection, call its `Begin()` method to start the animation.

The CreateImageFromAssets() method takes an image filename and turns it into a BitmapImage object. In Chapter 14 you bound an Image's Source property to a path like `"Assets/picture.jpg"`, but that won't work with key frame animations—it would be really slow to load an image from a file every time a frame is displayed. Using BitmapImage objects lets us load each file once when we set up the animation.

Make your bees fly around a page

Let's take your AnimatedImage control out for a test flight.

DO THIS!

- **❶ REPLACE MAINPAGE.XAML WITH A BASIC PAGE IN THE VIEW FOLDER.**
  Add a **Basic Page to your *View* folder** called *FlyingBees.xaml*. Delete *MainPage.xaml* from the project. Then modify *App.xaml.cs* to navigate to your new page on startup:

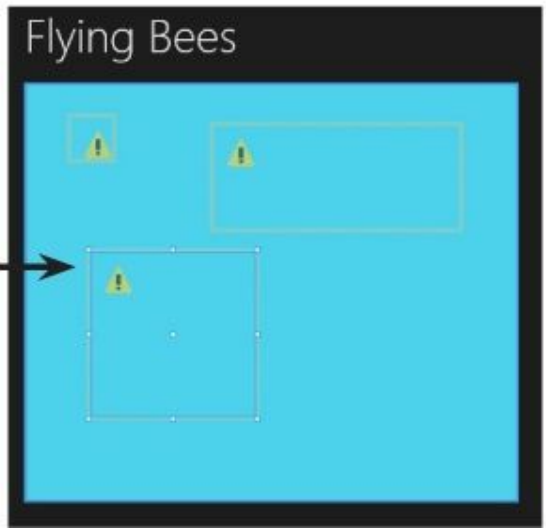
```
rootFrame.Navigate(typeof(View.FlyingBees), e.Arguments);
```

- **❷ THE BEES WILL FLY AROUND A CANVAS CONTROL.**
  You'll need a container for the AnimatedImage controls. So the next thing to do is **add a Canvas control to *FlyingBees.xaml***. A Canvas control is a container, so it can contain other controls like a Grid or StackPanel. The difference is that a Canvas lets you set the coordinates of the controls using the `Canvas.Left` and `Canvas.Top` properties. You used a Canvas back in Chapter 1 to create the play area for *Save the Humans*. Here's the XAML to add to *FlyingBees.xaml*:

This bee's AnimatedImage control starts with its Canvas.Left set to 50 and its Canvas.Top set to 40, so it will be drawn at (50, 40) on the canvas. But it won't stay there for long, because we'll use a double animation to animate Canvas.Left from 50 to 450 and back.

```
<Canvas Grid.Row="1" Background="SkyBlue" Width="600"
        HorizontalAlignment="Left" Margin="120,0,120,120">

    <local:AnimatedImage Canvas.Left="50" Canvas.Top="40"
                         x:Name="firstBee" Width="50" Height="50"/>

    <local:AnimatedImage Canvas.Left="80" Canvas.Top="260"
                         x:Name="secondBee" Width="200" Height="200"/>

    <local:AnimatedImage Canvas.Left="230" Canvas.Top="100"
                         x:Name="thirdBee" Width="300" Height="125"/>
</Canvas>
```

The AnimatedImage control is invisible until its `StartAnimation()` method is called, so the controls in the Canvas will only show up as outlines. You can select them using the Document Outline. Try dragging the controls around the canvas to see the `Canvas.Left` and `Canvas.Top` properties change.



- **❸ ADD THE CODE-BEHIND FOR THE PAGE.**
  You'll need this `using` statement for the namespace that contains `Storyboard` and `DoubleAnimation`:

```
using Windows.UI.Xaml.Media.Animation;
```

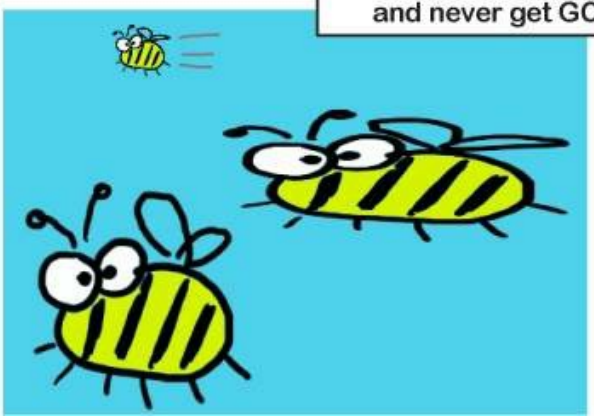
Now you can **modify the constructor in *FlyingBees.xaml.cs*** to start up the bee animation. Let's also create a `DoubleAnimation` to animate the `Canvas.Left` property. Compare the code for creating a storyboard and animation to the XAML code with `<DoubleAnimation>` earlier in the chapter.

```
public FlyingBees() {
    this.InitializeComponent();
    this.navigationHelper = new NavigationHelper(this);
    this.navigationHelper.LoadState += navigationHelper_LoadState;
    this.navigationHelper.SaveState += navigationHelper_SaveState;

    List<string> imageNames = new List<string>();
    imageNames.Add("Bee animation 1.png");
    imageNames.Add("Bee animation 2.png");
    imageNames.Add("Bee animation 3.png");
    imageNames.Add("Bee animation 4.png");

    firstBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(50));
    secondBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(10));
    thirdBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(100));

    Storyboard storyboard = new Storyboard();
    DoubleAnimation animation = new DoubleAnimation();
    Storyboard.SetTarget(animation, firstBee);
    Storyboard.SetTargetProperty(animation, "(Canvas.Left)");
    animation.From = 50;
    animation.To = 450;
    animation.Duration = TimeSpan.FromSeconds(3);
    animation.RepeatBehavior = RepeatBehavior.Forever;
    animation.AutoReverse = true;
    storyboard.Children.Add(animation);
    storyboard.Begin();
}
```

The AnimatedImage control has a construtor that takes a sequence of image names and a TimeSpan, and uses them to animate an image.

Instead of using a <Storyboard> tag and a <DoubleAnimation> tag like earlier in the chapter, you can create the Storyboard and DoubleAnimation objects and set their properties in code.

The `Storyboard` is garbage-collected after the animation completes. You can see this for yourself by using Make Object ID to watch it and clicking ↻ to refresh it after the animation ends—but first change its RepeatBehavior to `new RepeatBehavior(2)` so it doesn't repeat forever and never get GC'd.



Run your program. Now you can see three bees flapping their wings. You gave them different intervals, so they flap at different rates because their timers are waiting for different timespans before changing frames. The top bee has its Canvas. Left property animated from 50 to 450 and back, which causes it to move around the page. Take a close look at the properties that are set on the DoubleAnimation object and compare them to the XAML properties you used earlier in the chapter.

Something's not right about this project. Can you spot it?

Something's not right: there's nothing in your *Model* or *ViewModel* folder, and you're creating dummy data in the View. That's not MVVM!

If we wanted to add more bees, we'd have to create more controls in the View and then initialize them individually. What if we want different sizes or kinds of bees? Or other things to be animated? If we had a Model that was optimized for data, it would be a lot easier. How can we make this project follow the MVVM pattern?

That won't work. Data binding doesn't work with container controls' `Children` property — and for good reason.

Data binding is built to work with **attached properties**, which are the properties that show up in the XAML code. The `Canvas` object *does* have a public `Children` property, but if you try to set it using XAML (`Children="{Binding ...}"`) your code ***won't compile***.

However, you already know how to bind a collection of objects to a XAML control, because you did that with ListView and GridView controls using the `ItemsSource` property. We can take advantage of that data binding to add child controls to a Canvas.

Use ItemsPanelTemplate to bind controls to a Canvas

When you used the `ItemsSource` property to bind items to a ListView, GridView, or ListBox it didn't matter which one you were binding to, because the `ItemsSource` property always worked the same way. If you were going to build three classes that had exactly the same behavior, you would put that behavior in a base class and have the three classes extend it, right? Well, the Microsoft team did exactly the same thing when they built the selector controls. The ListView, GridView, and ListBox all extend a class called `Selector`, which is a subclass of **the `ItemsControl` class that displays a collection of items**.

- ❶ We're going to use its `ItemsPanel` property to **set up a template for the panel that controls the layout of the items**. Start by adding the `ViewModel` namespace to *FlyingBees.xaml*:

```
xmlns:viewmodel="using:AnimatedBee.ViewModel"
```

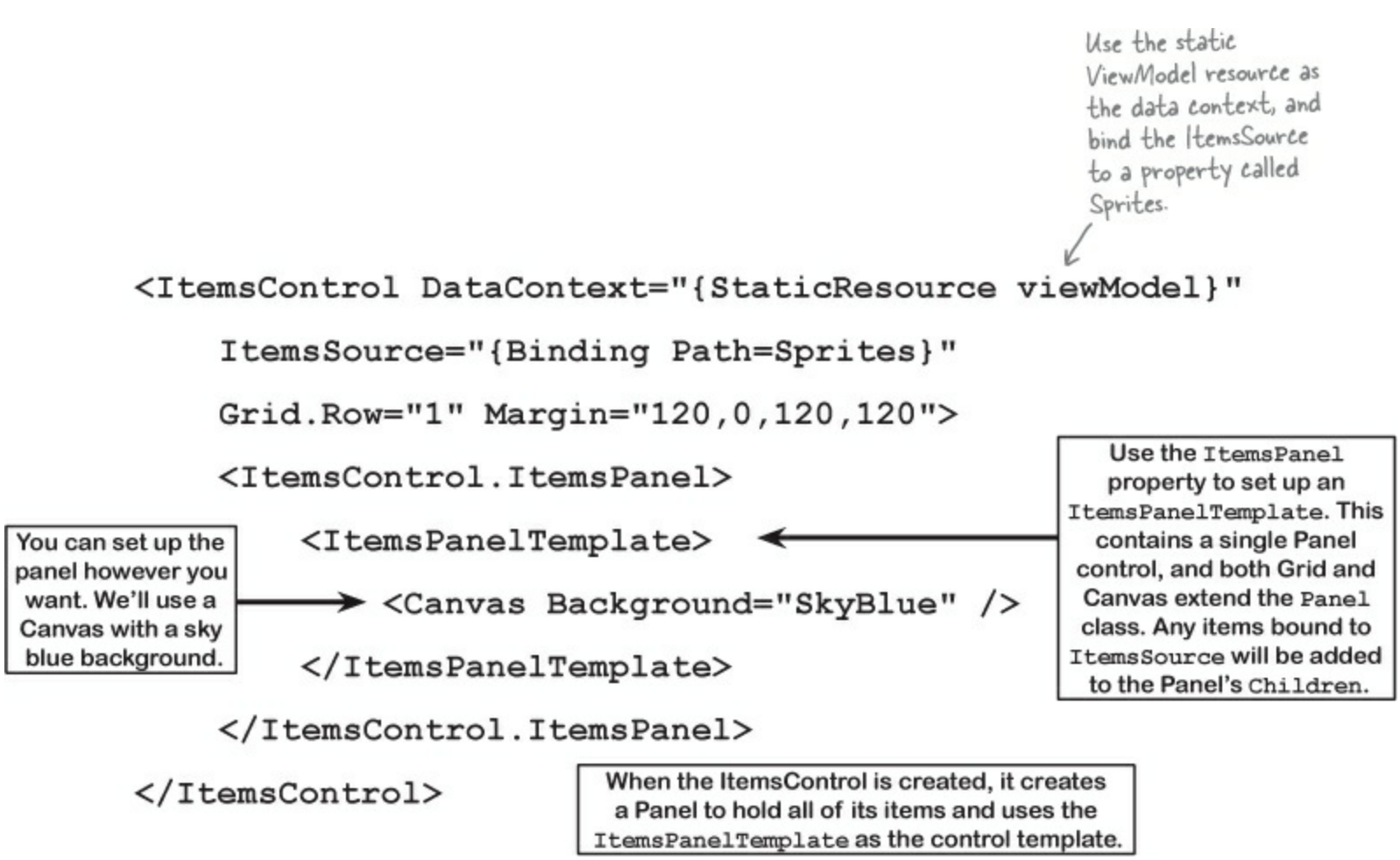
If you used a different project name, change AnimatedBee to the correct namespace.

- ❷ Next, **add an *empty* class called `BeeViewModel` to your *ViewModel* folder**, and then add an instance of that class as a static resource to *FlyingBees.xaml*:

```
<viewmodel:BeeViewModel x:Key="viewModel"/>
```

Edit *FlyingBees.xaml.cs* and **delete all the additional code that you added to the `FlyingBees()` constructor** in the `FlyingBees` control. Make sure that you ***don't delete*** the `InitializeComponents()` method or the three lines after it that initialize the `NavigationHelper` object! (If you're not sure which lines to delete, you can just delete the whole *FlyingBees.xaml* page and add a new one.)

- ❸ Here's the XAML for the ItemsControl. Open *FlyingBees.xaml*, **delete the `<Canvas>` tag** you added, and **replace it with this ItemsControl**:



- ❹ Create a **new class in the *View* folder called `BeeHelper`**. Make sure it's a static class, because it'll only have static methods to help your ViewModel manage its bees.

```
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media.Animation;

static class BeeHelper {
    public static AnimatedImage BeeFactory(
                double width, double height, TimeSpan flapInterval) {
        List<string> imageNames = new List<string>();
        imageNames.Add("Bee animation 1.png");
        imageNames.Add("Bee animation 2.png");
        imageNames.Add("Bee animation 3.png");
        imageNames.Add("Bee animation 4.png");

        AnimatedImage bee = new AnimatedImage(imageNames, flapInterval);
        bee.Width = width;
        bee.Height = height;
        return bee;
    }

    public static void SetBeeLocation(AnimatedImage bee, double x, double y) {
        Canvas.SetLeft(bee, x);
        Canvas.SetTop(bee, y);
    }

    public static void MakeBeeMove(AnimatedImage bee,
                                   double fromX, double toX, double y) {
        Canvas.SetTop(bee, y);
        Storyboard storyboard = new Storyboard();
        DoubleAnimation animation = new DoubleAnimation();
        Storyboard.SetTarget(animation, bee);
        Storyboard.SetTargetProperty(animation, "(Canvas.Left)");
        animation.From = fromX;
        animation.To = toX;
        animation.Duration = TimeSpan.FromSeconds(3);
        animation.RepeatBehavior = RepeatBehavior.Forever;
        animation.AutoReverse = true;
        storyboard.Children.Add(animation);
        storyboard.Begin();
    }
}
```

This factory method creates Bee controls. It makes sense to keep this in the View, because it's all UI-related code.

When you take a small block of code that's reused a lot and put it in its own (often static) method, it's sometimes called a helper method. Putting helper methods in a static class with a name that ends with "Helper" makes your code easier to read.

This is the same code that was in the page's constructor. Now it's in a static helper method.

THE FACTORY METHOD PATTERN

MVVM is just one of many design patterns. One of the most common — and most useful — patterns is the factory method pattern, where you have a "factory" method that creates objects. The factory method is usually static, and the name often ends with "Factory" so it's obvious what's going on.

- ❺ Here's the code for the empty `BeeViewModel` class that you added to the *ViewModel* folder. By moving the UI-specific code to the View, we can keep the code in the ViewModel simple and specific to managing bee-related logic.

This will come in handy in the last lab.

All XAML controls inherit from the `UIElement` base class in the `Windows.UI.Xaml` namespace. We explicitly used the namespace (`Windows.UI.Xaml.UIElement`) in the body of the class instead of adding a `using` statement to limit the amount of UI-related code we added to the ViewModel.

We used `UIElement` because it's the most abstract class that all of the sprites extend. For some projects, a subclass like `FrameworkElement` may be more appropriate, because that's where many properties are defined, including `Width`, `Height`, `Opacity`, `HorizontalAlignment`, etc.

When the AnimatedImage control is added to the _sprites ObservableCollection that's bound to the ItemsControl's ItemsSource property, the control is added to the item panel, which is created based on the ItemsPanelTemplate.

```
using View;
using System.Collections.ObjectModel;
using System.Collections.Specialized;

class BeeViewModel {
    private readonly ObservableCollection<Windows.UI.Xaml.UIElement>
                _sprites = new ObservableCollection<Windows.UI.Xaml.UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    public BeeViewModel() {
        AnimatedImage firstBee =
            BeeHelper.BeeFactory(50, 50,
                                 TimeSpan.FromMilliseconds(50));
        _sprites.Add(firstBee);

        AnimatedImage secondBee =
            BeeHelper.BeeFactory(200, 200, TimeSpan.FromMilliseconds(10));
        _sprites.Add(secondBee);

        AnimatedImage thirdBee =
            BeeHelper.BeeFactory(300, 125, TimeSpan.FromMilliseconds(100));
        _sprites.Add(thirdBee);

        BeeHelper.MakeBeeMove(firstBee, 50, 450, 40);
        BeeHelper.SetBeeLocation(secondBee, 80, 260);
        BeeHelper.SetBeeLocation(thirdBee, 230, 100);
    }
}
```

We're taking two steps to encapsulate Sprites property. The backing field is marked readonly so it can't be overwritten later, and we expose it as an INotifyCollectionChanged property so other classes can only observe it but not modify it.

A sprite is the term for any 2D image or animation that gets incorporated into a larger game or animation.

You're changing properties and adding animations on the controls after they were added to the `ObservableCollection`. Why does that work?

- ❻ Run your app. It should look exactly the same as before, but now the behavior is split across the layers, with UI-specific code in the View and code that deals with bees and moving in the ViewModel.

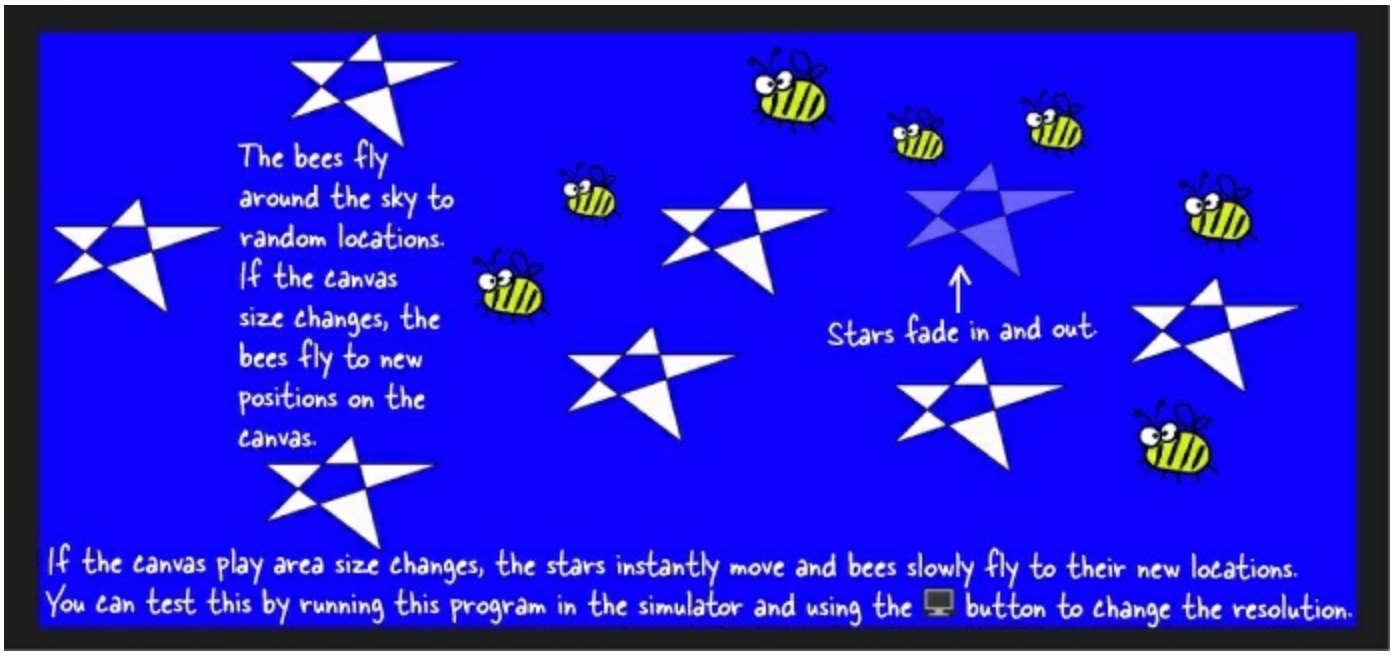
THE READONLY KEYWORD

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.

LONG EXERCISE

This is the last exercise in the book. Your job is to build a program that animates bees and stars. There's a lot of code to write, but you're up to the task...and once you have this working, you'll have all the tools you need to build a complete video game. *(Can you guess what's in Lab #3?)*

- **❶ HERE'S THE APP YOU'LL CREATE.**
  Bees with flapping wings fly around a dark blue canvas, while behind them, stars fade in and out. You'll build a View that contains the bees, stars, and page to display them, a Model that keeps track of where they are and fires off events when bees move or stars change, and a ViewModel to connect the two together.



- **❷ CREATE A NEW WINDOWS STORE APP PROJECT.**
  Create a new project called ***StarryNight***. Next, **add the *Model*, *View*, and *ViewModel* folders**. Once that's done, you'll need to add **an empty class called `BeeStarViewModel`** to the *ViewModel* folder.
- **❸ CREATE A NEW BASIC PAGE IN THE VIEW FOLDER.**
  Add a **Basic Page in the *View* folder** called *BeesOnAStarryNight.xaml*. Add the namespace to the top-level tag in the *BeesOnAStarryNight.xaml* (it should match your project's name, `StarryNight`):

```
xmlns:viewmodel="using:StarryNight.ViewModel"
```

Add the ViewModel as a static resource and change the page name:

```
<Page.Resources>
    <viewmodel:BeeStarViewModel x:Name="viewModel"/>
    <x:String x:Key="AppName">Bees on a Starry Night</x:String>
</Page.Resources>
```

The XAML for the page is **exactly the same** as *FlyingBees.xaml* in the last project, except the Canvas control's background is **`Blue`** and it has a `SizeChanged` event handler:

```
<Canvas Background="Blue" SizeChanged="SizeChangedHandler" />
```

The `SizeChanged` event is fired when a control changes size, with `EventArgs` properties for the new size.

Visual Studio comes with a fantastic tool to help you experiment with shapes! Fire up Blend for Visual Studio 2013 and use the pen, pencil, and toolbox to create XAML shapes that you can copy and paste into your C# projects.

- **❹ ADD CODE-BEHIND FOR THE PAGE AND THE APP.**
  Add the `SizeChanged` event handler to *BeesOnAStarryNight.xaml.cs* in the *View* folder:

> **NOTE**
>
> The code in step 4 won't compile until you add the PlayAreaSize property to the ViewModel in step 9. You can use the IDE to generate a property stub for it for now.

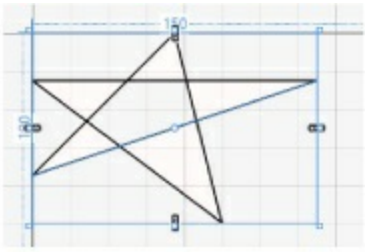
```
private void SizeChangedHandler(object sender, SizeChangedEventArgs e) {
    viewModel.PlayAreaSize = new Size(e.NewSize.Width, e.NewSize.Height);
}
```

Then modify *App.xaml.cs* to change the call to `rootFrame.Navigate()` so the app starts on your new page::

```
rootFrame.Navigate(typeof(View.BeesOnAStarryNight), e.Arguments);
```

- **❺ ADD THE ANIMATEDIMAGE CONTROL TO THE VIEW FOLDER.**
  Go back to the *View* folder and add the AnimatedImage control. This is exactly the same control from earlier in the chapter. Make sure you **add the image files** for the animation frames to the *Assets* folder.
- **❻ ADD A USER CONTROL CALLED STARCONTROL TO THE VIEW FOLDER.**
  This control draws a star. It also has two storyboards, one to fade in and one to fade out. **Add methods called `FadeIn()` and `FadeOut()`** to the code-behind to trigger the storyboards.

> **A Polygon control uses a set of points to draw a polygon. This UserControl uses it to draw a star.**



```
<UserControl
    // The usual XAML code that the IDE generates is fine,
    // no extra namespaces are needed for this User Control.
    >

    <UserControl.Resources>
        <Storyboard x:Name="fadeInStoryboard">
            <DoubleAnimation From="0" To="1" Storyboard.TargetName="starPolygon"
                        Storyboard.TargetProperty="Opacity" Duration="0:0:1.5" />
        </Storyboard>
        <Storyboard x:Name="fadeOutStoryboard">
           <DoubleAnimation From="1" To="0" Storyboard.TargetName="starPolygon"
                        Storyboard.TargetProperty="Opacity" Duration="0:0:1.5" />
        </Storyboard>
    </UserControl.Resources>

    <Grid>
        <Polygon Points="0,75 75,0 100,100 0,25 150,25" Fill="Snow"
                 Stroke="Black" x:Name="starPolygon"/>
    </Grid>
</UserControl>
```

You'll need to add public FadeIn() and FadeOut() methods to the code-behind that start these storyboards. That's how the stars will fade in and out.

This polygon draws the star. You can replace it with other shapes to experiment with how they work.

There are even more shapes beyond ellipses, rectangles, and polygons: http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh465055.aspx

- **❼ ADD THE BEESTARHELPER CLASS TO THE VIEW.**
  Here's a useful helper class. It's got some familiar tools, and a couple of new ones. Put it in the *View* folder.

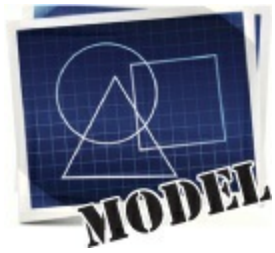
```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media.Animation;
using Windows.UI.Xaml.Shapes;

static class BeeStarHelper {
    public static AnimatedImage BeeFactory(double width, double height, TimeSpan flapInterval) {
        List<string> imageNames = new List<string>();
        imageNames.Add("Bee animation 1.png");
        imageNames.Add("Bee animation 2.png");
        imageNames.Add("Bee animation 3.png");
        imageNames.Add("Bee animation 4.png");

        AnimatedImage bee = new AnimatedImage(imageNames, flapInterval);
        bee.Width = width;
        bee.Height = height;
        return bee;
    }

    public static void SetCanvasLocation(UIElement control, double x, double y) {
        Canvas.SetLeft(control, x);
        Canvas.SetTop(control, y);
    }

    public static void MoveElementOnCanvas(UIElement uiElement, double toX, double toY) {
        double fromX = Canvas.GetLeft(uiElement);
        double fromY = Canvas.GetTop(uiElement);

        Storyboard storyboard = new Storyboard();
        DoubleAnimation animationX = CreateDoubleAnimation(uiElement,
                                        fromX, toX, "(Canvas.Left)");
        DoubleAnimation animationY = CreateDoubleAnimation(uiElement,
                                        fromY, toY, "(Canvas.Top)");
        storyboard.Children.Add(animationX);
        storyboard.Children.Add(animationY);
        storyboard.Begin();
    }

    public static DoubleAnimation CreateDoubleAnimation(UIElement uiElement,
                                    double from, double to, string propertyToAnimate) {
        DoubleAnimation animation = new DoubleAnimation();
        Storyboard.SetTarget(animation, uiElement);
        Storyboard.SetTargetProperty(animation, propertyToAnimate);
        animation.From = from;
        animation.To = to;
        animation.Duration = TimeSpan.FromSeconds(3);
        return animation;
    }

    public static void SendToBack(StarControl newStar) {
        Canvas.SetZIndex(newStar, -1000);
    }
}
```

Canvas has `SetLeft()` and `GetLeft()` methods to set and get the X position of a control. The `SetTop()` and `GetTop()` methods set and get the Y position. They work even after a control is added to the Canvas.

We added a helper called CreateDoubleAnimation() that creates a three-second DoubleAnimation. This method uses it to move a UIElement from its current location to a new point by animating its Canvas.Left and Canvas.Top properties.

"Z Index" means the order the controls are layered on a panel. A control with a higher Z index is drawn on top of one with a lower Z index.

- **❽ ADD THE BEE, STAR, AND EVENTARGS CLASSES TO THE MODEL.**
  Your model needs to keep track of the bees' positions and sizes, and the stars' positions, and it will fire off events so the ViewModel knows whenever there's a change to a bee or a star.




```
using Windows.Foundation;
class Bee {
    public Point Location { get; set; }
    public Size Size { get; set; }
    public Rect Position { get { return new Rect(Location, Size); } }
    public double Width { get { return Position.Width; } }
    public double Height { get { return Position.Height; } }

    public Bee(Point location, Size size) {
        Location = location;
        Size = size;
    }
}
```

```
using Windows.Foundation;
class Star {
    public Point Location {
        get; set;
    }

    public Star(Point location) {
        Location = location;
    }
}
```

Once you get your program working, try adding a Boolean Rotating property to the Star class and use it to make some of your stars slowly spin around.

```
class BeeMovedEventArgs : EventArgs {
    public Bee BeeThatMoved { get; private set; }
    public double X { get; private set; }
    public double Y { get; private set; }

    public BeeMovedEventArgs(Bee beeThatMoved, double x, double y) {
        BeeThatMoved = beeThatMoved;
        X = x;
        Y = y;
    }
}
```

The model will fire events that use these EventArgs to tell the ViewModel when changes happen.

```
class StarChangedEventArgs : EventArgs {
    public Star StarThatChanged { get; private set; }
    public bool Removed { get; private set; }

    public StarChangedEventArgs(Star starThatChanged, bool removed) {
        StarThatChanged = starThatChanged;
        Removed = removed;
    }
}
```

THE POINT, SIZE, AND RECT STUCTS

The Windows.Foundation namespace has several very useful structs. Point uses X and Y double properties to store a set of coordinates. Size has two double properties too, Width and Height, and also a special Empty value. Rect stores two coordinates for the top-left and bottom-right corner of a rectangle. It has a lot of useful methods to find its width, height, intersection with other Rects, and more.

> The Points property on the Polygon control is a collection of Point structs.
> The Rect struct has several overloaded constructors, and methods that let you extract its width, height, size, and location (either as a Point or individual X and Y double coordinates).

- **❾ ADD THE BEESTARMODEL CLASS TO THE MODEL.**
  We've filled in the private fields and a couple of useful methods. Your job is to finish building the `BeeStarModel` class.

```
using Windows.Foundation;

class BeeStarModel {
    public static readonly Size StarSize = new Size(150, 100);

    private readonly Dictionary<Bee, Point> _bees = new Dictionary<Bee, Point>();
    private readonly Dictionary<Star, Point> _stars = new Dictionary<Star, Point>();
    private Random _random = new Random();

    public BeeStarModel() {
        _playAreaSize = Size.Empty;
    }

    public void Update() {
        MoveOneBee();
        AddOrRemoveAStar();
    }

    private static bool RectsOverlap(Rect r1, Rect r2) {
        r1.Intersect(r2);
        if (r1.Width > 0 || r1.Height > 0)
            return true;
        return false;
    }

    public Size PlayAreaSize {
        // Add a backing field, and have the set accessor call CreateBees() and CreateStars()
    }
    private void CreateBees() {
        // If the play area is empty, return. If there are already bees, move each of them.
        // Otherwise, create between 5 and 15 randomly sized bees (40 to 150 pixels), add
        // it to the _bees collection, and fire the BeeMoved event.
    }
    private void CreateStars() {
        // If the play area is empty, return. If there are already stars,
        // set each Star's location to a new point and fire the StarChanged
        // event, otherwise call CreateAStar() between 5 and 10 times.
    }
    private void CreateAStar() {
        // Find a new non-overlapping point, add a new Star object to the
        // _stars collection, and fire the StarChanged event.
    }
    private Point FindNonOverlappingPoint(Size size) {
        // Find the upper left-hand corner of a rectangle that doesn't overlap any bees or stars
        // You'll need to try random Rects, then use LINQ queries to find any bees or stars
        // that overlap (the RectsOverlap() method will be useful).
    }
    private void MoveOneBee(Bee bee = null) {
        // If there are no bees, return. If the bee parameter is null, choose a random bee,
        // otherwise use the bee argument Then find a new non-overlapping point, update the bee's
        // location, update the _bees collection, then fire the OnBeeMoved event.
    }
    private void AddOrRemoveAStar() {
        // Flip a coin (_random.Next(2) == 0) and either create a star using CreateAStar() or
        // remove a star and fire OnStarChanged. Always create a star if there are <= 5, remove
        // one if >= 20. _stars.Keys.ToList()[_random.Next(_stars.Count)] will find a random star.
    }
    // You'll need to add the BeeMoved and StarChanged events and methods to call them.
    // They use the BeeMovedEventArgs and StarChangedEventArgs classes.
}
```

You can use readonly to create a constant struct value.

Size.Empty is a value of Size that's reserved for an empty size. You'll use it to only create bees and stars when the play area is resized.

The ViewModel will use a timer to call this Update() method periodically.

This method checks two Rect structs and returns true if they overlap each other using the Rect.Intersect() method.

PlayAreaSize is a property.

If the method's tried 1,000 random locations and hasn't found one that doesn't overlap, the play area has probably run out of space, so just return any point.

You can debug your app with the simulator to make sure it works with different screen sizes and orientations.

- **⑩ ADD THE BEESTARVIEWMODEL CLASS TO THE VIEWMODEL.**
  Fill in the commented methods. You'll need to look closely at how the Model works, and what the View expects. The helper methods will also come in very handy.

We wanted to make sure that `DispatcherTimer` and `UIElement` are the only classes from the `Windows.UI.Xaml` namespace that we used in the ViewModel. The `using` keyword lets you use = to declare a single member in another namespace.

```
using View;
using Model;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using Windows.Foundation;
using DispatcherTimer = Windows.UI.Xaml.DispatcherTimer;
using UIElement = Windows.UI.Xaml.UIElement;

class BeeStarViewModel {
    private readonly ObservableCollection<UIElement>
                _sprites = new ObservableCollection<UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    private readonly Dictionary<Star, StarControl> _stars = new Dictionary<Star, StarControl>();
    private readonly List<StarControl> _fadedStars = new List<StarControl>();

    private BeeStarModel _model = new BeeStarModel();

    private readonly Dictionary<Bee, AnimatedImage> _bees = new Dictionary<Bee, AnimatedImage>();

    private DispatcherTimer _timer = new DispatcherTimer();

    public Size PlayAreaSize { /* get and set accessors return and set _model.PlayAreaSize */ }

    public BeeStarViewModel() {
        // Hook up the event handlers to the BeeStarModel's BeeMoved and StarChanged events,
        // and start the timer ticking every two seconds.
    }
    void timer_Tick(object sender, object e) {
        // Every time the timer ticks, find all StarControl references in the _fadedStars
        // collection and remove each of them from _sprites, then call the BeeViewModel's
        // Update() method to tell it to update itself.
    }
    void BeeMovedHandler(object sender, BeeMovedEventArgs e) {
        // The _bees dictionary maps Bee objects in the Model to AnimatedImage controls
        // in the view. When a bee is moved, the BeeStarModel fires its BeeMoved event to
        // tell anyone listening which bee moved and its new location. If the _bees
        // dictionary doesn't already contain an AnimatedImage control for the bee, it needs
        // to create a new one, set its canvas location, and update both _bees and _sprites.
        // If the _bees dictionary already has it, then we just need to look up the corresponding
        // AnimatedImage control and move it on the canvas to its new location with an animation.
    }
    void StarChangedHandler(object sender, StarChangedEventArgs e) {
        // The _stars dictionary works just like the _bees one, except that it maps Star objects
        // to their corresponding StarControl controls. The EventArgs contains references to
        // the Star object (which has a Location property) and a boolean to tell you if the star
        // was removed. If it is then we want it to fade out, so remove it from _stars, add it
        // to _fadedStars, and call its FadeOut() method (it'll be removed from _sprites the next
        // time the Update() method is called, which is why we set the timer's tick interval to
        // be greater than the StarControl's fade out animation).
        //
        // If the star is not being removed, then check to see if _stars contains it - if so, get
        // the StarControl reference; if not, you'll need to create a new StarControl, fade it in,
        // add it to _sprites, and send it to back so the bees can fly in front of it. Then set
        // the canvas location for the StarControl.
    }
}
```

When you set the new Canvas location, the control is updated—even if it's already on the Canvas. This is how the stars move themselves around when the play area is resized.

LONG EXERCISE SOLUTION

Here are the filled-in methods in the `BeeStarModel` class.

```
using Windows.Foundation;

class BeeStarModel {
    public static readonly Size StarSize = new Size(150, 100);

    private readonly Dictionary<Bee, Point> _bees = new Dictionary<Bee, Point>();
    private readonly Dictionary<Star, Point> _stars = new Dictionary<Star, Point>();
    private Random _random = new Random();

    public BeeStarModel() {
        _playAreaSize = Size.Empty;
    }

    public void Update() {
        MoveOneBee();
        AddOrRemoveAStar();
    }

    private static bool RectsOverlap(Rect r1, Rect r2) {
        r1.Intersect(r2);
        if (r1.Width > 0 || r1.Height > 0)
            return true;
        return false;
    }

    private Size _playAreaSize;
    public Size PlayAreaSize {
        get { return _playAreaSize; }
        set
        {
            _playAreaSize = value;
            CreateBees();
            CreateStars();
        }
    }

    private void CreateBees() {
        if (PlayAreaSize == Size.Empty) return;

        if (_bees.Count() > 0) {
            List<Bee> allBees = _bees.Keys.ToList();
            foreach (Bee bee in allBees)
                MoveOneBee(bee);
        } else {
            int beeCount = _random.Next(5, 16);
            for (int i = 0; i < beeCount; i++) {
                int s = _random.Next(40, 151);
                Size beeSize = new Size(s, s);
                Point newLocation = FindNonOverlappingPoint(beeSize);
                Bee newBee = new Bee(newLocation, beeSize);
                _bees[newBee] = new Point(newLocation.X, newLocation.Y);
                OnBeeMoved(newBee, newLocation.X, newLocation.Y);
            }
        }
    }
```

We gave these to you.

Here are the methods for the StarControl code-behind:

```
public void FadeIn() {
    fadeInStoryboard.Begin();
}

public void FadeOut() {
    fadeOutStoryboard.Begin();
}
```

Whenever the PlayAreaSize property changes, the Model updates the _playAreaSize backing field and then calls CreateBees() and CreateStars(). This lets the ViewModel tell the Model to adjust itself whenever the size changes—which will happen if you run the program on a tablet and change the orientation.

If there are already bees, move each of them. MoveOneBee() will find a new nonoverlapping location for each bee and fire a BeeMoved event.

If there aren't any bees in the model yet, this creates new Bee objects and sets their locations. Any time a bee is added or changes, we need to fire a BeeMoved event.

```
private void CreateStars() {
    if (PlayAreaSize == Size.Empty) return;

    if (_stars.Count > 0) {
        foreach (Star star in _stars.Keys) {
            star.Location = FindNonOverlappingPoint(StarSize);
            OnStarChanged(star, false);
        }
    } else {
        int starCount = _random.Next(5, 11);
        for (int i = 0; i < starCount; i++)
            CreateAStar();
    }
}

private void CreateAStar() {
    Point newLocation = FindNonOverlappingPoint(StarSize);
    Star newStar = new Star(newLocation);
    _stars[newStar] = new Point(newLocation.X, newLocation.Y);
    OnStarChanged(newStar, false);
}

private Point FindNonOverlappingPoint(Size size) {
    Rect newRect;
    bool noOverlap = false;
    int count = 0;
    while (!noOverlap) {
        newRect = new Rect(_random.Next((int)PlayAreaSize.Width - 150),
            _random.Next((int)PlayAreaSize.Height - 150),
            size.Width, size.Height);

        var overlappingBees =
            from bee in _bees.Keys
            where RectsOverlap(bee.Position, newRect)
            select bee;

        var overlappingStars =
            from star in _stars.Keys
            where RectsOverlap(
                new Rect(star.Location.X, star.Location.Y, StarSize.Width, StarSize.Height),
                newRect)
            select star;

        if ((overlappingBees.Count() + overlappingStars.Count() == 0) || (count++ > 1000))
            noOverlap = true;
    }
    return new Point(newRect.X, newRect.Y);
}

private void MoveOneBee(Bee bee = null) {
    if (_bees.Keys.Count() == 0) return;
    if (bee == null) {
        List<Bee> bees = _bees.Keys.ToList();
        bee = bees[_random.Next(bees.Count)];
    }
    bee.Location = FindNonOverlappingPoint(bee.Size);
    _bees[bee] = bee.Location;
    OnBeeMoved(bee, bee.Location.X, bee.Location.Y);
}
```

If there are already stars, we just set each existing star's location to a new point on the PlayArea and fire the StarChanged event. It's up to the ViewModel to handle that event and move the corresponding control.

This creates a random Rect and then checks if it overlaps. We gave it a 150-pixel gap on the right and 150-pixel gap on the bottom so the stars and bees don't leave the play area.

These LINQ queries call RectsOverlap() to find any bees or stars that overlap the new Rect. If either return value has a count, the new Rect overlaps something.

If this iterated 1,000 times, it means we're probably out of nonoverlapping spots in the play area and need to break out of an infinite loop.

LONG EXERCISE SOLUTION

The last few members of the `BeeStarModel` class.

Flip a coin by choosing either 0 or 1 at random, but always create a star if there are under 5 and remove if 20 or more.

```
    private void AddOrRemoveAStar() {
        if (((_random.Next(2) == 0) || (_stars.Count <= 5)) && (_stars.Count < 20 ))
            CreateAStar();
        else {
            Star starToRemove = _stars.Keys.ToList()[_random.Next(_stars.Count)];
            _stars.Remove(starToRemove);
            OnStarChanged(starToRemove, true);
        }
    }
```

Every time the Update() method is called, we want to either add or remove a star. The CreateAStar() method already creates stars. If we're removing a star, we just remove it from _stars and fire a StarChanged event.

```
    public event EventHandler<BeeMovedEventArgs> BeeMoved;

    private void OnBeeMoved(Bee beeThatMoved, double x, double y)
    {
        EventHandler<BeeMovedEventArgs> beeMoved = BeeMoved;
        if (beeMoved != null)
        {
            beeMoved(this, new BeeMovedEventArgs(beeThatMoved, x, y));
        }
    }

    public event EventHandler<StarChangedEventArgs> StarChanged;

    private void OnStarChanged(Star starThatChanged, bool removed)
    {
        EventHandler<StarChangedEventArgs> starChanged = StarChanged;
        if (starChanged != null)
        {
            starChanged(this, new StarChangedEventArgs(starThatChanged, removed));
        }
    }
}
```

These are typical event handlers and methods to fire them.

Here are the filled-in methods of the `BeeStarViewModel` class.

```
using View;
using Model;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using Windows.Foundation;
using DispatcherTimer = Windows.UI.Xaml.DispatcherTimer;
using UIElement = Windows.UI.Xaml.UIElement;

class BeeStarViewModel {
    private readonly ObservableCollection<UIElement>
                _sprites = new ObservableCollection<UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    private readonly Dictionary<Star, StarControl> _stars = new Dictionary<Star, StarControl>();
    private readonly List<StarControl> _fadedStars = new List<StarControl>();

    private BeeStarModel _model = new BeeStarModel();

    private readonly Dictionary<Bee, AnimatedImage> _bees
                                        = new Dictionary<Bee, AnimatedImage>();

    private DispatcherTimer _timer = new DispatcherTimer();
```

We gave these to you.

If you've done a good job with separation of concerns, your designs often tend to naturally end up being loosely coupled.

The ViewModel's `PlayAreaSize` property just passes through to the property on the Model—but the Model's `PlayAreaSize` set accessor calls methods that fire `BeeMoved` and `StarChanged` events. So when the screen resolution changes: 1) the Canvas fires its `SizeChanged` event, which 2) updates the ViewModel's `PlayAreaSize` property, which 3) updates the Model's property, which 4) calls methods to update bees and stars, which 5) fire `BeeMoved` and `StarChanged` events, which 6) trigger the ViewModel's event handlers, which 7) update the `Sprites` collection, which 8) updates the controls on the Canvas. This is an example of loose coupling, where there's no single, central object to coordinate things. This is a very stable way to build software because each object doesn't need to have explicit knowledge of how the other objects work. It just needs to know one small job: handle an event, fire an event, call a method, set a property, etc.

```
public Size PlayAreaSize {
    get { return _model.PlayAreaSize; }
    set { _model.PlayAreaSize = value; }
}

public BeeStarViewModel() {
    _model.BeeMoved += BeeMovedHandler;
    _model.StarChanged += StarChangedHandler;

    _timer.Interval = TimeSpan.FromSeconds(2);
    _timer.Tick += timer_Tick;
    _timer.Start();
}

void timer_Tick(object sender, object e) {
    foreach (StarControl starControl in _fadedStars)
        _sprites.Remove(starControl);

    _model.Update();
}

void BeeMovedHandler(object sender, BeeMovedEventArgs e) {
    if (!_bees.ContainsKey(e.BeeThatMoved)) {
        AnimatedImage beeControl = BeeStarHelper.BeeFactory(
                    e.BeeThatMoved.Width, e.BeeThatMoved.Height, TimeSpan.FromMilliseconds(20));
        BeeStarHelper.SetCanvasLocation(beeControl, e.X, e.Y);
        _bees[e.BeeThatMoved] = beeControl;
        _sprites.Add(beeControl);
    } else {
        AnimatedImage beeControl = _bees[e.BeeThatMoved];
        BeeStarHelper.MoveElementOnCanvas(beeControl, e.X, e.Y);
    }
}

void StarChangedHandler(object sender, StarChangedEventArgs e) {
    if (e.Removed) {
        StarControl starControl = _stars[e.StarThatChanged];
        _stars.Remove(e.StarThatChanged);
        _fadedStars.Add(starControl);
        starControl.FadeOut();
    } else {
        StarControl newStar;
        if (_stars.ContainsKey(e.StarThatChanged))
            newStar = _stars[e.StarThatChanged];
        else {
            newStar = new StarControl();
            _stars[e.StarThatChanged] = newStar;
            newStar.FadeIn();
            BeeStarHelper.SendToBack(newStar);
            _sprites.Add(newStar);
        }
        BeeStarHelper.SetCanvasLocation(
                    newStar, e.StarThatChanged.Location.X, e.StarThatChanged.Location.Y);
    }
}
}
```

The _fadedStars collection contains the controls that are currently fading and will be removed the next time the ViewModel's Update() method is called.

If a star is being added, it needs to have its FadeIn() method called. If it's already there, it's just being moved because the play area size changed. Either way, we want to move it to its new location on the Canvas.

Congratulations! (But you're not done yet...)

Did you finish that last exercise? Did you understand everything that was going on? If so, then **congratulations** — you've learned a whole lot of C#, and probably in less time than you'd expected! The world of programming awaits you.

Still, there are a few things that you should do before you move on to the last lab, if you really want to make sure all the information you put in your brain stays there.

Take one last look through Save the Humans.

If you did everything we asked you to do, you've built *Save the Humans* twice, once at the beginning of the book and again before you started Chapter 10. Even the second time around, there were parts of it that seemed like magic. But when it comes to programming, **there is no magic**. So take one last pass through the code that you built. You'll be surprised at how much you understand! There's almost nothing that seals a lesson into your brain like positive reinforcement.

Talk about it with your friends.

Humans are social animals, and when you talk through things you've learned with your social circle you do a better job of retaining them. And these days, "talking" means social networking, too! Plus, you've really accomplished something here. Go ahead and **claim your bragging rights**!

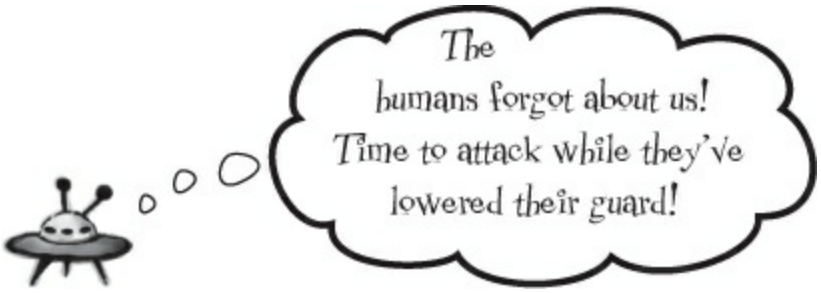
Take a break. Even better, take a nap.

Your brain has absorbed a lot of information, and sometimes the best thing you can do to "lock in" all that new knowledge is to sleep on it. There's a lot of neuroscience research that shows that information absorption is significantly improved **after a good night's sleep**. So give your brain a well-deserved rest!

When it comes to programming, there is no magic. Every program works because it was built to work, and all code can be understood.

NOTE

...but it's a lot easier to understand code if the programmer used good design patterns and object-oriented programming principles.



Part III. C# Lab Invaders

Name:	_____	Date:	_____

This lab gives you a spec that describes a program for you to build, using the knowledge you've gained throughout this book.

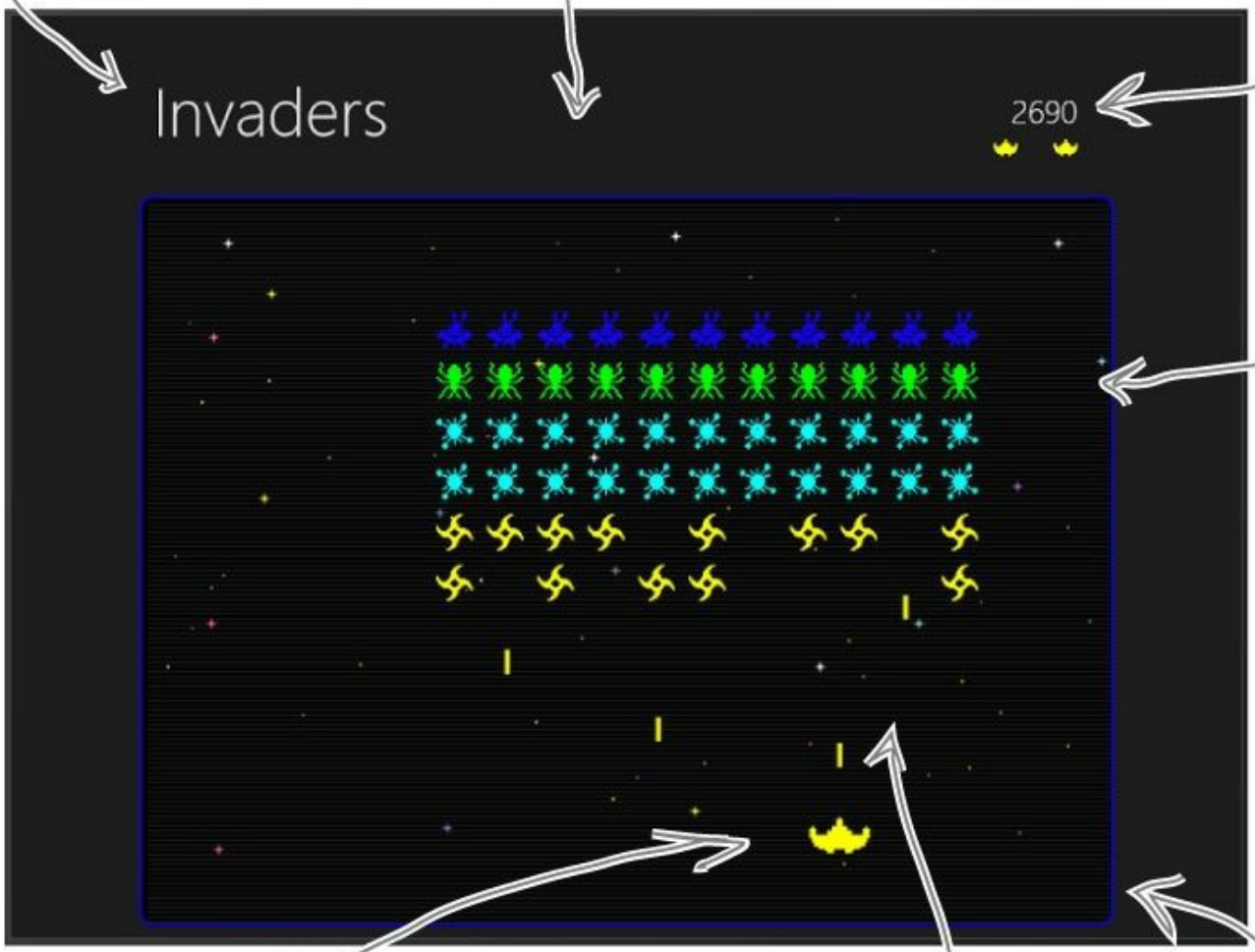
This project is bigger than the ones you've seen so far. So read the whole thing before you get started, and give yourself a little time. If you did all of the exercises throughout the book, you have all of the tools that you need to do this lab.

We've filled in a few design details for you, and we've made sure you've got all the pieces you need...and nothing else.

It's up to you to finish the job. You can get our version of the finished Invaders game from the Windows Store as an **open source project**.

The grandfather of video games

In this lab you'll pay homage to one of the most popular, revered, and replicated icons in video game history, a game that needs no further introduction: **it's time to build *Invaders*!**



Invaders is a Windows Store app using a single Basic Page.

The invaders attack in waves of 11 columns with six invaders. The first wave moves slowly and fires a few shots at a time. The next wave moves faster, and fires more shots more frequently. If all invaders in a wave are destroyed, the next wave attacks.

As the player destroys the invaders, the score goes up. It's displayed in the upper-righthand corner.

The player starts out with three ships. The first ship is in play, and the other two are kept in reserve. His spare ships are shown underneath the score.

The invaders' ships are animated with blocky, pixelated, retro 80s-style graphics. The whole playing area has a 4:3 aspect ratio just like an old arcade cabinet, complete with simulated scan lines to make it look authentic.

The player moves the ship left and right, and fires shots at the invaders. If a shot hits an invader, the invader is destroyed and the player's score goes up.

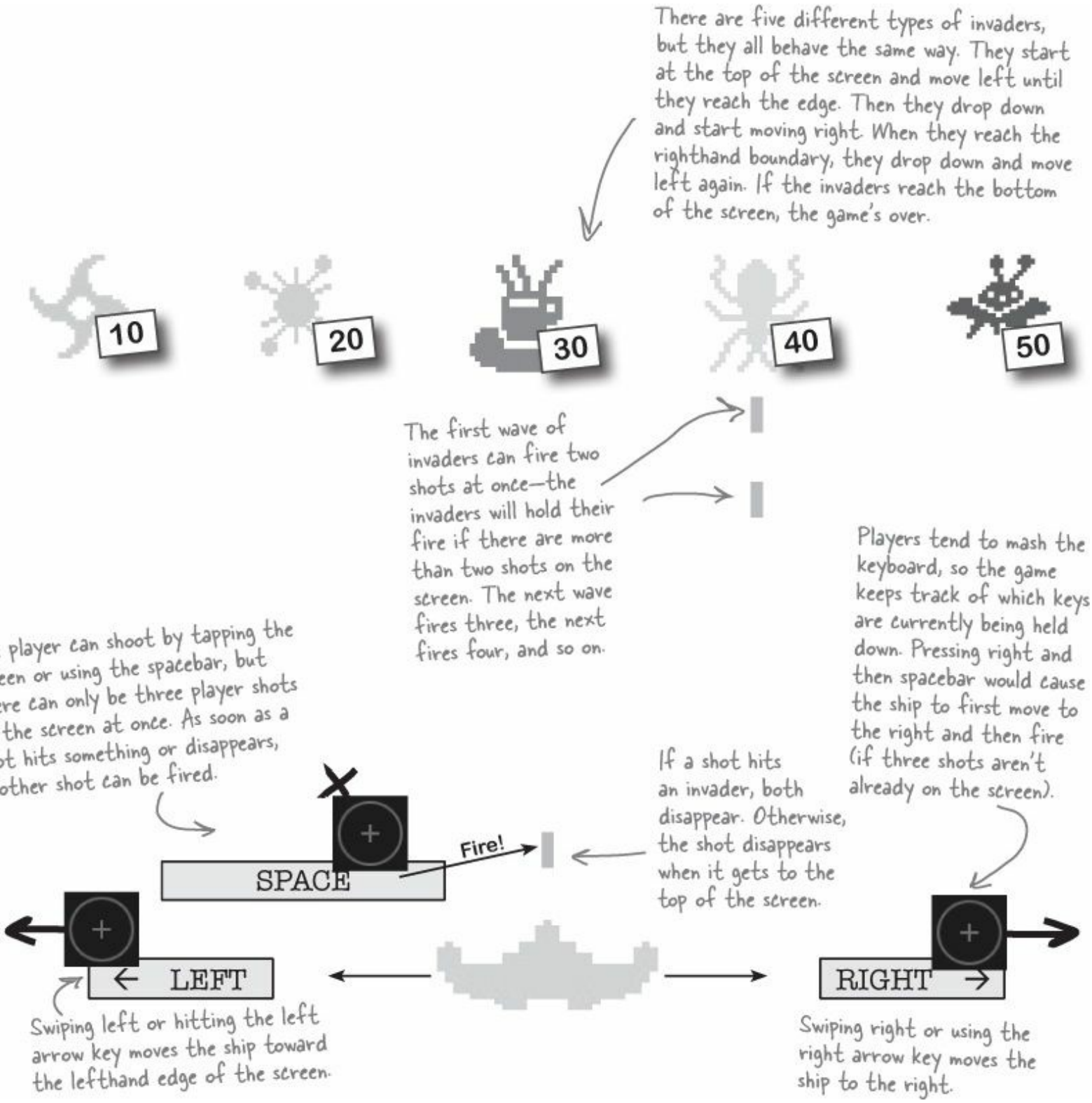
The invaders return fire. If one of the shots hits the ship, the player loses a life. Once all lives are gone, or if the invaders reach the bottom of the screen, the game ends and a big "GAME OVER" is displayed in the middle of the screen.

The multicolored stars in the background twinkle on and off, but don't affect gameplay at all.

Your mission: defend the planet against wave after wave of invaders

The invaders attack in waves, and each wave is a tight formation of 66 individual invaders. As the player destroys invaders, his score goes up. The bottom two rows of invaders are shaped like stars and worth 10 points. The satellites are worth 20, the saucers are worth 30, the bugs are worth 40, and those pesky alien invader spaceships that have been invading Earth since Chapter 1 are worth 50. The player starts with three lives. If he loses all three lives or the invaders reach the bottom of the screen, the game's over.

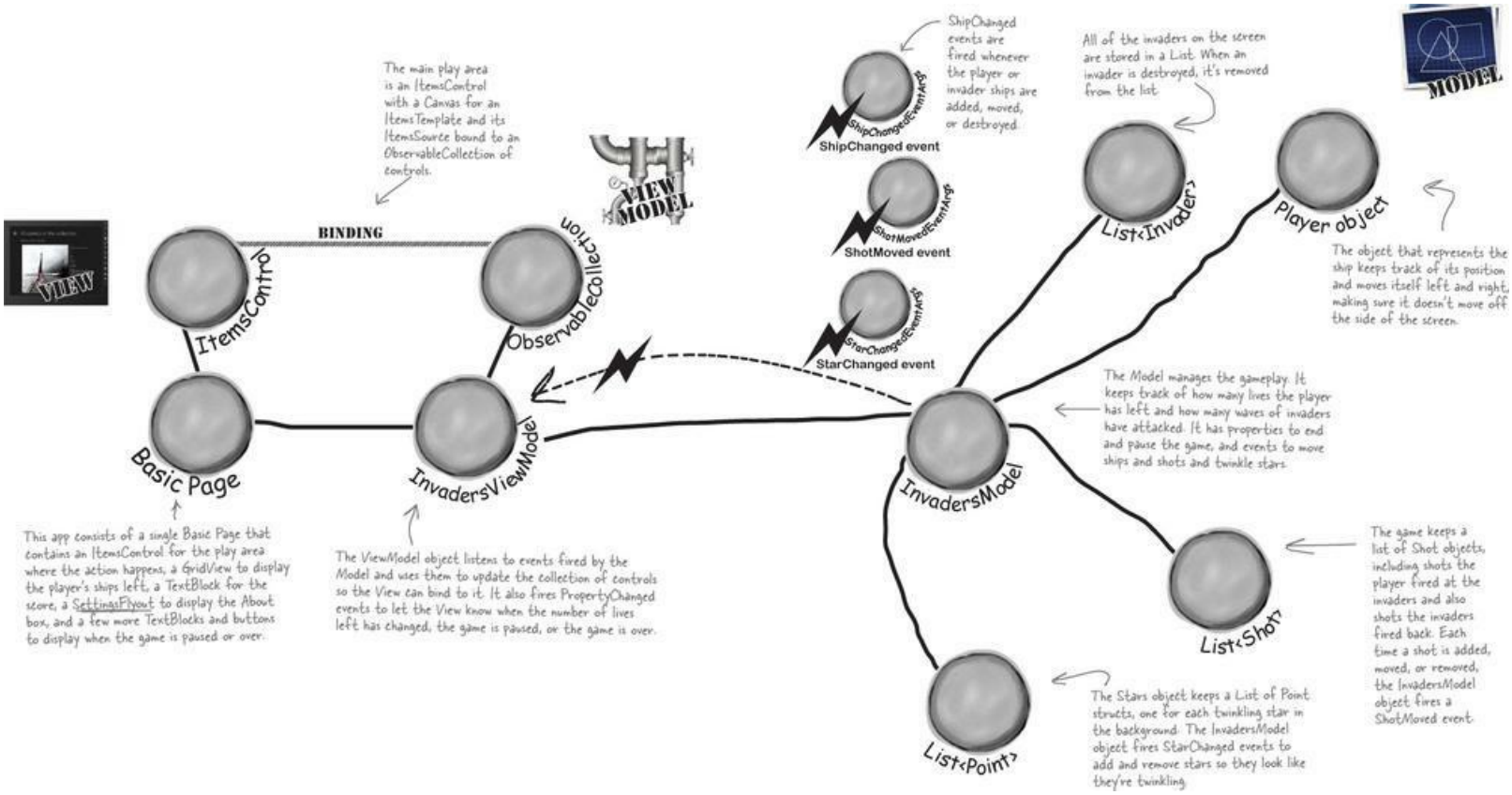
Remember, you can download the Invaders graphics from headfirstlabs.com/hfcsharp



The architecture of Invaders

Invaders is an MVVM app. The Model needs to keep track of a wave of invaders (including their location, type, and score value), the player's ship, shots that the player and invaders fire at each other, and stars in the background. The View uses a Basic Page and controls for animated images and stars, as well as a static helper class to help the ViewModel.

Here's an overview of what you'll need to create:



NOTE

You haven't seen flyouts before.

An important part of programming is figuring out how to use tools that you haven't seen before. You'll use a button `Flyout`, or a window that temporarily pops up when you press a button, and is dismissed when you click anywhere else. And you'll use a `SettingsFlyout`, or a window opens up from the side of the page the user chooses About from the Settings charm. This is a good chance to test your developer skills!

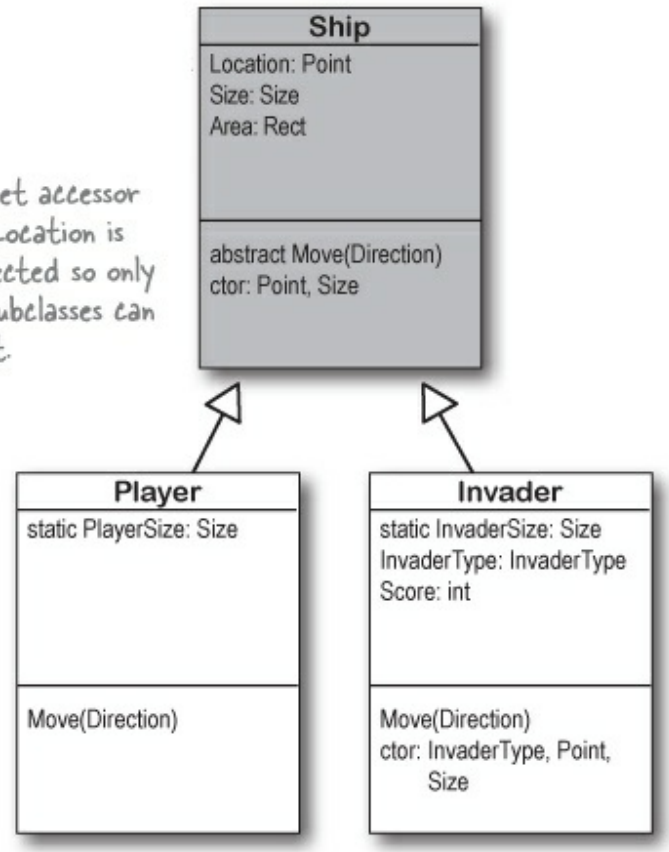
Build out the object model for the Model

Before you can build out the `InvadersModel` class, you'll need the classes that it uses to keep track of the gameplay. It's going to have an object for the player, and collections of invaders, shots, and stars. That means it'll need classes for invaders and shots (it'll use a `Point` struct for each star, because all it needs to know is the star's location).

The `Player` and `Invader` classes **extend an abstract class called `Ship`** that has properties (set in the constructor) to keep track of its location and size. It's also got a convenient property that uses the location and size to create a `Rect`, which can be used for collision detection. You'll need to implement these two subclasses.

Here's the abstract `Ship` class for the *Model* folder:

Collision detection means discovering when two moving sprites have bumped into each other.



The set accessor for Location is protected so only the subclasses can set it.

```
using Windows.Foundation;

abstract class Ship {
    public Point Location { get; protected set; }

    public Size Size { get; private set; }

    public Rect Area {
        get { return new Rect(Location, Size); }
    }

    public Ship(Point location, Size size) {
        Location = location;
        Size = size;
    }

    public abstract void Move(Direction direction);
}
```


The Player moves left and right.

The Model will call a `Player` object's `Move()` method to tell it to move left or right, using a `Direction` enum to tell it which way it's moving. The Player can't move off the end of the screen. It can use the `InvadersModel`'s static `PlayAreaSize` property to stop moving when it hits the side of

the play area. You'll also need a static read-only `Size` for the `Player`'s size (25 × 15 pixels) and `const` double for its speed (10 pixels per `Move()` call).

Invaders move left, right, and down.

The `Invader` and `Player` classes both have a `Move()` method that uses a switch statement to determine which way to move. The `Invader` class also has an additional constructor that takes parameters to set its `InvaderType` and `Score` properties. These properties determine which graphic is displayed on the page, and how many points get added to the score when the ship is destroyed.

You'll need this `Shot` class.

The Model uses it to keep track of the shots that the player fires, and the shots the invaders fire back. Have a close look at the `Move()` method: it uses a private `DateTime` field to keep track of the last time it moved. Each time `Move()` is called, the shot's `Location` is moved either up or down at a velocity of 95 pixels per second.

You'll also need these three `EventArgs` classes, which the Model uses to let the ViewModel know when stars appear and disappear; when shots move, appear, and disappear; and when ships move and die. When a player or invader fires a shot, the Model will create a `Shot` object, and then fire a `ShotMoved` event. The ViewModel will handle this event and update its `Sprites` collection, which will notify the View that it changed.

```
using Windows.Foundation;

class Shot {
    public const double ShotPixelsPerSecond = 95;

    public Point Location { get; private set; }
    public static Size ShotSize = new Size(2, 10);

    public Direction Direction { get; private set; }

    private DateTime _lastMoved;

    public Shot(Point location, Direction direction) {
        Location = location;
        Direction = direction;

        _lastMoved = DateTime.Now;
    }

    public void Move() {
        TimeSpan timeSinceLastMoved = DateTime.Now - _lastMoved;
        double distance = timeSinceLastMoved.Milliseconds
            * ShotPixelsPerSecond / 1000;
        if (Direction == Direction.Up) distance *= -1;
        Location = new Point(Location.X, Location.Y + distance);
        _lastMoved = DateTime.Now;
    }
}
```

You can speed up or slow down the shots by changing the pixel speed.

Can you figure out a way to change ShotSize so it scales up and down when the page size changes?

```
using Windows.Foundation;

class StarChangedEventArgs : EventArgs {
    public Point Point { get; private set; }
    public bool Disappeared { get; private set; }

    public StarChangedEventArgs(Point point,
                                bool disappeared) {
        Point = point;
        Disappeared = disappeared;
    }
}
```

```
using Windows.Foundation;

class ShotMovedEventArgs : EventArgs {
    public Shot Shot { get; private set; }
    public bool Disappeared { get; private set; }

    public ShotMovedEventArgs(Shot shot, bool disappeared) {
        Shot = shot;
        Disappeared = disappeared;
    }
}
```

This enum determines which type of ship an invader is flying.

```
enum InvaderType {
    Bug,
    Saucer,
    Satellite,
    Spaceship,
    Star,
}
```

```
using Windows.Foundation;

class ShipChangedEventArgs : EventArgs {
    public Ship ShipUpdated { get; private set; }
    public bool Killed { get; private set; }

    public ShipChangedEventArgs(Ship shipUpdated, bool killed) {
        ShipUpdated = shipUpdated;
        Killed = killed;
    }
}
```

Ships and shots use this enum to determine which direction they're moving.

```
enum Direction {
    Left,
    Right,
    Up,
    Down,
}
```

Building the InvadersModel class

The `InvadersModel` class controls the Invaders game. Here's a start on what this class should look like — there's still lots of work for you to do.

```
using Windows.Foundation;

class InvadersModel {
    public readonly static Size PlayAreaSize = new Size(400, 300);
    public const int MaximumPlayerShots = 3;
    public const int InitialStarCount = 50;

    private readonly Random _random = new Random();

    public int Score { get; private set; }
    public int Wave { get; private set; }
    public int Lives { get; private set; }

    public bool GameOver { get; private set; }

    private DateTime? _playerDied = null;
    public bool PlayerDying { get { return _playerDied.HasValue; } }

    private Player _player;

    private readonly List<Invader> _invaders = new List<Invader>();
    private readonly List<Shot> _playerShots = new List<Shot>();
    private readonly List<Shot> _invaderShots = new List<Shot>();
    private readonly List<Point> _stars = new List<Point>();

    private Direction _invaderDirection = Direction.Left;
    private bool _justMovedDown = false;

    private DateTime _lastUpdated = DateTime.MinValue;

    public InvadersModel() {
        EndGame();
    }

    public void EndGame() {
        GameOver = true;
    }

    // You'll need to finish the rest of the InvadersModel class
}
```

When the player dies, the ViewModel makes the player's ship flash for 2.5 seconds. The model uses a DateTime? to keep track of this time and prevents the ships or shots from moving while the player is dying.

The InvadersModel methods

The `InvadersModel` class has five public methods that are used by the ViewModel. The `EndGame()` method is on the facing page — here are the rest:

- **❶ THE STARTGAME() METHOD STARTS THE GAME PLAYING.**
  This method sets the `GameOver` property to `false`. Then it clears any invaders from the `_invaders` collection and shots from the `_playerShots` and `_invaderShots` collections (but before it does, it fires a `ShipChanged` or `ShotMoved` event for each of them). Then it clears the stars (firing the `StarChanged` event for each star) and creates new stars. Finally, it creates a new `Player` object (firing a `ShipChanged` event), sets Lives to 2, Wave to 0, and adds the first wave.
- **❷ THE FIRESHOT() METHOD MAKES THE PLAYER FIRE A SHOT.**
  This method checks the number of player shots on screen to make sure there aren't too many, then it adds a new `Shot` to the `_playerShots` collection and fires the `ShotMoved` event.
- **❸ THE MOVEPLAYER() METHOD MOVES THE PLAYER.**
  If the player has already died, this does nothing; otherwise, it calls the `Player` object's `Move()` method and then fires the `ShipChanged` event to let the ViewModel know the ship moved.
- **❹ THE TWINKLE() METHOD TWINKLES THE STARS.**
  This method flips a coin and either adds or removes a star, firing the `StarChanged` event. There are always fewer than 50% more and greater than 15% fewer than the initial number of stars.
- **❺ THE UPDATE() METHOD MAKES THE GAME GO.**
  The ViewModel uses a timer to call the `Update()` method many times a second as long as the game isn't over — this is what keeps advancing the gameplay. First it checks to see if the game is paused. If it isn't, here's what it does (it always twinkles the stars, whether or not the game is paused):

  If there are no more invaders, it creates the next wave.
  If the player hasn't died, it moves each invader (more about this on the next page).
  Then every shot needs to be updated (unless the player is dead). The game needs to loop through both shot collections, calling each shot's `Move()` method. If any shot went off the edge of the play area, it's removed from the collection and a `ShotMoved` event is fired.
  The invaders return fire (more about this on the next page too).
  Finally, it checks for collisions: first for any shot that overlaps an invader (and removing both from their collections), and then to see if the player's been shot. This is where that `Rect` property on the `Ship` base class will come in very handy — you can use the method that checks for overlapping `Rects` from Chapter 16 to detect the collisions (more on the next page).

Here's a tip: If you try to remove an object from a collection while you're enumerating though it using `foreach`, it'll throw an exception. But you can use the LINQ `ToList()` extension method to make a copy of the collection first and loop through that instead.

Filling out the InvadersModel class

The problem with class diagrams is that they usually leave out any nonpublic properties and methods. So even after you've got the methods from the previous page done, you've still got a lot of work to do. Here are some things to think about:

The game play happens on a 400x300 battlefield

The first line in the `InvadersModel` class creates a public `Size` field called `PlayAreaSize`. It's static and read-only, which means it can't change throughout the life of the `InvadersModel`. This defines the boundaries of the play area for all of the Model objects: the shots can use it to determine when they've reached the top or the bottom of the play area, and the invader and player ships can use it to determine when they've hit the sides. The objects in the View will typically move around a Canvas that's larger than 400×300, so part of the ViewModel's job will be to scale all of the coordinates up so that they're moved to the right place.

> **NOTE**
>
> It's the ViewModel's job to translate the Model's coordinates on a 400x300 play area to whatever size the Canvas happens to be on the page.

Build a NextWave() method

A simple method to create the next wave of invaders will come in handy. It should increment the `Wave` property, clear the private `_invaders` collection, and then create all of the `Invader` objects, giving each of them a `Location` field with the correct coordinates. Try spacing them out so that they're spaced 1.4 invader lengths apart horizontally, and 1.4 invader heights vertically.

NOTE

Here's an example of a private method that will really help out your ViewModel.

A few other ideas for private methods

Here are a few of the private method ideas you might play with, to see if these would also help the design of your `Game` class:

- A method to see if the player's been hit (`CheckForPlayerCollisions()`)
- A method to see if any invaders have been hit (`CheckForInvaderCollisions()`)
- A method to move all the invaders (`MoveInvaders()`)
- A method allowing invaders to return fire (`ReturnFire()`)

NOTE

The invaders move individually from side to side. When they get to the edge of the battlefield, they move down. A method to move all invaders calls each invader's Move() method. It can use the _lastUpdated field to speed up the invaders by reducing the time between marches as the number of invaders left in the formation shrinks.

BRAIN POWER

It's possible to show protected and private properties and methods in a class diagram, but you'll rarely see that put into practice. Why do you think that is?

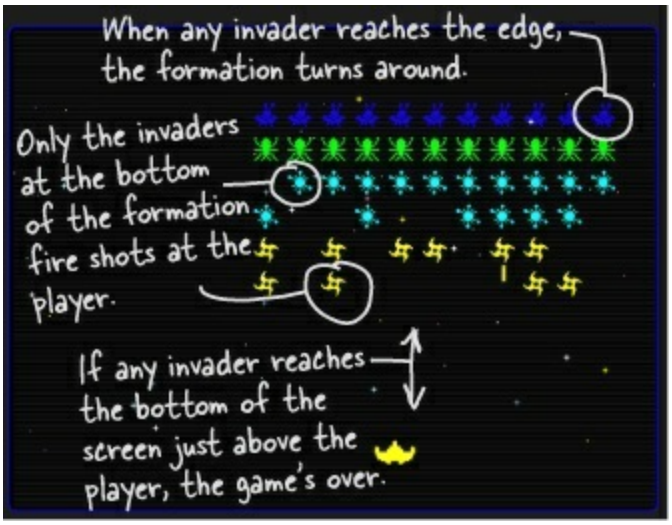
LINQ makes collision detection much easier

NOTE

The next page may seem a bit complex when you first read it, but each LINQ query is just a couple of lines of code. Here's a hint: don't overcomplicate it!

You've got collections of invaders and shots, and you need to search through those collections to find certain invaders and shots. Any time you hear *collections* and search in the same sentence, you should think LINQ. Here's what you need to do:

- **❶ FIGURE OUT IF THE INVADERS' FORMATION HAS REACHED THE EDGE.**
  The invaders need to change direction if any one invader is within twice its horizontal move interval from the edge of the battlefield. When the invaders are marching to the right, once they reach the righthand side of the play area, the game needs to tell them to drop down and start marching to the left. And when the invaders are marching to the left, the game needs to check if they've reached the left edge. To make this happen, add a private `MoveInvaders()` method that gets called by `Update()`. The first thing it should do is calculate the amount of time since the last movement using the `_lastUpdated` field, and do nothing if not enough time has passed, check and update the private `framesSkipped` field. If the invaders are moving to the right, `MoveInvaders()` should use LINQ to search the `_invaders` collection for any invader whose location's X value is within range of the righthand boundary. If it finds any, then it should tell the invaders to march downward and then set `invaderDirection` equal to `Direction.Left`; if not, it can tell each invader to march to the right. On the other hand, if the invaders are moving to the left, then it should do the opposite, using another LINQ query to see if the invaders are near the lefthand boundary, marching them down and changing direction if they are. It can use the `_justMovedDown` field to keep track of when the formation just switched direction and marched down.
- **❷ DETERMINE WHICH INVADERS CAN RETURN FIRE.**
  Add a private method called `ReturnFire()` that gets called by `Update()`. First, it should `return` if the invaders' shot list already has `wave + 1` shots. It should also `return` if `_random.Next(10) < 10 - Wave`. (That makes the invaders fire at random, and not all the time.) If it gets past both tests, it can use LINQ to group the invaders by their `Location.X` and sort them `descending`. Once it's got those groups, it can choose a group at random, and use its `Last()` method to find the invader at the bottom of the column. All right, now you've got the shooter — you can add a shot to `_invaderShots` list just below the middle of the invader (use the invader's `Area` to find the shot's location).



- **❸ CHECK FOR INVADER AND PLAYER COLLISIONS.**
  You'll want to create a method to check for collisions. There are three collisions to check for, and the method to find overlapping `Rects` from Chapter 16 will come in handy.

  Use LINQ to find any dead invaders by looping through the shots in the player's shot list and selecting any invader where `Area` contains the shot's location. Remove the invader and the shot. Add a query to figure out if any invaders reached the bottom of the battlefield — if so, end the game.

  You don't need LINQ to look for shots that collided with the player, just a loop and the player's `Area` property. (Remember, **you can't modify a collection inside a `foreach` loop**. If you do, you'll get an `InvalidOperationException` with a message that the collection was modified. You may need to create a temporary List of objects to remove, or use the `ToList()` extension method to copy it first.)

Build the Invaders page for the View

The main page for Invaders is a Basic Page that lives in the *View* folder. It has a ViewModel object as a static resource, which is used for the `DataContext` for all of the controls on the page.

All action is handled with binding.

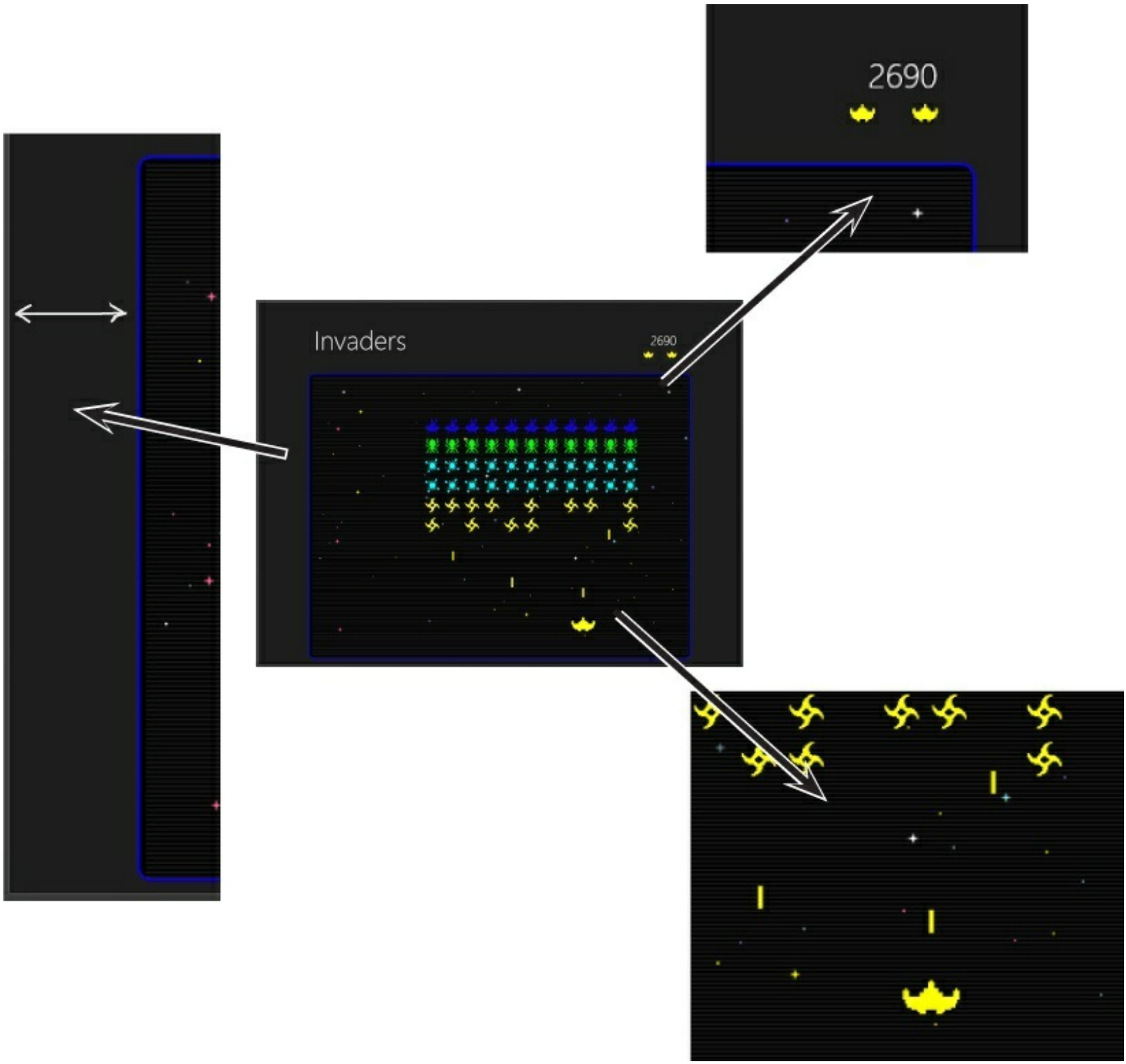
The invaders, player ship, shots, stars, and even the simulated scan lines are all controls that are added to an `ObservableCollection` of controls in the ViewModel. You'll also need **a TextBlock with the text "Game Over"** with its `Visibility` bound to the `GameOver` property, and another with the text "Paused" bound to the `Paused` property.

The score and extra lives are separate controls.

There's a StackPanel in the upper-righthand corner with a TextBlock bound to the `Score` property and a GridView bound to the `Lives` property. The GridView displays ships because **its `DataTemplate` is an `Image` control**, so the `Lives` property in the ViewModel needs to be a collection of objects — `new object()` — to make the GridView add or remove images.

The play area is always resized to keep a 4:3 aspect ratio.

The main play area is a `Border` with rounded corners that contains an `ItemsControl` with the `ItemsSource` bound to the `Sprites` property, and whose `ItemsPanel` is a `Canvas` with a black background. We'll give you code on the next page that updates its margins to make sure it always keeps a **4:3 aspect ratio** — it will modify the `Margin` property of the `Border` to keep the `Height` 4/3 the size of the `Width`, even if the screen is rotated or resized.


2690
Invaders
2690

Maintain the play area's aspect ratio

The code-behind for the main page needs to do two things. It has to handle events when the page resizes to **maintain the play area's 4:3 aspect ratio**, and it needs to handle both keyboard and swipe input. If the player is using a tablet, rotating it will change the play area size. So you'll need to handle a few events in your page root's XAML code:

```
<Page
    x:Name="pageRoot"
    x:Class="(the namespace you used).MainPage"
    ...
    xmlns:viewmodel="using:(the namespace you used).ViewModel"

    SizeChanged="pageRoot_SizeChanged"

    ManipulationMode="TranslateX" ManipulationCompleted="pageRoot_ManipulationCompleted"
    ManipulationDelta="pageRoot_ManipulationDelta" Tapped="pageRoot_Tapped"
    >
```

You'll need these Manipulation and Tapped events to handle input, which we'll get to on the next page.

and in the `Border` around the play area:

```
<Border x:Name="playArea" BorderBrush="Blue" BorderThickness="2" CornerRadius="10"
        Background="Black" Margin="5" Grid.Row="1" Loaded="playArea_Loaded">
    <ItemsControl
     ...
```

Here's the code-behind that keeps the play area's 4:3 aspect ratio by adding either left and right margins or top and bottom margins.

```
private void playArea_Loaded(object sender, RoutedEventArgs e) {
    UpdatePlayAreaSize(playArea.RenderSize);
}

private void pageRoot_SizeChanged(object sender, SizeChangedEventArgs e) {
    UpdatePlayAreaSize(new Size(e.NewSize.Width, e.NewSize.Height - 160));
}

private void UpdatePlayAreaSize(Size newPlayAreaSize) {
    double targetWidth;
    double targetHeight;
    if (newPlayAreaSize.Width > newPlayAreaSize.Height) {
        targetWidth = newPlayAreaSize.Height * 4 / 3;
        targetHeight = newPlayAreaSize.Height;
        double leftRightMargin = (newPlayAreaSize.Width - targetWidth) / 2;
        playArea.Margin = new Thickness(leftRightMargin, 0, leftRightMargin, 0);
    } else {
        targetHeight = newPlayAreaSize.Width * 3 / 4;
        targetWidth = newPlayAreaSize.Width;
        double topBottomMargin = (newPlayAreaSize.Height - targetHeight) / 2;
        playArea.Margin = new Thickness(0, topBottomMargin, 0, topBottomMargin);
    }
    playArea.Width = targetWidth;
    playArea.Height = targetHeight;
    viewModel.PlayAreaSize = new Size(targetWidth, targetHeight);
}
```

The UpdatePlayAreaSize() method calculates the new height and width, changes the controls, and then updates the ViewModel's PlayAreaSize property.

Respond to swipe and keyboard input

Your game will need to be able to respond to the user pressing keys and swiping a touchscreen to control the player ship. And since this is an MVVM app, there's an important separation of concerns. It's the page's job to keep track of the keypresses, swipes, and taps, and let the ViewModel know when they happen. It's the ViewModel's job to interpret them as game actions and call the appropriate methods on the Model.

Keyboard event handlers are added in code-behind

Override the `OnNavigatedTo()` and `OnNavigatedFrom()` methods (like you did in Chapter 14) to add and remove event handlers for the `KeyUp` and `KeyDown` events, calling methods on the ViewModel to interpret the keystrokes. (You'll need to add `using Windows.UI.Core;` to use the KeyEventArgs class.)

```
protected override void OnNavigatedTo(NavigationEventArgs e) {
    Window.Current.CoreWindow.KeyDown += KeyDownHandler;
    Window.Current.CoreWindow.KeyUp += KeyUpHandler;
    base.OnNavigatedTo(e);
}

protected override void OnNavigatedFrom(NavigationEventArgs e) {
    Window.Current.CoreWindow.KeyDown -= KeyDownHandler;
    Window.Current.CoreWindow.KeyUp -= KeyUpHandler;
    base.OnNavigatedFrom(e);
}

private void KeyDownHandler(object sender, KeyEventArgs e) {
    viewModel.KeyDown(e.VirtualKey);
}

private void KeyUpHandler(object sender, KeyEventArgs e) {
    viewModel.KeyUp(e.VirtualKey);
}
```

Window.Current.CoreWindow gives you a reference to a CoreWindow object that has events for basic UI behaviors like keypresses. This makes sure your keypresses are always handled by the event handler.

Add page root event handlers for swipes and taps

You'll need to handle left and right swipes to move the player ship, and taps to fire. The event handlers were hooked up in the XAML on the previous page, so now you just need to add the event handlers.

```
private void pageRoot_ManipulationDelta(object sender, ManipulationDeltaRoutedEventArgs e) {
    if (e.Delta.Translation.X < -1)
        viewModel.LeftGestureStarted();
    else if (e.Delta.Translation.X > 1)
        viewModel.RightGestureStarted();
}
```

The ManipulationDelta event constantly fires during a swipe as the user's finger moves, and the e.Delta.Translation tells you how far the user moved since the last firing.

```
private void pageRoot_ManipulationCompleted(object sender, ManipulationCompletedRoutedEventArgs e) {
    viewModel.LeftGestureCompleted();
    viewModel.RightGestureCompleted();
}
```

The ManipulationCompleted event fires when the user's finger is lifted. The ViewModel will decide how to deal with these events.

```
bool firstTapOfGame = false;
private void pageRoot_Tapped(object sender, TappedRoutedEventArgs e) {
    if (!firstTapOfGame)
        viewModel.Tapped();

    firstTapOfGame = false;
}
```

We're "swallowing" the first tap of the game to prevent the ship from firing when the player hits the Start Game button. Make sure you reset firstTapOfGame to false when the game starts.

An AnimatedImage control displays the ships

You can use the same `AnimatedImage` control that you used in Chapter 16 to display both the invader and player ships. The player ship only has a single, nonanimated image, so you can pass it an image list with one image (this gives you options to animate it later if you want).

When the invader ships are hit, they should fade out rather than disappearing entirely. And anyone who's played 80s arcade games knows that when the player ship is hit, it should flash for 2.5 seconds before the game resumes. So you'll need to add these methods to the `AnimatedImage` control's code-behind:

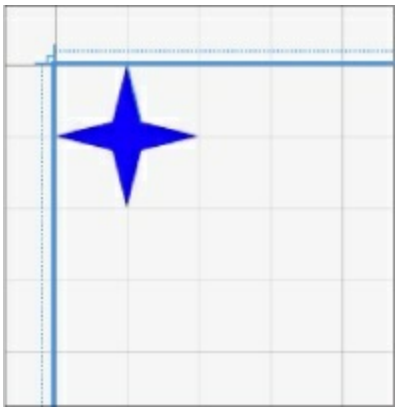
```
public void InvaderShot() {
    invaderShotStoryboard.Begin();
}

public void StartFlashing() {
    flashStoryboard.Begin();
}

public void StopFlashing() {
    flashStoryboard.Stop();
}
```

And you'll need to add the appropriate storyboards as well. The `invaderShotStoryboard` is a `DoubleAnimation` that fades the `Opacity` property from 1 to 0. `flashStoryboard` is a key frame animation that toggles `Visibility` to make the control disappear and reappear.

Add a control for the big stars

The starry background has three types of stars: circles, rectangles, and big stars. The big stars are still pretty small — just 10 pixels by 10 pixels. So you'll need to create a user control that has a `Polygon`. The stars can be different colors, so your control will need a public method to change the color of the `Polygon`:

```
public void SetFill(SolidColorBrush solidColorBrush) {
    polygon.Fill = solidColorBrush;
}
```



A static InvadersHelper class helps the ViewModel

The ViewModel could use a helper class with factory methods for the invader, player ship, shot, and star controls. The `StarControlFactory()` method should pick a random number and return either a rectangle, ellipse, or big star. You can also add a private method to return a color at random (`return Colors.LightBlue;`) so `StarControlFactory()` can return different stars with different colors.

You'll also need a `ScanLineFactory()` method to create the simulated scan lines. Each scan line is a rectangle with `Fill` set to `new SolidColorBrush(Colors.White)`, `Height` set to 2, and `Opacity` set to .1.

All of the factory methods should take a double `scale` parameter, which we'll talk about with the ViewModel.

Use the Settings charm to open a SettingsFlyout

In Chapter 15 you learned how to add a callback for the About command in the Settings charm. Your job now is to figure out how to add a SettingsFlyout to your page. Here's the code-behind to hook it up to the Settings charm:

```
public InvadersPage() {
    this.InitializeComponent();

    SettingsPane.GetForCurrentView().CommandsRequested
                              += InvadersPage_CommandsRequested;
}
void InvadersPage_CommandsRequested(SettingsPane sender,
                    SettingsPaneCommandsRequestedEventArgs args) {
    UICommandInvokedHandler invokedHandler =
              new UICommandInvokedHandler(AboutInvokedHandler);
    SettingsCommand aboutCommand = new SettingsCommand(
                     "About", "About Invaders", invokedHandler);
    args.Request.ApplicationCommands.Add(aboutCommand);
}
private void AboutInvokedHandler(IUICommand command) {
    viewModel.Paused = true;
    AboutSettingsFlyout aboutSettingsFlyout = new
AboutSettingsFlyout();
    aboutSettingsFlyout.Unloaded += aboutSettingsFlyout_Unloaded;
    aboutSettingsFlyout.Show();
}
void aboutSettingsFlyout_Unloaded(object sender, RoutedEventArgs e) {
    viewModel.Paused = false;
}
```

Choose Add→New Item... and add a Settings Flyout called AboutSettingsFlyout to your project. Change the Title property to "About Invaders" and modify the XAML to display the flyout contents.

You should also add a button to the main page that displays a Flyout whenever it's clicked. You don't need any code-behind or a Click event handler for this. All you need is a <Button.Flyout> section that

contains the XAML for the flyout window that you want to display. Here's some XAML to get you started:

```
<Button Content="Learn to build this app" Background="Black"
        HorizontalAlignment="Center" FontSize="30" >
    <Button.Flyout>
        <Flyout Placement="Full">
           <StackPanel Width="400" VerticalAlignment="Center">
              <!-- XAML for your button flyout goes here -->
        </Flyout>
    </Button.Flyout>
</Button>
```

You can see your button's flyout in the designer by clicking inside the <Button.Flyout> XAML.

We've been pointing you to MSDN pages throughout this book, because they're a great way to learn more. You can learn more about the Flyout and SettingsFlyout controls here: http://msdn.microsoft.com/en-us/library/windows/apps/bg182878.aspx

Build the ViewModel

The ViewModel has two classes in it. `InvadersViewModel` is the main ViewModel object, and `BooleanVisibilityConverter` is the same as the one you used in Chapter 16 — you can use it to bind the "Game Over" and "Paused" TextBlock controls' `Visible` properties to the `GameOver` and `Paused` properties on the ViewModel. So the rest of this lab is all about building out the ViewModel.

Here's the top of the `InvadersViewModel` class, to get you started:

```
using View;
using Model;
using System.ComponentModel;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using Windows.Foundation;
using DispatcherTimer = Windows.UI.Xaml.DispatcherTimer;
using FrameworkElement = Windows.UI.Xaml.FrameworkElement;

class InvadersViewModel : INotifyPropertyChanged {
    private readonly ObservableCollection<FrameworkElement>
                _sprites = new ObservableCollection<FrameworkElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    public bool GameOver { get { return _model.GameOver; } }

    private readonly ObservableCollection<object> _lives =
                                        new ObservableCollection<object>();
    public INotifyCollectionChanged Lives { get { return _lives; } }

    public bool Paused { get; set; }
    private bool _lastPaused = true;

    public static double Scale { get; private set; }

    public int Score { get; private set; }

    public Size PlayAreaSize {
        set {
            Scale = value.Width / 405;
            _model.UpdateAllShipsAndStars();
            RecreateScanLines();
        }
    }

    private readonly InvadersModel _model = new InvadersModel();
    private readonly DispatcherTimer _timer = new DispatcherTimer();
    private FrameworkElement _playerControl = null;
    private bool _playerFlashing = false;
    private readonly Dictionary<Invader, FrameworkElement> _invaders =
                                    new Dictionary<Invader, FrameworkElement>();
    private readonly Dictionary<FrameworkElement, DateTime> _shotInvaders =
                                    new Dictionary<FrameworkElement, DateTime>();
    private readonly Dictionary<Shot, FrameworkElement> _shots =
                                    new Dictionary<Shot, FrameworkElement>();
    private readonly Dictionary<Point, FrameworkElement> _stars =
                                    new Dictionary<Point, FrameworkElement>();
    private readonly List<FrameworkElement> _scanLines =
                                    new List<FrameworkElement>();
```

This is the same pattern you used in Chapter 16 to manage the sprites for the "Bees on a Starry Night" project: creating a private, read-only ObservableCollection field of controls, and only exposing an INotifyCollectionChanged property to encapsulate it from the View.

The `Scale` property is a multiplier that, when multiplied with any `X`, `Y`, `Width`, and `Height` value, translates that value from the 400×300 model coordinates to the correct coordinate Canvas control in the play area.

The `PlayAreaSize` field only has a set accessor, and it's updated by the View whenever the play area size changes. When the `PlayAreaSize` property is set, the set accessor calculates a new `Scale` multiplier, then tells the model to fire events to update all of its ships and stars. It also recreates the scan lines.

Handling user input

You already saw how the main page in the View calls methods in the ViewModel to handle keypresses, swipes, and taps. Here are the methods that it calls. The user needs to be able to use the keyboard or touch screen interchangeably. To accomplish this, both the tap and spacebar cause the ViewModel to call the Model's `FireShot()` method. Moving the player's ship left and right is a little more complex: both keypresses and swipes update `DateTime`? fields that contain the date of the most recent keypress or swipe, or are `null` if there is no current keypress or swipe.

Did you notice how all of the methods on this page have the `internal` access modifier? That's because we added these methods by first adding the code from a few pages earlier, and then using the Generate Method Stub feature of the IDE to create the method declarations. The `internal` modifier means the method is publicly accessible from inside this assembly, but appears as private to other assemblies. You can learn more about assemblies in leftover #3 in the appendix.

```
private DateTime? _leftAction = null;
private DateTime? _rightAction = null;
```

The ViewModel uses Nullable<DateTime> fields to keep track of the most recent left or right action triggered by a keypress or a gesture.

```
internal void KeyDown(Windows.System.VirtualKey virtualKey) {
    if (virtualKey == Windows.System.VirtualKey.Space)
        _model.FireShot();

    if (virtualKey == Windows.System.VirtualKey.Left)
        _leftAction = DateTime.Now;

    if (virtualKey == Windows.System.VirtualKey.Right)
        _rightAction = DateTime.Now;
}
```

The View calls the KeyDown method from the page's key event handler, passing it in the key that was pressed. If the user hit the spacebar the ViewModel tells the Model to fire a shot. If the player pressed the left or right arrow, the _leftAction or _rightAction field gets upated.

```
internal void KeyUp(Windows.System.VirtualKey virtualKey) {
    if (virtualKey == Windows.System.VirtualKey.Left)
        _leftAction = null;

    if (virtualKey == Windows.System.VirtualKey.Right)
        _rightAction = null;
}
```

If the player hit the left or right key, it sets the appropriate field to null.

This lets the ViewModel's timer event handler figure out if it should move the player ship by checking the _leftAction and _rightAction fields.

```
internal void LeftGestureStarted() {
    _leftAction = DateTime.Now;
}

internal void LeftGestureCompleted() {
    _leftAction = null;
}

internal void RightGestureStarted() {
    _rightAction = DateTime.Now;
}

internal void RightGestureCompleted() {
    _rightAction = null;
}

internal void Tapped() {
    _model.FireShot();
}
```

The View calls these methods in its event handlers for the user swipe and tap gestures. When the user swipes left or right it updates the _leftAction or _rightAction field, and if the user taps it fires a shot.

Can you figure out how to pause the game when the player hits the P key?

Build the InvadersViewModel methods

We'll get you started with a constructor and two useful methods.

THE INVADERSVIEWMODEL CONSTRUCTOR HOOKS UP THE INVADERSMODEL EVENT HANDLERS AND ENDS THE GAME.

```
public InvadersViewModel() {
    Scale = 1;

    _model.ShipChanged += ModelShipChangedEventHandler;
    _model.ShotMoved += ModelShotMovedEventHandler;
    _model.StarChanged += ModelStarChangedEventHandler;

    _timer.Interval = TimeSpan.FromMilliseconds(100);
    _timer.Tick += TimerTickEventHandler;

    EndGame();
}
```

When the Scale property is set to 1, the ViewModel will update the View with a 1-to-1 scale 400x300 battlefield—but the View will quickly update the ViewModel's PlayAreaSize, which updates the Scale property.

Ticking every 100 milliseconds updates the View 10 times a second. That's not the same thing as a frame rate, because we're using animations to move sprites around.

End the game so it loads with the Game Over screen.

THE STARTGAME() METHOD CLEARS THE INVADERS AND SHOTS FROM THE SPRITES COLLECTION, TELLS THE MODEL TO START THE GAME, AND STARTS THE TIMER.

```
public void StartGame(){
    Paused = false;
    foreach (var invader in _invaders.Values) _sprites.Remove(invader);
    foreach (var shot in _shots.Values) _sprites.Remove(shot);
    _model.StartGame();
    OnPropertyChanged("GameOver");
    _timer.Start();
}
```

When the Model starts the game, it updates its GameOver property, so the ViewModel fires a PropertyChanged event to update the View.

THE RECREATESCANLINES() METHOD ADDS SIMULATED SCAN LINES.

```
private void RecreateScanLines(){
    foreach (FrameworkElement scanLine in _scanLines)
        if (_sprites.Contains(scanLine))
            _sprites.Remove(scanLine);
    _scanLines.Clear();
    for (int y = 0; y < 300; y += 2) {
        FrameworkElement scanLine = InvadersHelper.ScanLineFactory(y, 400, Scale);
        _scanLines.Add(scanLine);
        _sprites.Add(scanLine);
    }
}
```

You'll need to build this factory method.

The factory method uses this argument to scale the rectangles up to the proper size and location.

The View's updated when the timer ticks

When the `InvadersModel` fires a `ShipChanged` event, the ViewModel needs to figure out what kind of ship changed, and update its collections appropriately so that the View accurately reflects the current state of the Model. Here's how the `ShipChanged` event handler works:

```
void TimerTickEventHandler(object sender, object e) {
   if (_lastPaused != Paused)
   {
      Use the _lastPaused field to fire a PropertyChanged event any time
      the Paused property changes.
   }
   if (!Paused)
   {
      If both the _leftAction and _rightAction fields have a value, that means
      there are either two keys being mashed or a key and a swipe at the same
      time — use the one with the later time to choose a direction to move the player.
      If not, choose the one with a value and use that to pass to _model.MovePlayer().
   }

   Tell the InvadersModel to update itself. Then check the Score property. If it
   doesn't match _model.Score, update it and fire a PropertyChanged event.

   Update the Lives so that it matches _model.Lives by either removing an
   object or adding a new object().

   foreach (FrameworkElement control in _shotInvaders.Keys.ToList())
   {

      Each key in the _shotInvaders Dictionary is an AnimatedImage control,
      and its value is the time that it died. It takes half a second for the invader
      fade-out animation to complete, so any invader who died more than
      half a second ago should be removed from both _sprites and _shotInvaders.
   }

   If the game is over, fire a PropertyChanged event and stop the timer.
}
```

The player's ship can move and die

When the `InvadersModel` fires a `ShipChanged` event, the ViewModel needs to figure out what kind of ship changed, and update its collections appropriately so that the View accurately reflects the current state of the Model. Here's how the `ShipChanged` event handler works:

```
void ModelShipChangedEventHandler(object sender, ShipChangedEventArgs e) {
    if (!e.Killed) {
        if (e.ShipUpdated is Invader) {
            Invader invader = e.ShipUpdated as Invader;
```

You'll need to downcast e.ShipUpdated to the appropriate class, either Invader or PlayerShip.

If the _invaders collection doesn't contain this invader, use the InvadersControlFactory() method to create a new control and add it to the collection, and to the sprites. Otherwise, move the invader control to its correct location and resize it—and don't forget to pass in the Scale value! Here's a helpful method that you might want to add to InvadersHelper:

```
                InvadersHelper.ResizeElement(invaderControl, invader.Size.Width * Scale,
                                             invader.Size.Height * Scale);
```

```
        } else if (e.ShipUpdated is Player) {
```

If the _playerFlashing field is true, then the player ship is currently flashing because it previously died; stop it from flashing. Then check if _playerControl is null. If it is, use PlayerControlFactory() to create a player and add it to the sprites; otherwise, move and resize the player ship.

```
    } else {
        if (e.ShipUpdated is Invader) {
```

If the invader isn't null, call its InvaderShot() method (you'll need to look up its control in the _invaders dictionary, then cast it to an AnimatedImage). Then add the invader to the _shotInvaders dictionary and remove it from _invaders. The _shotInvaders dictionary contains the time that each invader was shot. The ViewModel doesn't remove the invader's AnimatedImage control from the sprites until it's finished fading out.

```
        } else if (e.ShipUpdated is Player) {
```

Cast _playerControl to the AnimatedImage, start it flashing, and set the _playerFlashing field to true. The flashing animation can keep going until the ViewModel gets another ShipChanged event from the Model, because that means gameplay has resumed.

```
        }
    }
}
```

“Shots fired!”

The `InvadersViewModel's` event handlers for the `ShotMoved` and `StarChanged` events are pretty similar.

```
void ModelShotMovedEventHandler(object sender, ShotMovedEventArgs e) {
    if (!e.Disappeared)
    {
       If the shot is not a key in the _shots Dictionary, use its factory method to
       create a new shot control, then add it to _shots and _sprites. If it is in the
       _shots Dictionary, then it's already on screen, so look up its control and
       use the helper method to move it using its Location property.
    } else {
       The shot disappeared, so check _shots to see if it's there. If it is, remove its
       control from _sprites, and remove the Shot object from _shots.
    }
}

void ModelStarChangedEventHandler(object sender, StarChangedEventArgs e) {
    if (e.Disappeared && _stars.ContainsKey(e.Point))
    {
        Look up the control in _stars and remove it from _sprites.
    } else {
        if (!_stars.ContainsKey(e.Point))
        {
            Use the factory method to create a new control, add it to _stars (using
            the Point from the EventArgs as the key), and add it to the sprites.
        } else {
            Stars typically won't change locations, so this else clause probably won't
            get hit — but you can use it to add shooting stars if you
            want. Look up the star's control in _stars and use a
            helper method to move it to the new location.
        }
    }
}
```

And yet there's more to do...

Think the game's looking pretty good? You can take it to the next level with a few more additions:

Add sounds

The MediaElement XAML tag lets you add sounds to your apps. Can you figure out how to use it to add sounds when the invaders march, the player fires shots, and ships are destroyed? This page will help:

http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh465160.aspx

Add a mothership

Once in a while, a mothership worth 250 points can travel across the top of the battlefield. If the player hits it, he gets a bonus.

Add shields

Add floating shields the player can hide behind. You can add simple shields that the enemies and player can't shoot through. Then, if you really want your game to shine, add breakable shields that the player and invaders can blast holes through after a certain number of hits.

> **NOTE**
>
> Each shield can consist of lots of little blocks that disappear just like invaders when they're hit, but don't add to the score.

Add divebombers

Create a special type of enemy that divebombs the player. A divebombing enemy should break formation, take off toward the player, fly down around the bottom of the screen, and then resume its position.

Add more weapons

Start an arms race! Smart bombs, lasers, guided missiles...there are all sorts of weapons that both the player and the invaders can use to attack each other. See if you can add three new weapons to the game.

Add a Preferences command to the Settings charm

You can add a Preferences command to the Settings charm just like you did with the About command to open a second pop up that lets you turn scan lines on and off, change the number of lives, mute sounds, etc.

This is your chance to show off! Did you come up with a cool new version of the game? Publish your Invaders code on CodePlex or another project hosting website, then claim your bragging rights on the Head First C# forum: www.headfirstlabs.com/books/hfcsharp/

Appendix A. Leftovers: The top 10 things we wanted to include in this book

The fun's just beginning!

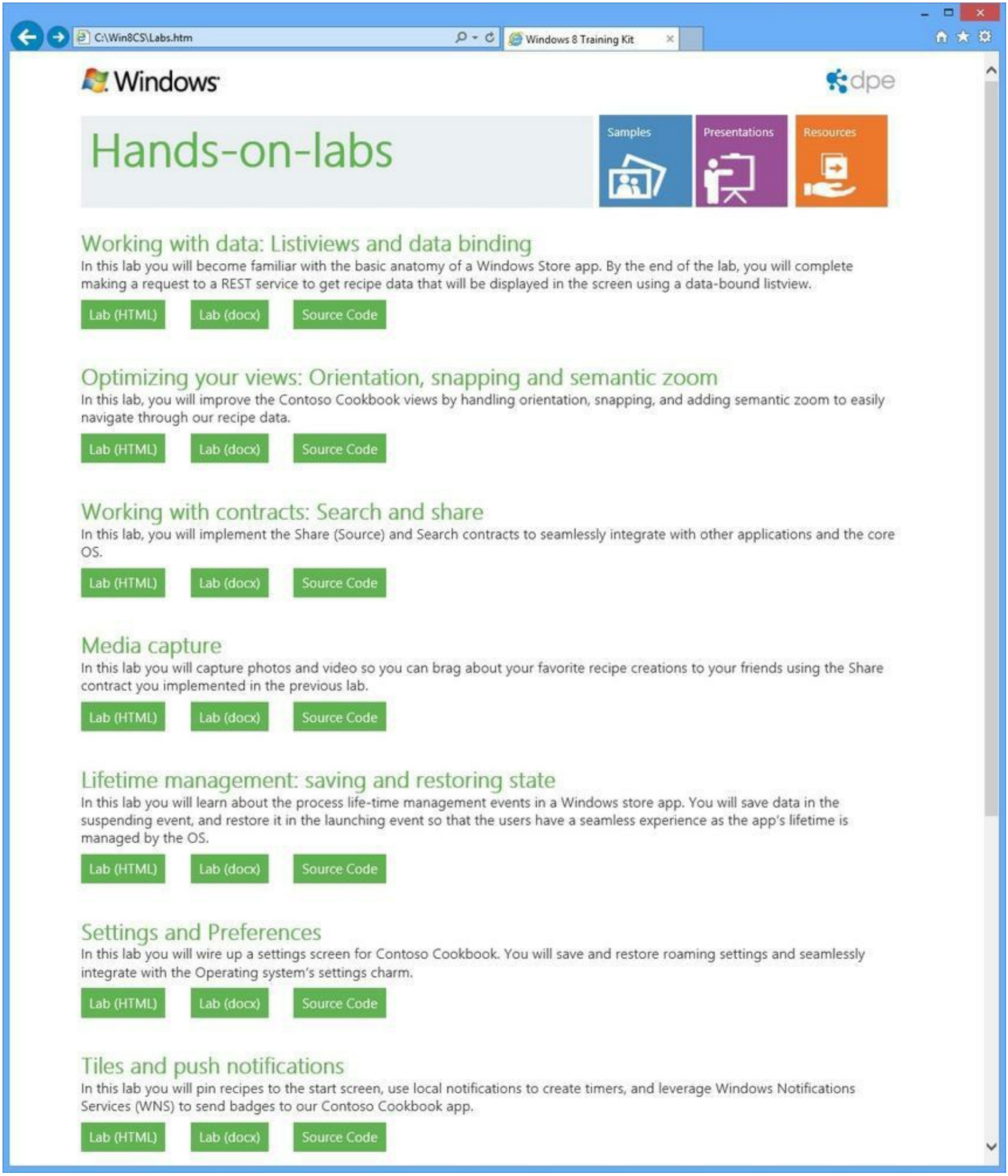
We've shown you a lot of great tools to build some really **powerful software** with C#. But there's no way that we could include **every single tool, technology, or technique** in this book — there just aren't enough pages. We had to make some *really tough choices* about what to include and what to leave out. Here are some of the topics that didn't make the cut. But even though we couldn't get to them, we still think that they're **important and useful**, and we wanted to give you a small head start with them.

#1. There's so much more to Windows Store

Looking to learn more about programming Windows Store apps? Microsoft has some fantastic resources to help you learn. The first step is downloading the **Windows 8 Camp Training Kit**, which has presentations, samples, links to really useful resources, and most importantly, ***a set of hands-on labs*** that teach you about everything from capturing data from a device's camera to adding live tiles and push notifications to your apps. You can download the installer for the Windows 8 Camp Training Kit here:

http://www.microsoft.com/en-us/download/details.aspx?id=29854

Once you install it, you'll get a set of web pages, media files, and presentations, as well as documentation and source code for the hands-on labs. It's a great next step for continuing to get C# concepts into your brain.


C:\Win8CS\Labs.htm
Windows 8 Training Kit
Windows
dpe
Hands-on-labs
Samples
Presentations
Resources
Working with data: Listviews and data binding
In this lab you will become familiar with the basic anatomy of a Windows Store app. By the end of the lab, you will complete making a request to a REST service to get recipe data that will be displayed in the screen using a data-bound listview.
Lab (HTML)
Lab (docx)
Source Code
Optimizing your views: Orientation, snapping and semantic zoom
In this lab, you will improve the Contoso Cookbook views by handling orientation, snapping, and adding semantic zoom to easily navigate through our recipe data.
Lab (HTML)
Lab (docx)
Source Code
Working with contracts: Search and share
In this lab, you will implement the Share (Source) and Search contracts to seamlessly integrate with other applications and the core OS.
Lab (HTML)
Lab (docx)
Source Code
Media capture
In this lab you will capture photos and video so you can brag about your favorite recipe creations to your friends using the Share contract you implemented in the previous lab.
Lab (HTML)
Lab (docx)
Source Code
Lifetime management: saving and restoring state
In this lab you will learn about the process life-time management events in a Windows store app. You will save data in the suspending event, and restore it in the launching event so that the users have a seamless experience as the app's lifetime is managed by the OS.
Lab (HTML)
Lab (docx)
Source Code
Settings and Preferences
In this lab you will wire up a settings screen for Contoso Cookbook. You will save and restore roaming settings and seamlessly integrate with the Operating system's settings charm.
Lab (HTML)
Lab (docx)
Source Code
Tiles and push notifications
In this lab you will pin recipes to the start screen, use local notifications to create timers, and leverage Windows Notifications Services (WNS) to send badges to our Contoso Cookbook app.
Lab (HTML)
Lab (docx)
Source Code

#2. The Basics

Before we get started, here's a `Guy` class that we'll be using throughout this appendix. Take a look at how it's commented. Notice how the class, its methods, and its properties are all commented with triple-slash (///) comments? Those are called **XML comments**, and the IDE will help you add them. Just type "///" right before a class, method, property, or field declaration (and a few other places, too), and the IDE will fill in the skeleton of the XML comment for it. Then later, when you go to use the property, method, etc., the IDE will display information from the XML comments in its IntelliSense window.

> **NOTE**
>
> We wish we could give this material the same kind of thorough treatment we were able to provide throughout the book, but we just didn't have enough pages to do it! But we still want to give you a good starting point and a place to go for more information.

```
/// <summary>
/// A guy with a name, age and a wallet full of bucks
/// </summary>
class Guy
{
    /*
     * Notice how Name and Age are properties with backing fields that are
     * marked readonly. That means those backing fields can only be set when
     * the object is initialized (in their declarations or in the constructor).
     */

    /// <summary>
    /// Read-only backing field for the Name property
    /// </summary>
    private readonly string name;

    /// <summary>
    /// The name of the guy
    /// </summary>
    public string Name { get { return name; } }

    /// <summary>
    /// Read-only backing field for the Age property
    /// </summary>
    private readonly int age;

    /// <summary>
    /// The guy's age
    /// </summary>
    public int Age { get { return age; } }

    /*
     * Cash is not readonly because it might change during the life of the Guy.
     */

    /// <summary>
    /// The number of bucks the guy has
    /// </summary>
    public int Cash { get; private set; }
```

The XML comment for a class consists of a <summary> block. Notice how it starts with <summary> and ends with </summary>.

Marking a field readonly is a useful tool for encapsulation, because it means that field can never be changed once the object is instantiated.

You'll learn about the readonly keyword in Chapter 16, but we want to make sure you know what you're looking at in case you flip to this earlier in the book.

When the IDE adds the skeleton for a constructor or another method, it adds <param> tags for each of the parameters.

```
    /// <summary>
    /// The constructor sets the name, age and cash
    /// </summary>
    /// <param name="name">The name of the guy</param>
    /// <param name="age">The guy's age</param>
    /// <param name="cash">The amount of cash the guy starts with</param>
    public Guy(string name, int age, int cash) {
        this.name = name;
        this.age = age;
        Cash = cash;
    }

    public override string ToString() {
        return String.Format("{0} is {1} years old and has {2} bucks", Name, Age, Cash);
    }

    /// <summary>
    /// Give cash from my wallet
    /// </summary>
    /// <param name="amount">The amount of cash to give</param>
    /// <returns>The amount of cash I gave, or 0 if I don't have enough cash</returns>
    public int GiveCash(int amount) {
        if (amount <= Cash && amount > 0)
        {
            Cash -= amount;
            return amount;
        }
        else
        {
            return 0;
        }
    }

    /// <summary>
    /// Receive some cash into my wallet
    /// </summary>
    /// <param name="amount">Amount to receive</param>
    /// <returns>The amount of cash received, or 0 if no cash was received</returns>
    public int ReceiveCash(int amount) {
        if (amount > 0)
        {
            Cash += amount;
            return amount;
        }
        Console.WriteLine("{0} says: {1} isn't an amount I'll take", Name, amount);
        return 0;
    }
}
```

Here's where we're overriding ToString(). This is covered in Chapter 8.

...more basics...

It's easy to get overwhelmed when learning any computer language, and C# is no exception. That's why we concentrated on the parts of the language that, in our experience, are most common for novice and intermediate developers. But there's some basic C# and .NET syntax that's really useful, but are a lot easier to approach at your own speed once you're used to things. Here's a console application that demonstrates some of it.

```
static void Main(string[] args)
{
    // We'll use these Guy and Random instances throughout this example.
    Guy bob = new Guy("Bob", 43, 100);
    Guy joe = new Guy("Joe", 41, 100);
    Random random = new Random();
```

A really good way to get a handle on this is to debug through it and use watches to see what's happening. As you go through the book, try experimenting with some of these concepts.

A lot of people say that jump statements are bad practice. There are typically other ways that you can achieve the same results. But it's useful to know how they work in case you run across them.

```
    /*
     * Here are two useful keywords that you can use with loops. The "continue" keyword
     * tells the loop to jump to the next iteration of a loop, and the "break" keyword
     * tells the loop to end immediately.
     *
     * The break, continue, throw, and return statements are called "jump statements"
     * because they cause your program to jump to another place in the code when they're
     * executed. (You learned about break with switch/case statements in Chapter 8, and
     * the throw statement in Chapter 10.) There's one more jump statement, goto, which
     * jumps to a label. (You'll recognize these labels as having very similar syntax
     * to what you use in a case statement.)
     *
     * You could easily write this next loop without continue and break. That's a good
     * example of how C# lets you do the same thing many different ways. That's why you
     * don't need break, continue, or any of these other keywords or operators to write
     * any of the programs in this book.
     *
     * The break statement is also used with "case", which you can see in Chapter 8.
     */

    while (true) {
        int amountToGive = random.Next(20);

        // The continue keyword jumps to the next iteration of a loop
        // Use the continue keyword to only give Joe amounts over 10 bucks
        if (amountToGive < 10)
            continue;

        // The break keyword terminates a loop early
        if (joe.ReceiveCash(bob.GiveCash(amountToGive)) == 0)
            break;

        Console.WriteLine("Bob gave Joe {0} bucks, Joe has {1} bucks, Bob has {2} bucks",
            amountToGive, joe.Cash, bob.Cash);
    }
    Console.WriteLine("Bob's left with {0} bucks", bob.Cash);
```

The continue statement causes the program to jump over the rest of the iteration and back to the top of the loop.

The break statement causes the loop to end, and the program to move to the Console.WriteLine() statement.

```
// The ?: conditional operator is an if/then/else collapsed into a single expression
// [boolean test] ? [statement to execute if true] : [statement to execute if false]
Console.WriteLine("Bob {0} more cash than Joe",
    bob.Cash > joe.Cash ? "has" : "does not have");

// The ?? null coalescing operator checks if a value is null, and either returns
// that value if it's not null, or the value you specify if it is
// [value to test] ?? [value to return if it's null]
bob = null;
Console.WriteLine("Result of ?? is '{0}'", bob ?? joe);
```

Since bob is null, the ?? operator returns joe instead.

```
// Here's a loop that uses goto statements and labels. It's rare to see them, but
// they can be useful with nested loops. (The break statement only breaks out of
// the innermost loop)
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 3; j++)
    {
        if (i > 3)
            goto afterLoop;
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
afterLoop:
```

The goto statement causes execution to jump directly to a label.

A label is a string of letters, numbers, or underscores, followed by a colon.

```
// When you use the = operator to make an assignment, it returns a value that you
// can turn around and use in an assignment or an if statement
int a;
int b = (a = 3 * 5);
Console.WriteLine("a = {0}; b = {1};", a, b);
```

This statement first sets a to 3 * 5, and then sets b to the result.

```
// When you put the ++ operator before a variable, it increments the variable
// first, and then executes the rest of the statement.
a = ++b * 10;
Console.WriteLine("a = {0}; b = {1};", a, b);
```

++b means that b is incremented first, and a is set to b * 10.

```
// Putting it after the variable executes the statement first and then increments
a = b++ * 10;
Console.WriteLine("a = {0}; b = {1};", a, b);
```

b++ means that first a is set to b * 10, and then b is incremented.

```
/*
 * When you use && and || to do logical tests, they "short-circuit" -- which means
 * that as soon as the test fails, they stop executing. When (A || B) is being
 * evaluated, if A is true then (A || B) will always be true no matter what B is.
 * And when (A && B) is being evaluated, then if A is false then (A && B) will always
 * be false no matter what B is. In both of those cases, B will never get executed
 * because the operator doesn't need its value in order to come up with a return value.
 */
```

When you use /* and */ to add comments, you don't have to add a * at the beginning of each line, but it makes them easier to read.

```
int x = 0;
int y = 10;
int z = 20;
```

We'll use these values in the code on the next page!

Using the logical "or" and "and" operators' **short-circuiting** properties is another way you can effectively write an if/else statement. This is the same as saying "only execute (y / x == 4) if (y < z) is true.

```
// y / x will throw a DivideByZeroException because x is 0. But since (y < z) is true,
// the || operator knows it will be true without ever having to execute the other
// statement, so it short-circuits and never executes (y / x == 4)
if ((y < z) || (y / x == 4))
    Console.WriteLine("this line printed because || short-circuited");

// Since (y > z) is false, the && operator knows it will return false without
// executing the other statement, so it short-circuits and doesn't throw the exception
if ((y > z) && (y / x == 4))
    Console.WriteLine("this line will never print because && short-circuited");
```

```
/*
 * A lot of us think of 1's and 0's when we think of programming, and manipulating
 * those 1's and 0's is what logic operators are all about.
 */

// Use Convert.ToString() and Convert.ToInt32() to convert a number to or from a
// string of 1's and 0's in its binary form. The second argument specifies that you're
// converting to base 2.
string binaryValue = Convert.ToString(217, 2);
int intValue = Convert.ToInt32(binaryValue, 2);
Console.WriteLine("Binary {0} is integer {1}", binaryValue, intValue);

// The &, |, ^, and ~ operators are logical AND, OR, XOR, and bitwise complement
int val1 = Convert.ToInt32("100000001", 2);
int val2 = Convert.ToInt32("001010100", 2);
int or = val1 | val2;
int and = val1 & val2;
int xor = val1 ^ val2;
int not = ~val1;
```

The logical operators &, |, and ^ are built-in on all the integral numeric types, all enums, and bool. The only difference between & and && (and | and ||) on bool is that these don't short-circuit.

~ is logical negation on integral numeric types and enums, which, in a way, is an analog to ! for bool.

```
// Print the values -- and use the String.PadLeft() method to add leading 0's
Console.WriteLine("val1: {0}", Convert.ToString(val1, 2));
Console.WriteLine("val2: {0}", Convert.ToString(val2, 2).PadLeft(9, '0'));
Console.WriteLine("  or: {0}", Convert.ToString(or, 2).PadLeft(9, '0'));
Console.WriteLine(" and: {0}", Convert.ToString(and, 2).PadLeft(9, '0'));
Console.WriteLine(" xor: {0}", Convert.ToString(xor, 2).PadLeft(9, '0'));
Console.WriteLine(" not: {0}", Convert.ToString(not, 2).PadLeft(9, '0'));
// Notice what the ~ operator returned:  11111111111111111111111011111110
// It's the 32-bit complement of val1: 00000000000000000000000100000001
// The logical operators are operating on int, which is a 32-bit integer.
```

Convert.ToString() returns a String object, and we're calling the PadLeft() method on that object to pad the result out with zeroes.

This will make a lot more sense when you run the program and look at the output. Remember, you don't need to type in all of this code—you can download it all from the Head First Labs website!
http://www.headfirstlabs.com/books/hfcsharp

```
// The << and >> operators shift bits left and right. And you can combine any
// logical operator with =, so >>= or &= is just like += or *=.
int bits = Convert.ToInt32("11", 2);
for (int i = 0; i < 5; i++)
{
    bits <<= 2;
    Console.WriteLine(Convert.ToString(bits, 2).PadLeft(12, '0'));
}
for (int i = 0; i < 5; i++)
{
    bits >>= 2;
    Console.WriteLine(Convert.ToString(bits, 2).PadLeft(12, '0'));
}
```

This doesn't have anything to do with logic, it's just something useful that you see reasonably often.

```
// You can instantiate a new object and call a method on it without
// using a variable to refer to it.
Console.WriteLine(new Guy("Harry", 47, 376).ToString());
```

```
// We've used the + operator for string concatenation throughout the book, and that
// works just fine. However, a lot of people avoid using + in loops that will have
// to execute many times over time, because each time + executes it creates an extra
// object on the heap that will need to be garbage collected later. That's why .NET
// has a class called StringBuilder, which is great for efficiently creating and
// concatenating strings together. Its Append() method adds a string onto the end,
// AppendFormat() appends a formatted string (using {0} and {1} just like
// String.Format() and Console.WriteLine() do), and AppendLine() adds a string
// with a line break at the end. To get the final concatenated string, call
// its ToString() method.
StringBuilder stringBuilder = new StringBuilder("Hi ");
stringBuilder.Append("there, ");
stringBuilder.AppendFormat("{0} year old guy named {1}. ", joe.Age, joe.Name);
stringBuilder.AppendLine("Nice weather we're having.");
Console.WriteLine(stringBuilder.ToString());

Console.ReadKey();
```

One thing to note here: in this particular example, StringBuilder performs worse than +, because + will pre-compute the length of the string and figure out exactly how much memory to allocate.

You typically use StringBuilder when you don't know in advance the number of concatenations you want to perform.

```
/*
 * This is a good start, but it's by no means complete. Luckily, Microsoft gives you
 * a reference that has a complete list of all of the C# operators, keywords, and
 * other features of the language. Take a look through it -- and if you're just getting
 * started with C#, don't worry if it seems a little difficult to understand. MSDN
 * is a great source of information, but it's meant to be a reference, not a learning
 * or teaching guide.
 *
 * C# Programmer Reference: http://msdn.microsoft.com/en-us/library/618ayhy6.aspx
 * C# Operators: http://msdn.microsoft.com/en-us/library/6a71f45d.aspx
 * C# Keywords: http://msdn.microsoft.com/en-us/library/x53a06bb.aspx
 */
}
```

#3. Namespaces and assemblies

We made the decision to focus this book on the really practical stuff you need to know in order to build and run applications. Throughout every chapter, you create your projects in Visual Studio and run them in the debugger. We showed you where your compiled code ended up in an executable, and how to publish that executable so that other people can install it on their machines. That's enough to get you through every exercise in this book, but it's worth taking a step back and looking a little closer at what it is that you're building.

When you compile your C# program, you're creating an assembly. An assembly is a file that contains the compiled code. There are two kinds of assemblies. Executables (occasionally called "process assemblies") have the EXE file extension. All of the programs you write in this book are compiled as executables. Those are the assemblies that you can execute (you know, EXE files you can double-click and run). There are also library assemblies, which have the DLL file extension. They contain classes that you can use in your programs, and, as you'll see shortly, namespaces play a big role in how you use them.

You can get a handle on the basics of assemblies by first creating a class library, and then building a program that uses it. Start by **opening Visual Studio 2013 for Desktop** and creating a new Class Library project called `Headfirst.Csharp.Leftover3`. When the library is first created, it contains the file `Class.cs`. **Delete** that file and **add a new class** called `Guy.cs`. Open up the new `Guy.cs` file:

```
namespace Headfirst.Csharp.Leftover3
{
    class Guy
    {
    }
}
```

You can also create class libraries in Visual Studio for Windows 8. We asked you to create this project in the Desktop edition because all of the Framework assemblies are already referenced, so the "Add Reference" window that we show on the facing page will be empty.

Notice how Visual Studio made the namespace match your class library name? That's a very standard pattern.

Go ahead and **fill in the `Guy` class** with the code from leftover #2 — we'll use it in a minute. Next, **add two more classes** called `HiThereWriter` and `LineWriter`. Here's the code for `HiThereWriter`:

```
namespace Headfirst.Csharp.Leftover3
{
    public static class HiThereWriter
    {
      public static void HiThere(string name)
       {
          MessageBox.Show("Hi there! My name is " + name);
       }
    }
}
```

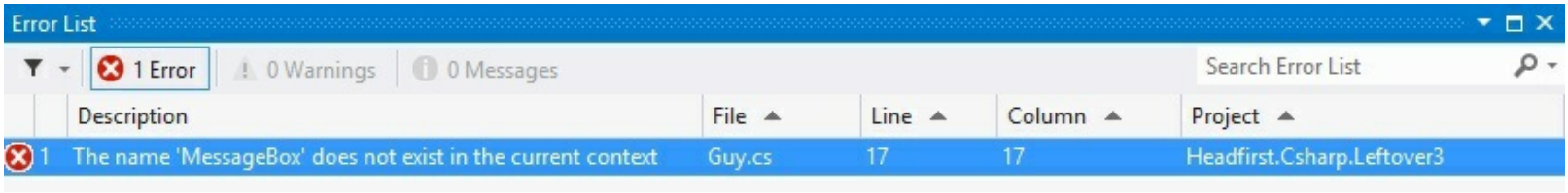
And here's the code for `LineWriter` (it's also in the `Headfirst.Csharp.Leftover3` namespace):

```
internal static class LineWriter {
    public static void WriteALine(string message)
    {
        Console.WriteLine(message);
```

```
    }
}
```

We named the class library `Headfirst.Csharp.Leftover3` because that's a pretty standard way of naming assemblies. Read more about assembly naming here: http://msdn.microsoft.com/en-us/library/ms229048.aspx

Now try to compile your program. You'll get an error:

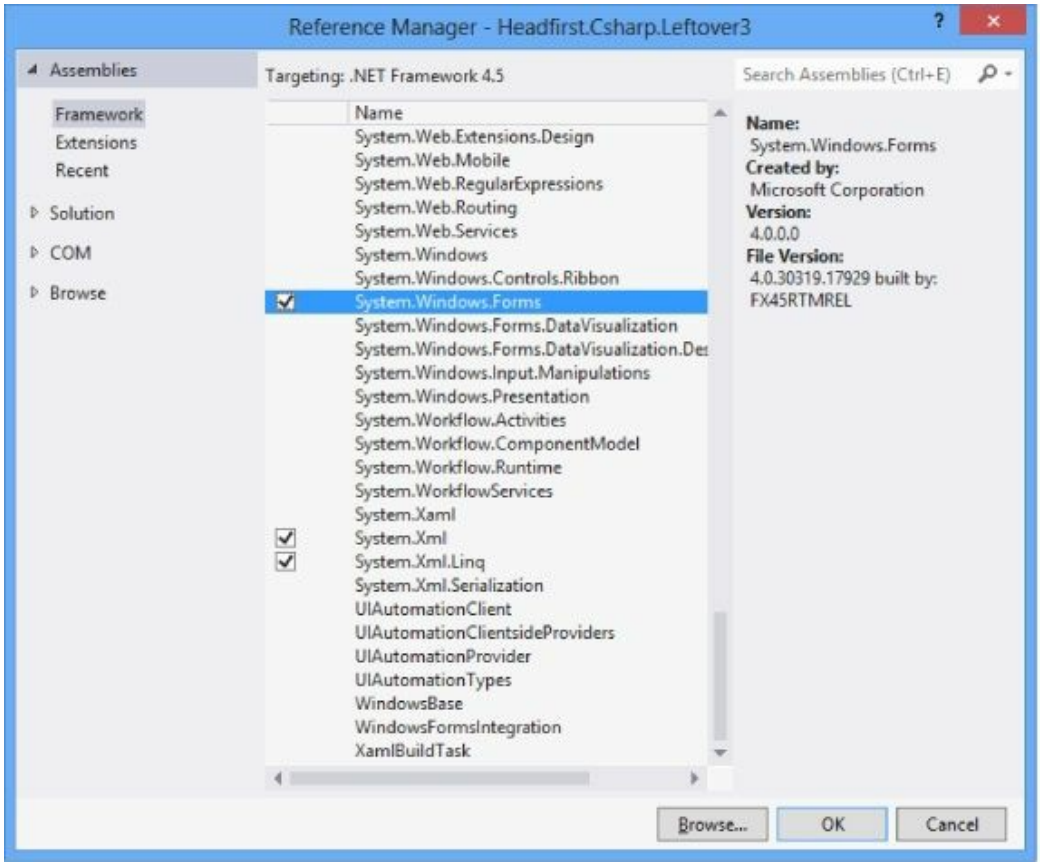


OK, no problem — we know how to fix this in a Desktop app. Add a line to the top of your class:

```
using System.Windows.Forms;
```

Wait, it still doesn't compile! And something's weird here. When you typed in that line, did you notice that when you got as far as "using System.Win" the IntelliSense window stopped giving you suggestions? That's because your project **hasn't referenced the `System.Windows.Forms` assembly**.

Let's fix this by referencing the correct assembly. Go to the Solution Explorer and expand the "References" folder in your project. Right-click on it and choose "Add Reference..."; a window should pop up:



This window is showing you the assemblies your program can access. Some of them are stored in the Global Assembly Cache (GAC), but not every assembly in the GAC shows up in this window. The GAC is a central, machine-wide set of assemblies that all of the .NET programs on the computer can access. You can see all of the assemblies in it by typing `%systemroot%\Microsoft.NET\assembly` into the Search box on the Start page (or Start/Run for older versions of Windows).

On the .NET tab, start typing "System.Windows.Forms" — it should jump down to that assembly. Make sure it's highlighted and click OK. Now `System.Windows.Forms` should show up under the References folder in the Solution Explorer — and your program compiles!

The "Add References" window figures out which assemblies to display by checking a registry key, not the GAC. For more info: http://support.microsoft.com/kb/306149

...so what did I just do?

Take a close look at the declarations for `LineWriter` and `HiThereWriter`:

```
public class HiThereWriter

internal static class LineWriter
```

There are **access modifiers on the class declarations**: `HiThereWriter` is declared with the **`public`** access modifier, and `LineWriter` is declared with the **`internal`** one. In a minute, you'll write a console application that references this class library. A program can only *directly* access another class library's public classes — although they can be accessed indirectly, like when one method calls another or returns an instance of an internal object that implements a public interface.

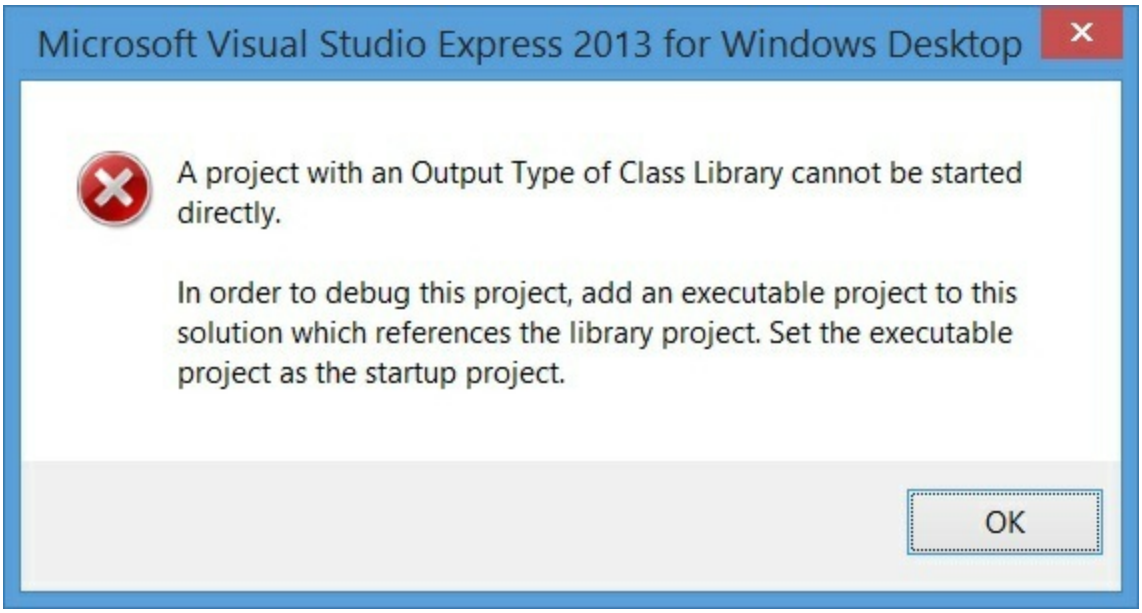
Now go back to your `Guy` class and look at its declaration:

```
class Guy
```

Since there's no access modifier, it defaults to `internal`. We'll want to expose `Guy` to other assemblies that reference this one, so **change the declaration** to be public:

```
public class Guy
```

Next, try running your program in the debugger. You'll see this error:



That makes sense when you think about it, because a class library doesn't have an entry point. It's just a bunch of classes that other programs can use. So let's add an executable program that uses those classes — that way the debugger has something to run. Visual Studio has a really useful feature that we'll take advantage of next: it can load multiple projects into a single solution. **Right-click on the Solution in the Solution Explorer and choose Add >> New Project...** to bring up the usual Add Project window. Add a new console application called `MyProgram`.

Once your new program's added, it should appear in the Solution Explorer right under the class library. Right-click on References underneath MyProgram, **choose "Add reference..." from the menu**, expand

, and **click on Projects**. You should see your class library project listed (

). Make sure it's checked. It should now appear in the Solution Explorer when you expand "References" under your MyProgram project.

Next, go to the top of your new project's `Program.cs` file and start adding this `using` line:

```
using Headfirst.Csharp.Leftover3;
```

Notice how the IntelliSense picks up "Csharp" and "Leftover3" as you're typing?

Now we can write a new program. Start by typing **`Guy`**. Watch what pops up:

```
static void Main(string[] args)
{
```

The IntelliSense window lists the entire namespace for `Guy`, so you can see that you're actually using the class that you defined in the other assembly. Finish the program:

```
static void Main(string[] args {
     Guy guy = new Guy("Joe", 43, 125);
     HiThereWriter.HiThere(guy.Name);
}
```

Now run your program. Oh, wait — you get the same error message as before, because you can't run a class library! No problem. Right-click on your new `MyProgram` project in the Solution Explorer and choose Set as StartUp Project. Your solution can have many different projects, and this is how you tell it which one to start when you run it in the debugger. Now run your program again — this time it runs!

NOTE

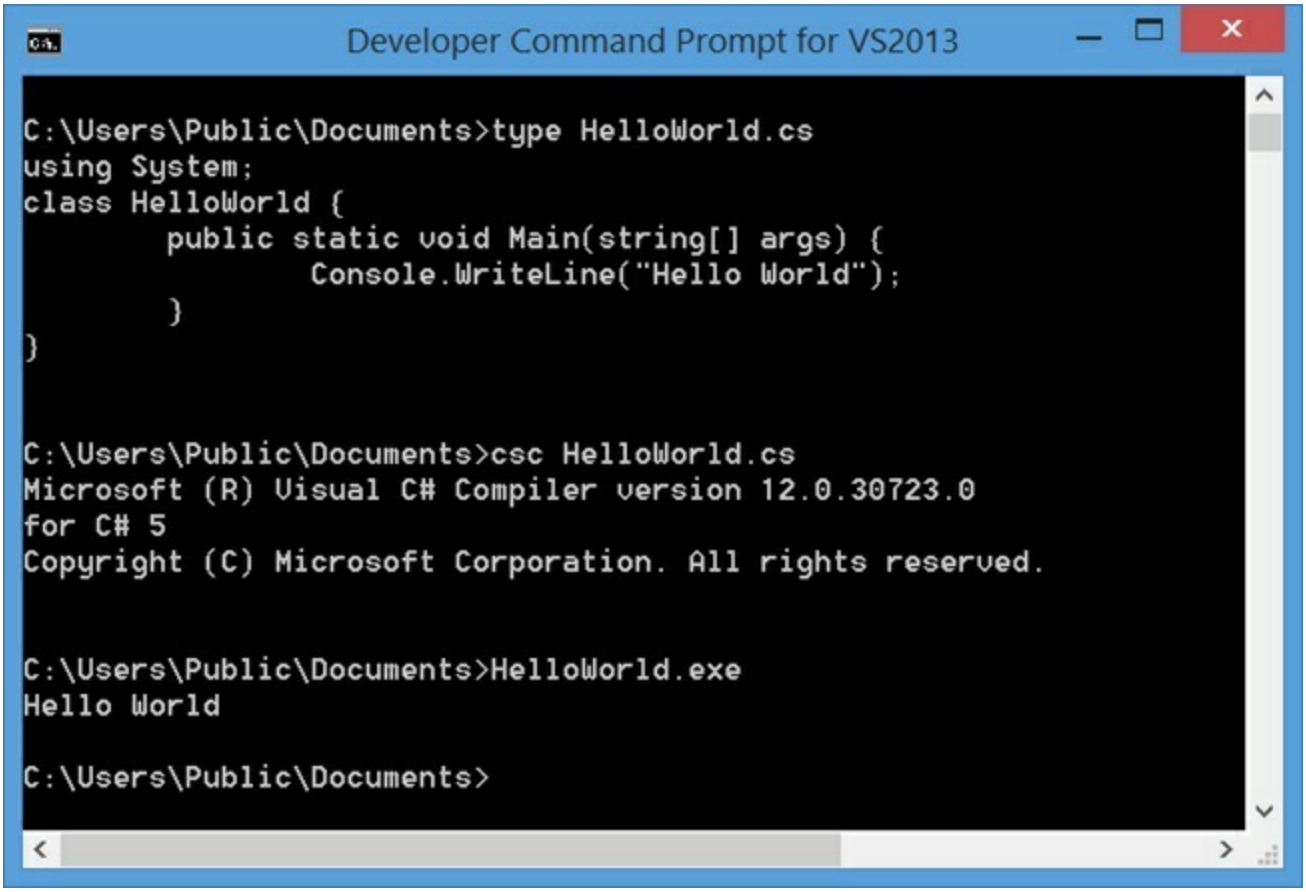
Throughout the book we tell you that you compile your code. When you do, it's compiled to **Common Intermediate Language** (IL), the low-level language used by .NET. It's a human-readable assembly language, and all .NET languages (including C# and Visual Basic) are compiled into it. The IL code is compiled into native machine language when you run your program using the CLR's **just-intime compiler**, so named because it compiles the IL into native code just in time to execute it (rather than pre-compiling it before it's run).

That means your EXEs and DLLs contain IL, and not native assembly code, which is important because it means many languages can compile to IL that the CLR can run — including Visual Basic .NET, F#, J#, managed C++/CLI, JScript .NET, Windows PowerShell, IronPython, Iron Ruby, and more. This is really useful: since VB.NET code compiles to IL, you can build an assembly in C# and use it in a VB.NET program (or vice versa).

If you have a Macintosh or Linux box, try installing Mono. It's an open source implementation of IL that runs EXE files that you've built on the PC (typically by typing "mono MyProgram.exe" — but this only works on *some* .NET assemblies). We're not going to talk any more about that, though, because this book is focused on Microsoft technology. But we do have to admit that it ***is*** pretty cool to see the Go Fish game or Hide and Seek running natively on Mac or Linux!

Building a “Hello World” program from the command line

There’s a tradition in programming called *Hello World*: a program that just prints one line of text (“Hello World”). This is typically the first program you’ll write in a new language, because if you can do that it proves that your tools work well enough to run more complex programs. The **Developer Command Prompt** is a shortcut installed with Visual Studio 2013 (typically in the *C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\Tools\Shortcuts* folder) that puts the C# compiler csc.exe and other tools in your path. Run the Developer Command Prompt, then try using Notepad to create HelloWorld.cs, using csc.exe to build an executable, and then running that executable:

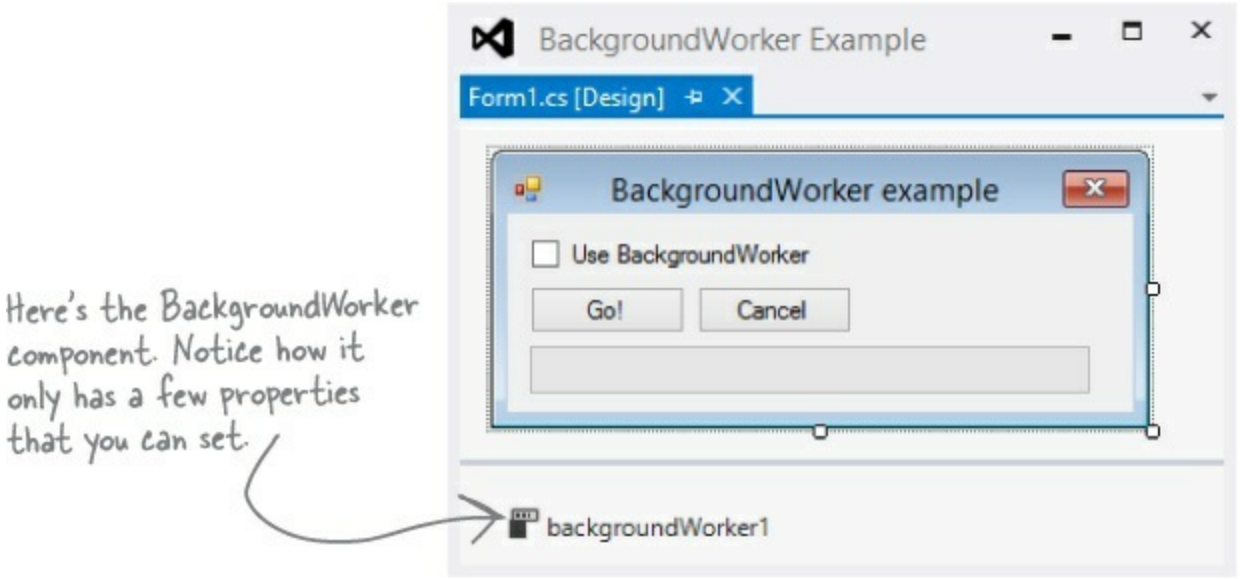


We’re just scratching the surface of assemblies. There’s a lot more (including versioning and signing them for security). You can read more about assemblies here: http://msdn.microsoft.com/en-us/library/k3677y81.aspx

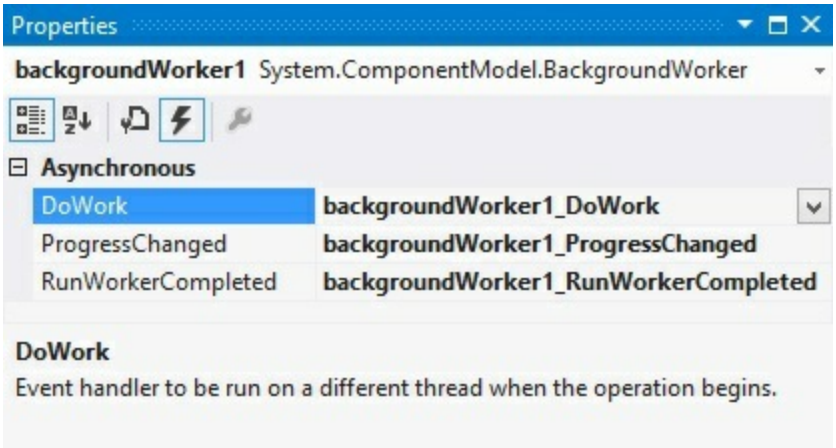
#4. Use BackgroundWorker to make your WinForms responsive

Throughout the book, we've shown you a few ways that you can make your programs do more than one thing at a time. In Chapter 2, you learned about how to use the `Application.DoEvents()` method to let your form respond to button clicks while still in a loop. But that's ***not a good solution*** (for a bunch of reasons we didn't get into), so we showed you a much better solution in Chapter 4: using a timer to trigger an event at a regular interval. Later on, you learned how ot use `async`, `await`, and `Task`. An alternative to asynchronous methods is threading, but it can be very tricky and can lead to some very nasty bugs if you're not careful. Luckily, .NET gives you a really useful component called **`BackgroundWorker`** that makes it easier to let your program use threads safely.

Here's a simple project to help you understand how BackgroundWorker works. Start by building this form. You'll need to drag a `CheckBox` onto it (name it `useBackgroundWorkerCheckbox`), two buttons (named `goButton` and `cancelButton`) and a `ProgressBar` (named `progressBar1`). Then drag a `BackgroundWorker` onto the form. It'll show up in the gray box on the bottom of the designer. Keep its name `backgroundWorker1`, and set its `WorkerReportsProgress` and `WorkerSupportsCancellation` properties to `true`.



Select the `BackgroundWorker` and go to the Events page in the Properties window (by clicking on the lightning-bolt icon). It's got three events: `DoWork`, `ProgressChanged`, and `RunWorkerCompleted`. Double-click on each of them to add an event handler for each event.



The code for the form is on the next two pages.

Here's the code for the form.

```
/// <summary>
/// Waste CPU cycles causing the program to slow down by doing calculations for 100ms
/// </summary>
private void WasteCPUCycles() {
    DateTime startTime = DateTime.Now;
    double value = Math.E;
    while (DateTime.Now < startTime.AddMilliseconds(100)) {
        value /= Math.PI;
        value *= Math.Sqrt(2);
    }
}
/// <summary>
/// Clicking the Go button starts wasting CPU cycles for 10 seconds
/// </summary>
private void goButton_Click(object sender, EventArgs e) {
    goButton.Enabled = false;
    if (!useBackgroundWorkerCheckbox.Checked) {
        // If we're not using the background worker, just start wasting CPU cycles
        for (int i = 1; i <= 100; i++) {
            WasteCPUCycles();
            progressBar1.Value = i;
        }
        goButton.Enabled = true;
    } else {
        cancelButton.Enabled = true;

        // If we are using the background worker, use its RunWorkerAsync()
        // to tell it to start its work
        backgroundWorker1.RunWorkerAsync(new Guy("Bob", 37, 146));
    }
}
/// <summary>
/// The BackgroundWorker object runs its DoWork event handler in the background
/// </summary>
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e) {
    // The e.Argument property returns the argument that was passed to RunWorkerAsync()
    Console.WriteLine("Background worker argument: " + (e.Argument ?? "null"));

    // Start wasting CPU cycles
    for (int i = 1; i <= 100; i++) {
        WasteCPUCycles();
        // Use the BackgroundWorker.ReportProgress method to report the % complete
        backgroundWorker1.ReportProgress(i);

        // If the BackgroundWorker.CancellationPending property is true, cancel
        if (backgroundWorker1.CancellationPending) {
            Console.WriteLine("Cancelled");
            break;
        }
    }
}
```

The WasteCPUCycles() does a whole bunch of mathematical calculations to tie up the CPU for 100 milliseconds, and then it returns.

If the form's using the background worker, it enables the Cancel button.

When the user clicks on the Go! button, the event handler checks to see if the "Use BackgroundWorker" checkbox is checked. If it isn't, the form wastes CPU cycles for 10 seconds. If it is, the form calls the BackgroundWorker's RunWorkerAsync() method to tell it to start doing its work in the background.

When you tell a BackgroundWorker to start work, you can give it an argument. In this case, we're passing it a Guy object (see leftover #2 for its definition).

Here's a good example of how to use the ?? null coalescing operator we talked about in leftover #2. If e.Argument is null, this returns "null", otherwise it returns e.Argument.

The CancellationPending method checks if the BackgroundWorker's CancelAsync() method was called.

When the BackgroundWorker's RunWorkerAsync() method is called, it starts running its DoWork event handler method in the background. Notice how it's still calling the same WasteCPUCycles() method to waste CPU cycles. It's also calling the ReportProgress() method to report a percent complete (a number from 0 to 100).

The BackgroundWorker only fires its ProgressChanged and RunWorkerCompleted events if its WorkerReportsProgress and WorkerSupportsCancellation properties are true.

```
/// <summary>
/// BackgroundWorker fires its ProgressChanged event when the worker thread reports progress
/// </summary>
private void backgroundWorker1_ProgressChanged(object sender, ProgressChangedEventArgs e) {
    progressBar1.Value = e.ProgressPercentage;
}

/// <summary>
/// BackgroundWorker fires its RunWorkerCompleted event when its work is done (or cancelled)
/// </summary>
private void backgroundWorker1_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    goButton.Enabled = true;
    cancelButton.Enabled = false;
}

/// <summary>
/// When the user clicks Cancel, call BackgroundWorker.CancelAsync() to send it a cancel message
/// </summary>
private void cancelButton_Click(object sender, EventArgs e) {
    backgroundWorker1.CancelAsync();
}
```

When the DoWork event handler calls the ProgressChanged() method, it causes the BackgroundWorker to raise its ProgressChanged event and set e.ProgressPercentage to the percent passed to it.

When the work is complete, the RunWorkerCompleted event handler re-enables the Go! button and disables the Cancel button.

If the user clicks Cancel, it calls the BackgroundWorker's CancelAsync() method to give it the message to cancel.

Once you've got your form working, run the program. It's easy to see how `BackgroundWorker` makes your program much more responsive:

- Make sure the "Use BackgroundWorker" checkbox isn't checked, then click the Go! button. You'll see the progress bar start to fill up. Try to drag the form around — you can't. The form's all locked up. If you're lucky, it might jump a bit as it eventually responds to your mouse drag.
- When it's done, check the "Use BackgroundWorker" checkbox and click the Go! button again. This time, the form is perfectly responsive. You can move it around and even close it, and there's no delay. When it finishes, it uses the `RunWorkerCompleted` method to re-enable the buttons.
- While the program is running (using `BackgroundWorker`), click the Cancel button. It will update its `CancellationPending` property, which will tell the program to cancel and exit the loop.

Are you wondering why you need to use the `ReportProgress()` method rather than setting the `ProgressBar`'s `Value` property directly? Try it out. Add the following line to the `DoWork` event handler:

```
progressBar1.Value = 10;
```

Then run your program again. As soon as it hits that line, it throws an `InvalidOperationException` with this message: "Cross-thread operation not valid: Control 'progressBar1' accessed from a thread other than the thread it was created on." The reason it throws that exception is that `BackgroundWorker` starts a separate thread and executes the `DoWork` method on it. So there are two threads: the GUI thread that's running the form and the background thread. One of the .NET threading rules is that only the GUI thread can update form controls; otherwise, that exception is thrown.

This is just one of the many threading pitfalls that can trap a new developer — that's why we didn't talk about threading anywhere in this book. If you're looking to get started with threads, we highly recommend Joe Albahari's excellent e-book about threading in C# and .NET: http://www.albahari.com/threading

#5. The Type class and GetType()

One of the most powerful aspects of the C# programming language is its rich type system. But until you've got some experience building programs, it's difficult to appreciate it — in fact, it can be a little baffling at first. But we want to give you at least a taste of how types work in C# and .NET. Here's a console application that gives you an introduction to some of the tools you have at your disposal to work with types.

```
class Program {
    class NestedClass {
        public class DoubleNestedClass {
            // Nested class contents ...
        }
    }

    static void Main(string[] args) {
        Type guyType = typeof(Guy);
        Console.WriteLine("{0} extends {1}",
            guyType.FullName,
            guyType.BaseType.FullName);
        // output: TypeExamples.Guy extends System.Object

        Type nestedClassType = typeof(NestedClass.DoubleNestedClass);
        Console.WriteLine(nestedClassType.FullName);
        // output: TypeExamples.Program+NestedClass+DoubleNestedClass

        List<Guy> guyList = new List<Guy>();
        Console.WriteLine(guyList.GetType().Name);
        // output: List`1

        Dictionary<string, Guy> guyDictionary = new Dictionary<string, Guy>();
        Console.WriteLine(guyDictionary.GetType().Name);
        // output: Dictionary`2

        Type t = typeof(Program);
        Console.WriteLine(t.FullName);
        // output: TypeExamples.Program

        Type intType = typeof(int);
        Type int32Type = typeof(Int32);
        Console.WriteLine("{0} - {1}", intType.FullName, int32Type.FullName);
        // System.Int32 - System.Int32

        Console.WriteLine("{0} {1}", float.MinValue, float.MaxValue);
        // output:-3.402823E+38 3.402823E+38

        Console.WriteLine("{0} {1}", int.MinValue, int.MaxValue);
        // output:-2147483648 2147483647

        Console.WriteLine("{0} {1}", DateTime.MinValue, DateTime.MaxValue);
        // output: 1/1/0001 12:00:00 AM 12/31/9999 11:59:59 PM

        Console.WriteLine(12345.GetType().FullName);
        // output: System.Int32

        Console.ReadKey();
    }
}
```

We only mentioned it briefly, but here's a reminder that you can nest classes inside of each other. Program contains NestedClass, which contains DoubleNestedClass.

Here's the entry point...

You can use the typeof keyword to turn a type (like Guy, int, or DateTime) into a Type object. Then you can find out its full name and base type (and if it didn't inherit from anything, its base type is System.Object).

When you get the type of a generic, its name is the type name followed by a backward quote and the number of its generic parameters.

This is the System.Type class. The GetType() method returns a Type object.

The FullName property we used in the first part of this program is a member of System.Type.

float is an alias for System.Single and int is an alias for System.Int32. They're both structs (which you learned all about in Chapter 14).

Numeric value types and DateTime have MinValue and MaxValue properties that return the lowest and highest valid value

Literals have types, too! And you can use GetType() to get those types.

There's so much more to learn about types! Read more about them here: http://msdn.microsoft.com/en-us/library/ms173104.aspx

#6. Equality, IEquatable, and Equals()

Throughout the book, when you've wanted to compare values in two variables, you'd use the == operator. But you already know that all things being equal, some values are more "equal" than others. The == operator works just fine for value types (like `int`s, `double`s, `DateTime`s, or other `struct`s), but when you use it on reference types you just end up comparing whether two reference variables are pointing to the same object (or if they're both null). That's fine for what it is, but it turns out that C# and .NET provide a rich set of tools for dealing with value equality in objects.

To start out, every object has a method `Equals()`, which by default returns true only if you pass it a reference to itself. And there's a static method, `Object.ReferenceEquals()`, which takes two parameters and returns true if they both point to the same object (or if they're both null). Here's an example, which you can try yourself in a console application:

```
Guy joe1 = new Guy("Joe", 37, 100);
Guy joe2 = joe1;
Console.WriteLine(Object.ReferenceEquals(joe1, joe2));  // True
Console.WriteLine(joe1.Equals(joe2));                   // True
Console.WriteLine(Object.ReferenceEquals(null, null));  // True

joe2 = new Guy("Joe", 37, 100);
Console.WriteLine(Object.ReferenceEquals(joe1, joe2));  // False
Console.WriteLine(joe1.Equals(joe2));                   // False
```

Again, we're using the same Guy class from leftover #2.

But that's just the beginning. There's an interface built into .NET called `IEquatable<T>` that you can use to add code to your objects so they can tell if they're equal to other objects. An object that implements `IEquatable<T>` knows how to compare its value to the value of an object of type T. It has one method, `Equals()`, and you implement it by writing code to compare the current object's value to that of another object. There's an MSDN page that has more information about it (http://msdn.microsoft.com/en-us/library/ms131190.aspx). Here's an important excerpt:

NOTE

"If you implement Equals, you should also override the base class implementations of Object.Equals(Object) and GetHashCode so that their behavior is consistent with that of the IEquatable<T>.Equals method. If you do override Object.Equals(Object), your overridden implementation is also called in calls to the static Equals(System.Object, System.Object) method on your class. This ensures that all invocations of the Equals method return consistent results, which the example illustrates."

If you don't do this, the compiler will give you a warning.

Here's a class called `EquatableGuy`, which extends `Guy` and implements `IEquatable<Guy>`:

```
/// <summary>
/// A guy that knows how to compare itself with other guys
/// </summary>
class EquatableGuy : Guy, IEquatable<Guy> {

    public EquatableGuy(string name, int age, int cash)
        : base(name, age, cash) { }

    /// <summary>
    /// Compare this object against another EquatableGuy
    /// </summary>
    /// <param name="other">The EquatableGuy object to compare with</param>
    /// <returns>True if the objects have the same values, false otherwise</returns>
    public bool Equals(Guy other) {
        if (ReferenceEquals(null, other)) return false;
        if (ReferenceEquals(this, other)) return true;
        return Equals(other.Name, Name) && other.Age == Age && other.Cash == Cash;
    }
    /// <summary>
    /// Override the Equals method and have it call Equals(Guy)
    /// </summary>
    /// <param name="obj">The object to compare to</param>
    /// <returns>True if the value of the other object is equal to this one</returns>
    public override bool Equals(object obj) {
        if (!(obj is Guy)) return false;
        return Equals((Guy)obj);
    }

    /// <summary>
    /// Part of the contract for overriding Equals is that you need to override
    /// GetHashCode() as well. It should compare the values and return true
    /// if the values are equal.
    /// </summary>
    /// <returns></returns>
    public override int GetHashCode() {
        const int prime = 397;
        int result = Age;
        result = (result * prime) ^ (Name != null ? Name.GetHashCode() : 0);
        result = (result * prime) ^ Cash;
        return result;
    }
}
```

The Equals() method compares the actual values in the other Guy object's fields, checking his Name, Age, and Cash to see if they're the same and only returning true if they are.

We're also overriding the Equals() method that we inherited from Object, as well as GetHashCode (because of the contract mentioned in that MSDN article).

Since our other Equals() method already compares guys, we'll just call it.

This is a pretty standard pattern for GetHashCode(). Note the use of the bitwise XOR (^) operator, a prime number, and the conditional operator (?:).

And here's what it looks like when you use `Equals()` to compare two `EquatableGuy` objects:

```
joe1 = new EquatableGuy("Joe", 37, 100);
joe2 = new EquatableGuy("Joe", 37, 100);
Console.WriteLine(Object.ReferenceEquals(joe1, joe2));  // False
Console.WriteLine(joe1.Equals(joe2));                   // True

joe1.GiveCash(50);
Console.WriteLine(joe1.Equals(joe2));                   // False
joe2.GiveCash(50);
Console.WriteLine(joe1.Equals(joe2));                   // True
```

Guy.Equals() will only return true if the actual values of the objects are the same.

And now that `Equals()` and `GetHashCode()` are implemented to check the values of the fields and properties, the method `List.Contains()` now works. Here's a `List<Guy>` that contains several `Guy`

objects, including a new `EquatableGuy` object with the same values as the one referenced by `joe1`.

```
List<Guy> guys = new List<Guy>() {
    new Guy("Bob", 42, 125),
    new EquatableGuy(joe1.Name, joe1.Age, joe1.Cash),
    new Guy("Ed", 39, 95)
};

Console.WriteLine(guys.Contains(joe1));          // True

Console.WriteLine(joe1 == joe2);                 // False
```

List.Contains() will go through its contents and call each object's Equals() method to compare it with the reference you pass to it.

Even though joe1 and joe2 point to objects with the same values, == and != still compare the references, not the values themselves.

Isn't there something we can do about that? Flip the page and find out!

If you try to compare two `EquatableGuy` references with the == or != operators, they'll just check if both references are pointing to the same object or if they're both null. But what if you want to make them actually compare the values of the objects? It turns out that you can actually **overload an operator** — redefining it to do something specific when it operates on references of a certain type. You can see an example of how it works in the `EquatableGuyWithOverload` class, which extends `EquatableGuy` and adds overloading of the == and != operators:

```
/// <summary>
/// A guy that knows how to compare itself with other guys
/// </summary>
class EquatableGuyWithOverload : EquatableGuy
{
    public EquatableGuyWithOverload(string name, int age, int cash)
        : base(name, age, cash) { }

    public static bool operator ==(EquatableGuyWithOverload left,
                                   EquatableGuyWithOverload right)
    {
        if (Object.ReferenceEquals(left, null)) return false;
        else return left.Equals(right);
    }

    public static bool operator !=(EquatableGuyWithOverload left,
                                   EquatableGuyWithOverload right)
    {
        return !(left == right);
    }

    public override bool Equals(object obj) {
        return base.Equals(obj);
    }

    public override int GetHashCode() {
        return base.GetHashCode();
    }
}
```

If we used == to check for null instead of Object.ReferenceEquals(), we'd get a StackOverflowException. Can you figure out why?

Since we've already defined ==, we can just invert it for !=.

If we don't override Equals() and GetHashCode(), the IDE will give this warning: 'EquatableGuyWithOverload' defines operator == or operator != but does not override Object.GetHashCode().

Since EquatableGuyWithOverload acts just like EquatableGuy and Guy, we can just call the base methods.

Here's some code that uses `EquatableGuyWithOverload` objects:

```
joe1 = new EquatableGuyWithOverload(joe1.Name, joe1.Age, joe1.Cash);
joe2 = new EquatableGuyWithOverload(joe1.Name, joe1.Age, joe1.Cash);
Console.WriteLine(joe1 == joe2);    // False
Console.WriteLine(joe1 != joe2);    // True

Console.WriteLine((EquatableGuyWithOverload)joe1 ==
                                    (EquatableGuyWithOverload)joe2);   // True
Console.WriteLine((EquatableGuyWithOverload)joe1 !=
                                    (EquatableGuyWithOverload)joe2);   // False
joe2.ReceiveCash(25);
Console.WriteLine((EquatableGuyWithOverload)joe1 ==
                                   (EquatableGuyWithOverload)joe2);    // False
Console.WriteLine((EquatableGuyWithOverload)joe1 !=
                                   (EquatableGuyWithOverload)joe2);    // True
```

Wait, what happened? It's calling Guy's == and != operators. Cast to EquatableGuyWithOverload to call the correct == and !=

#7. Using yield return to create enumerable objects

In Chapter 8 we learned about the `IEnumerable` interface and how it's used by the foreach loop. C# and .NET give you some useful tools for building your own collections and enumerable types, starting with the `IEnumerable` interface. Let's say you want to create your own enumerator that returns values from this `Sport` enum in order:

```
enum Sport
{
    Football, Baseball,
    Basketball, Hockey,
    Boxing, Rugby, Fencing,
}
```

You could manually implement `IEnumerable` yourself, building the `Current` property and `MoveNext()` method:

```
class SportCollection : IEnumerable<Sport> {
    public IEnumerator<Sport> GetEnumerator() {
        return new ManualSportEnumerator();
    }
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
    class ManualSportEnumerator : IEnumerator<Sport> {
        int current = -1;
        public Sport Current { get { return (Sport)current; } }
        public void Dispose() { return; } // Nothing to dispose
        object System.Collections.IEnumerator.Current { get { return Current; } }
        public bool MoveNext() {
            int maxEnumValue = Enum.GetValues(typeof(Sport)).Length;
            if ((int)current >= maxEnumValue)
                return false;
            current++;
            return true;
        }
        public void Reset() { current = 0; }
    }
}
```

IEnumerable just contains one method, GetEnumerator(), but we also need to build the class for the enumerator it returns.

The enumerator implements IEnumerator<Sport>. The foreach loop uses its Current property and MoveNext() method.

The MoveNext() method increments current and uses it to return the next sport in the enum.

Here's a foreach loop that loops through `SportCollection`. It returns the sports in order (Football, Baseball, Basketball, Hockey, Boxing, Rugby, Fencing):

```
Console.WriteLine("SportCollection contents:");
SportCollection sportCollection = new SportCollection();
foreach (Sport sport in sportCollection)
    Console.WriteLine(sport.ToString());
```

That's a lot of work to build an enumerator — it has to manage its own state, and keep track of which sport it returned. Luckily, C# gives you a really useful tool to help you easily build enumerators. It's called `yield return`, and you'll learn about it when you flip the page.

NOTE

Just a reminder of something from Chapter 14: all collections are enumerable, but not everything that's enumerable is technically a collection unless it implements the ICollection<T> interface. We didn't show you how to build collections from the ground up, but understanding enumerators is definitely enough to get you started down that road.

The `yield return` statement is a kind of all-in-one automatic enumerator creator. This `SportCollection` class does exactly the same thing as the one on the previous page, but its enumerator is only three lines long.:

```
class SportCollection : IEnumerable<Sport> {
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
    public IEnumerator<Sport> GetEnumerator() {
        int maxEnumValue = Enum.GetValues(typeof(Sport)).Length - 1;
        for (int i = 0; i <= maxEnumValue; i++) {
            yield return (Sport)i;
        }
    }
}
```

Like we said earlier, this is just the start for a SportCollection class. You'd still want to implement the ICollection<Sport> interface.

That looks a little odd, but if you actually debug through it you can see what's going on. When the compiler sees a method with a `yield return` statement that returns an `IEnumerator` or `IEnumerator<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 and step through it using Step Into (F11). To make it a little easier, here's a really simple enumerator called `NameEnumerator()` that iterates through four names:

```
static IEnumerable<string> NameEnumerator() {
    yield return "Bob"; // The method exits after this statement ...
    yield return "Harry"; // ... and resumes here the next time through
    yield return "Joe";
    yield return "Frank";
}
```

And here's a foreach loop that iterates through it. Use Step Into (F11) to see exactly what's going on:

```
IEnumerable<string> names = NameEnumerator(); // Put a breakpoint here
foreach (string name in names)
    Console.WriteLine(name);
```

There's another thing that you typically see in a collection: an **indexer**. When 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 actually 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. 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 { return (Sport)index; }
  }
```

Passing that indexer 3 will return the enum value `Hockey`.

Here's an `IEnumerable<Guy>` that keeps track of a bunch of guys, with an indexer that lets you get or set guys' ages.

```
class GuyCollection : IEnumerable<Guy> {
 private static readonly Dictionary<string, int> namesAndAges = new Dictionary<string, int>()
 {
        {"Joe", 41}, {"Bob", 43}, {"Ed", 39}, {"Larry", 44}, {"Fred", 45}
 };

 public IEnumerator<Guy> GetEnumerator() {
     Random random = new Random();
     int pileOfCash = 125 * namesAndAges.Count;

     int count = 0;
     foreach (string name in namesAndAges.Keys) {
         int cashForGuy = (++count < namesAndAges.Count) ? random.Next(125) : pileOfCash;
         pileOfCash -= cashForGuy;
         yield return new Guy(name, namesAndAges[name], cashForGuy);
     }
 }

 System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {
     return GetEnumerator();
 }

 /// <summary>
 /// Gets or sets the age of a given guy
 /// </summary>
 /// <param name="name">Name of the guy</param>
 /// <returns>Age of the guy</returns>
 public int this[string name] {
     get {
         if (namesAndAges.ContainsKey(name))
             return namesAndAges[name];
         throw new KeyNotFoundException("Name " + name + " was not found");
     }
     set {
         if (namesAndAges.ContainsKey(name))
             namesAndAges[name] = value;
         else
             namesAndAges.Add(name, value);
     }
 }
}
```

The enumerator uses this private Dictionary to keep track of the guys it'll create, but it doesn't actually create the Guy objects themselves until its enumerator is used.

It creates Guy objects with random amounts of cash. We're just doing this to show that the enumerator can create objects on the fly during a foreach loop.

When an invalid index is passed to an indexer, it typically throws an IndexOutOfRangeException.

This indexer has a set accessor that either updates a guy's age or adds a new guy to the Dictionary.

And here's some code that uses the indexers to update one guy's age and add two more guys, and then loop through them:

```
  Console.WriteLine("Adding two guys and modifying one guy");
  guyCollection["Bob"] = guyCollection["Joe"] + 3;
  guyCollection["Bill"] = 57;
  guyCollection["Harry"] = 31;
  foreach (Guy guy in guyCollection)
```

```
Console.WriteLine(guy.ToString());
```

#8. Refactoring

Refactoring means changing the way your code is structured without changing its behavior. Whenever you write a complex method, you should take a few minutes to step back and figure out how you can change it so that you make it easier to understand. Luckily, the IDE has some very useful refactoring tools built in. There are all sorts of refactorings you can do — here are some we use often.

Extract a method

When we were writing the control-based renderer for the GDI+ PDF, we originally included this foreach loop:

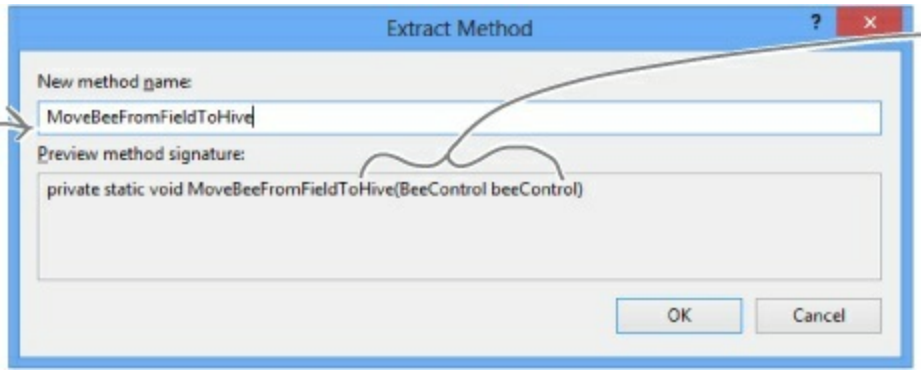
```
foreach (Bee bee in world.Bees) {
    beeControl = GetBeeControl(bee);
    if (bee.InsideHive) {
        if (fieldForm.Controls.Contains(beeControl)) {
            fieldForm.Controls.Remove(beeControl);
            beeControl.Size = new Size(40, 40);
            hiveForm.Controls.Add(beeControl);
            beeControl.BringToFront();
    } else if (hiveForm.Controls.Contains(beeControl)) {
        hiveForm.Controls.Remove(beeControl);
        beeControl.Size = new Size(20, 20);
        fieldForm.Controls.Add(beeControl);
        beeControl.BringToFront();
    }
    beeControl.Location = bee.Location;
}
```

These four lines move a BeeControl from the Field form to the Hive form.

And these four lines move a BeeControl from the Hive form to the Field form.

One of our tech reviewers, Joe Albahari, pointed out that this was a little hard to read. He suggested that we **extract those two four-line blocks into methods**. So we selected the first block, right-clicked on it, and selected "Refactor >> Extract Method...". This window popped up:

We typed in a name for the new method. We decided to call it MoveBeeFromFieldToHive() because that pretty much describes what the code does.



The IDE examined the code that we selected and figured out that it uses a BeeControl variable called beeControl, so it added it as a parameter to the method.

Then we did the same thing for the other four-line block, extracting it into a method that we named `MoveBeeFromHiveToField()`. Here's how that `foreach` loop ended up — it's a lot easier to read:

```
foreach (Bee bee in world.Bees) {
    beeControl = GetBeeControl(bee);
    if (bee.InsideHive) {
        if (fieldForm.Controls.Contains(beeControl))
            MoveBeeFromFieldToHive(beeControl);
    } else if (hiveForm.Controls.Contains(beeControl))
            MoveBeeFromHiveToField(beeControl, bee);
    beeControl.Location = bee.Location;
}
```

NOTE

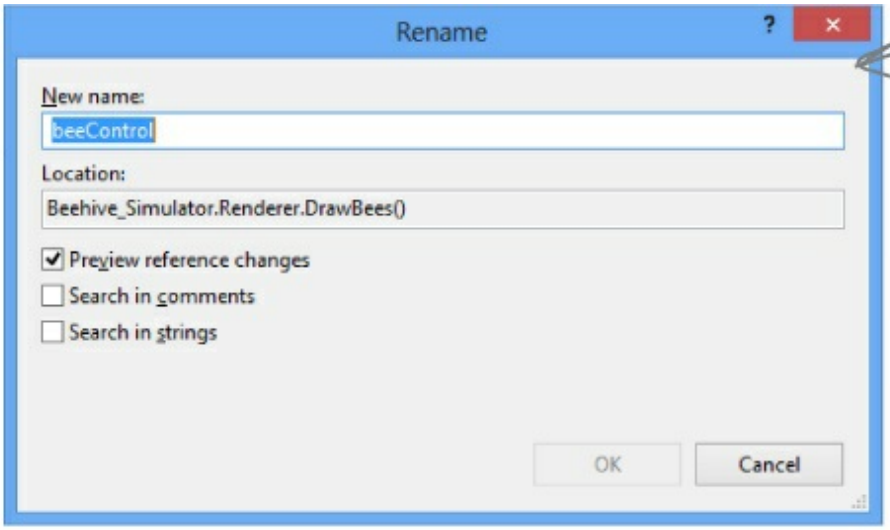
You did some refactoring back in Chapter 5 when you changed the structure of the Farmer class (by adding a constructor) without changing its behavior.

Rename a variable

Back in Chapter 3, we explained how choosing intuitive names for your classes, methods, fields, and variables makes your code a lot easier to understand. The IDE can really help you out when it comes to naming things in your code. Just right-click on any class, variable, field, property, namespace, constant — pretty much anything that you can name — and choose "Refactor >> Rename". You can also just use F2, which comes in handy because once you start renaming things, you find yourself doing it all the time.

We selected "beeControl" in the code from the simulator and renamed it. Here's what popped up:

This window lets you choose a new name for the item. If we renamed this, say, to "Bobbo", then the IDE would go through the code and **change every single occurrence** of it to "Bobbo".



The IDE does a really thorough job of renaming. If you rename a class, it'll change every statement that instantiates it or uses it. You can click on any occurrence of the name, anywhere in the code, and the IDE will make the change everywhere in your program.

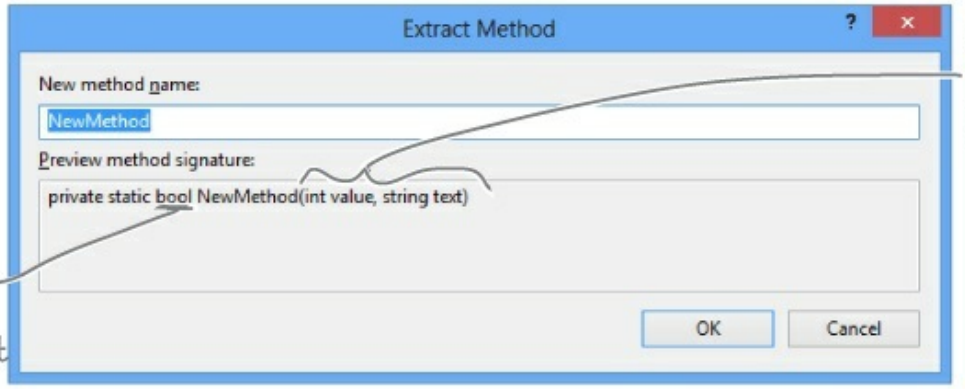
Consolidate a conditional expression

Here's a neat way to use the "Extract Method" feature. Open up any program, add a button, and add this code to its event handler:

```
private void button1_Click(object sender, EventArgs e) {
    int value = 5;
    string text = "Hi there";
    if (value == 36 || text.Contains("there"))
       MessageBox.Show("Pow!");
}
```

Select everything inside the `if` statement: `value == 36 || text.Contains("there")`. Then right-click on it and select "Refactor >> Extract Method...". Here's what pops up:

It's smart enough to figure out that it should create a static method, since it doesn't use any fields.



The expression uses two variables called value and text, so the IDE added parameters to the method using those names.

Not only will this make the code easier to read, but now you've got a new method that you can reuse elsewhere!

Every conditional expression evaluates to a bool, so the IDE will create a method that returns a bool and replace the conditional test with a call to that method.

#9. Anonymous types, anonymous methods, and lambda expressions

C# lets you create types and methods without using explicitly named declarations. A type or method that's declared without a name is called **anonymous**. These are very powerful tools — for example, LINQ wouldn't be possible without them. But it's a lot easier to master anonymous types, anonymous methods, and lambda expressions once you have a firm grasp on the language. So we only briefly covered anonymous types, and anonymous methods or lambda expressions didn't make the cut at all. Here's a quick introduction, so you can get started learning about them.

```
class Program {
    delegate void MyIntAndString(int i, string s);
    delegate int CombineTwoInts(int x, int y);

    static void Main(string[] args) {
        /*
         * In Chapter 14, you saw how the var keyword let the IDE determine the
         * type of an object at compile time.
         *
         * You can also create objects with anonymous types using var and new.
         *
         * You can learn more about anonymous types here:
         * http://msdn.microsoft.com/en-us/library/bb397696.aspx
         */

        // Create an anonymous type that looks a lot like a guy:
        var anonymousGuy = new { Name = "Bob", Age = 43, Cash = 137 };

        // When you type this in, the IDE's IntelliSense automatically picks up
        // the members -- Name, Age and Cash show up in the IntelliSense window.
        Console.WriteLine("{0} is {1} years old and has {2} bucks",
            anonymousGuy.Name, anonymousGuy.Age, anonymousGuy.Cash);
        // Output: Bob is 43 years old and has 137 bucks

        // An instance of an anonymous type has a sensible ToString() method.
        Console.WriteLine(anonymousGuy.ToString());
        // Output: { Name = Bob, Age = 43, Cash = 137 }

        /*
         * In Chapter 15, you learned about how you can use a delegate to reference
         * a method. In all of the examples of delegates that you've seen so far,
         * you assigned an existing method to a delegate.
         *
         * Anonymous methods are methods that you declare in a statement -- you
         * declare them using curly brackets { }, just like with anonymous types.
         *
         * You can learn more about anonymous methods here:
         * http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx
         */
```

```
        // Here's an anonymous method that writes an int and a string to the console.
        // Its declaration matches our MyIntAndString delegate (defined above), so
        // we can assign it to a variable of type MyIntAndString.
        MyIntAndString printThem = delegate(int i, string s)
                                   { Console.WriteLine("{0} - {1}", i, s); };
        printThem(123, "four five six");
        // Output: 123 - four five six

        // Here's another anonymous method with the same signature (int, string).
        // This one checks if the string contains the int.
        MyIntAndString contains = delegate(int i, string s)
                                  { Console.WriteLine(s.Contains(i.ToString())); };
        contains(123, "four five six");
        // Output: False

        contains(123, "four 123 five six");
        // Output: True

        // You can dynamically invoke a method using Delegate.DynamicInvoke(),
        // passing the parameters to the method as an array of objects.
        Delegate d = contains;
        d.DynamicInvoke(new object[] { 123, "four 123 five six" });
        // Output: True

        /*
         * A lambda expression is a special kind of anonymous method that uses
         * the => operator. It's called the lambda operator, but when you're
         * talking about lambda expressions you usually say "goes to" when
         * you read it. Here's a simple lambda expression:
         *
         *     (a, b) => { return a + b; }
         *
         * You could read that as "a and b goes to a plus b" -- it's an anonymous
         * method for adding two values. You can think of lambda expressions as
         * anonymous methods that take parameters and can return values.
         *
         * You can learn more about lambda expressions here:
         * http://msdn.microsoft.com/en-us/library/bb397687.aspx
         */

        // Here's that lambda expression for adding two numbers. Its signature
        // matches our CombineTwoInts delegate, so we can assign it to a delegate
        // variable of type CombineTwoInts. Notice how CombineTwoInts's return
        // type is int -- that means the lambda expression needs to return an int.
        CombineTwoInts adder = (a, b) => { return a + b; };
        Console.WriteLine(adder(3, 5));
        // Output: 8

        // Here's another lambda expression -- this one multiplies two numbers.
        CombineTwoInts multiplier = (int a, int b) => { return a * b; };
        Console.WriteLine(multiplier(3, 5));
        // Output: 15

        // You can do some seriously powerful stuff when you combine lambda
        // expressions with LINQ. Here's a really simple example:
        var greaterThan3 = new List<int> { 1, 2, 3, 4, 5, 6 }.Where(x => x > 3);
        foreach (int i in greaterThan3) Console.Write("{0} ", i);
        // Output: 4 5 6

        Console.ReadKey();
    }
}
```

#10. LINQ to XML

You've seen XML throughout the book as a format for files that represents complex data as text. The .NET Framework gives you some really powerful tools for creating, loading, and saving XML files. And once you've got your hands on XML data, you can use LINQ to query it. Add "`using System.Xml.Linq;`" to the top of a file and enter this method that generates an XML document with some Starbuzz Coffee customer loyalty data.

```
private static XDocument GetStarbuzzData() {
    XDocument doc = new XDocument(
        new XDeclaration("1.0", "utf-8", "yes"),
        new XComment("Starbuzz Customer Loyalty Data"),
        new XElement("starbuzzData",
            new XAttribute("storeName", "Park Slope"),
            new XAttribute("location", "Brooklyn, NY"),
            new XElement("person",
                new XElement("personalInfo",
                    new XElement("name", "Janet Venutian"),
                    new XElement("zip", 11215)),
                new XElement("favoriteDrink", "Choco Macchiato"),
                new XElement("moneySpent", 255),
                new XElement("visits", 50)),
            new XElement("person",
                new XElement("personalInfo",
                    new XElement("name", "Liz Nelson"),
                    new XElement("zip", 11238)),
                new XElement("favoriteDrink", "Double Cappuccino"),
                new XElement("moneySpent", 150),
                new XElement("visits", 35)),
            new XElement("person",
                new XElement("personalInfo",
                    new XElement("name", "Matt Franks"),
                    new XElement("zip", 11217)),
                new XElement("favoriteDrink", "Zesty Lemon Chai"),
                new XElement("moneySpent", 75),
                new XElement("visits", 15)),
            new XElement("person",
                new XElement("personalInfo",
                    new XElement("name", "Joe Ng"),
                    new XElement("zip", 11217)),
                new XElement("favoriteDrink", "Banana Split in a Cup"),
                new XElement("moneySpent", 60),
                new XElement("visits", 10)),
            new XElement("person",
                new XElement("personalInfo",
                    new XElement("name", "Sarah Kalter"),
                    new XElement("zip", 11215)),
                new XElement("favoriteDrink", "Boring Coffee"),
                new XElement("moneySpent", 110),
                new XElement("visits", 15))));
    return doc;
}
```

You can use an XDocument to create an XML file, and that includes XML files you can read and write using DataContractSerializer.

An XMLDocument object represents an XML document. It's part of the System.Xml.Linq namespace.

Use XElement objects to create elements under the XML tree.

Microsoft has a lot of great documentation about LINQ and LINQ to XML online. You can read more about LINQ to XML and classes in the System.Xml.Linq namespace here: http://msdn.microsoft.com/en-us/library/bb387098.aspx

Save and load XML files

You can write an `XDocument` object to the console or save it to a file, and you can load an XML file into it:

```
XDocument doc = GetStarbuzzData();
Console.WriteLine(doc.ToString());
doc.Save("starbuzzData.xml");
XDocument anotherDoc = XDocument.Load("starbuzzData.xml");
```

The XDocument object's Load() and Save() methods read and write XML files. And its ToString() method renders everything inside it as one big XML document.

Query your data

Here's a simple LINQ query that queries the Starbuzz data using its `XDocument`:

```
var data = from item in doc.Descendants("person")
    select new { drink = item.Element("favoriteDrink").Value,
        moneySpent = item.Element("moneySpent").Value,
        zipCode = item.Element("personalInfo").Element("zip").Value };
foreach (var p in data)
    Console.WriteLine(p.ToString());
```

The Descendants() method returns a reference to an object that you can plug right into LINQ.

You already know that LINQ lets you call methods and use them as part of the query, and that works really well with the Element() method.

Element() returns an XElement object, and you can use its properties to check specific values in your XML document.

And you can do more complex queries too:

```
var zipcodeGroups = from item in doc.Descendants("person")
    group item.Element("favoriteDrink").Value
    by item.Element("personalInfo").Element("zip").Value
        into zipcodeGroup
        select zipcodeGroup;
foreach (var group in zipcodeGroups)
    Console.WriteLine("{0} favorite drinks in {1}",
                    group.Distinct().Count(), group.Key);
```

Read data from an RSS feed

You can do some pretty powerful things with LINQ to XML. Here's a simple query to **read articles from our blog**:

```
XDocument ourBlog = XDocument.Load("http://www.stellman-greene.com/feed");
Console.WriteLine(ourBlog.Element("rss").Element("channel").Element("title").Value);
var posts = from post in ourBlog.Descendants("item")
    select new { Title = post.Element("title").Value,
        Date = post.Element("pubDate").Value};
foreach (var post in posts)
    Console.WriteLine(post.ToString());
```

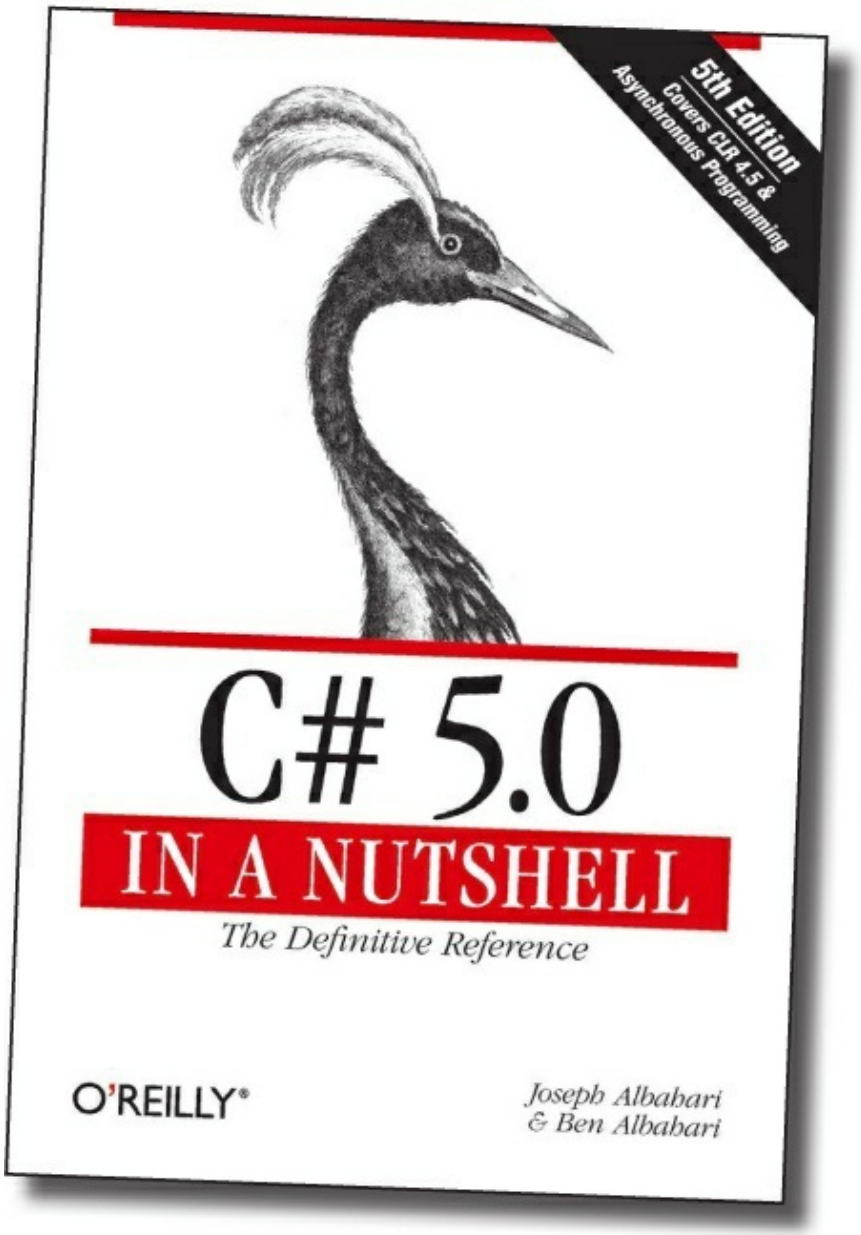
Create a new console application, make sure you've got "using System.Xml.Linq;" at the top, type this query into the Main() method, and check out what it prints to the console.

The XDocument.Load() method has several overloaded constructors. This one pulls XML data from a URL.

We used the URL of our blog, Building Better Software.
http://www.stellman-greene.com/

Did you know that C# and the .NET Framework can...

- Give you much more power over your data with advanced LINQ queries?
- Access websites and other network resources using built-in classes?
- Let you add advanced encryption and security to your programs?
- Create complex multithreaded applications?
- Let you deploy your classes so that other people can use them?
- Use regular expressions to do advanced text searching?
- And a whole lot more! You'll be amazed at how powerful C# can be.



There's a great book that explains it all!

NOTE

Joseph Albahari helped us out a whole lot by giving the first edition of this book a really thorough tech review. Thanks so much for all your help, Joe!

It's called ***C# 5.0 in a Nutshell*** by Joseph Albahari and Ben Albahari, and it's a thorough guide to everything that C# has to offer. You'll learn about advanced C# language features, you'll see all of the

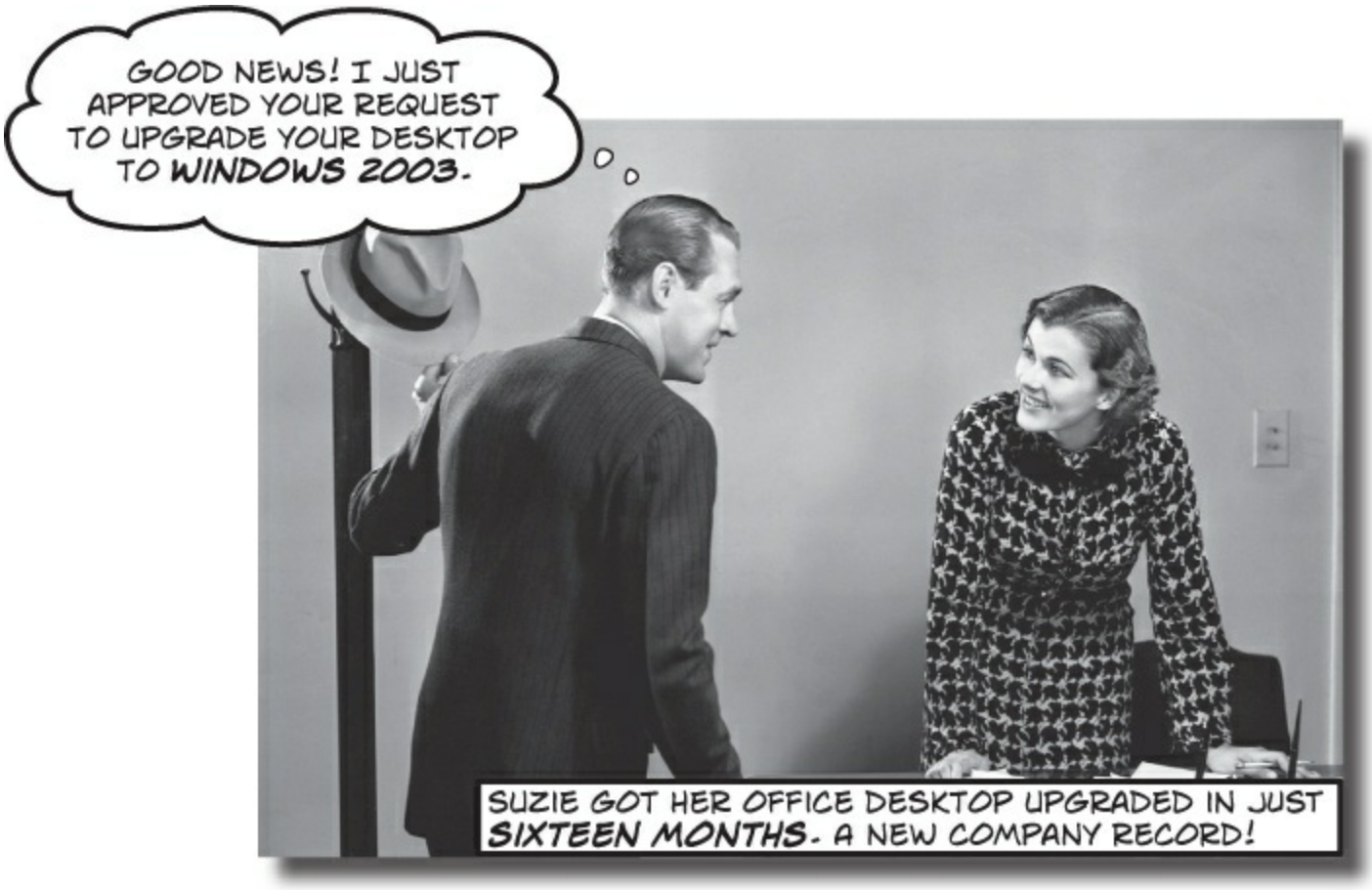
essential .NET Framework classes and tools, and you'll **learn more** about what's really going on under the hood of C#.

Check it out at: **http://www.oreilly.com/**.

Appendix B. Windows Presentation Foundation: WPF Learner's Guide to Head First C#

NOTE

There are many projects in Head First C# where you build Windows Store apps that require Windows 8. In this appendix, you'll use WPF to build them as desktop apps instead.



Not running Windows 8? Not a problem.

We wrote many chapters in the third edition of *Head First C#* using the latest technology available from Microsoft, which **requires Windows 8 and Visual Studio 2013**. But what if you're using this book at work, and you can't install the latest version? That's where **Windows Presentation Foundation** (or **WPF**) comes in. It's an older technology, so it works with Visual Studio 2010 and 2008 running on Windows editions as mature as 2003. But it's also a core C# technology, so even if you're running Windows 8 it's a good idea to **get some experience with WPF**. In this appendix, we'll **guide you through building** *most* of the Windows Store projects in the book using WPF.

Why you should learn WPF

Windows Presentation Foundation, or WPF, is a technology that's used to build user interfaces for programs written in .NET. WPF programs typically run on the Windows desktop and display their user interfaces in windows. WPF is one of the most popular technologies for developing Windows software, and familiarity with WPF is considered by many employers to be a required skill for professional C# and .NET developers.

WPF programs use XAML (Extensible Application Markup Language) to lay out their UIs. This is great news for *Head First C#* readers who have been reading about Windows Store apps. Most of the Windows Store projects in the book can be built for WPF **with few or no modifications to the XAML code**.

> **NOTE**
>
> Some things, like app bars and page navigation, are specific to Windows Store apps. In this appendix, we show you WPF alternatives wherever possible.

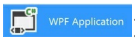
Every C# developer should work with WPF.

Almost every programming language can be used in lots of different environments and operating systems, and C# is no exception. If your goal is to improve as a C# developer, you should go out of your way to work with as many different technologies as possible. And WPF in particular is especially important for C# developers, because there are many programs that use WPF in companies, and this will continue for a long time. If your goal is to use C# in a professional environment, WPF is technology you'll want to list on your resumé.

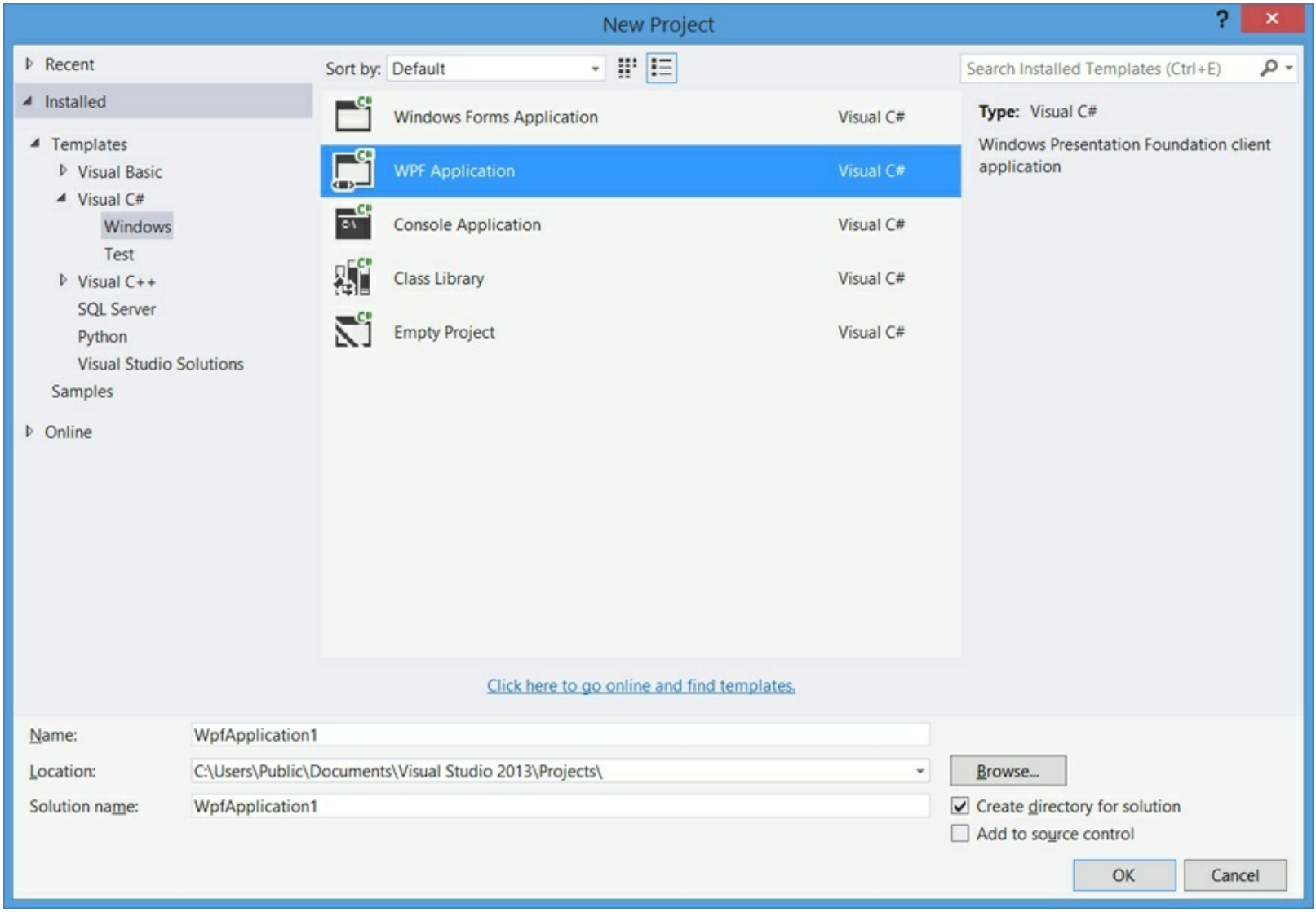
Learning WPF is also great for a hobby programmer who's using Windows 8 and can build all of the code in *Head First C#*. One of the most effective learning tools you have as a developer is seeing **the same problem solved in different ways**. This appendix will guide you through building many of the projects in *Head First C#* using WPF. Seeing those projects built in WPF and Windows 8 will give you valuable perspective, and that's one of the things that **helps turn good programmers into great**

developers.

Build WPF projects in Visual Studio

Creating a new WPF application in Visual Studio works just like creating other kinds of desktop applications. If you're using Visual Studio Express 2013, make sure you're using Visual Studio 2013 Express for *Desktop* (the edition for Windows 8 will not create WPF projects). You can also create programs using Visual Studio 2013 Professional, Premium, or Ultimate. When you create a new project, Visual Studio displays a "New Project" dialog. Make sure you select **Visual C#**, and then choose 

:



You can also create C# WPF applications using all editions of Visual Studio 2010, Visual C# 2010 Express, and Visual Studio 2008. Note that if you use the Express editions of Visual Studio 2010 or 2008, the project files are initially created in a temporary folder and are not saved to the location specified in the New Project dialog until you use Save or Save All to save your files.

NOTE

WPF can also be used to build XAML browser applications that run inside Internet Explorer and other browsers. We won't be covering it in this appendix, but you can learn more about it here: http://msdn.microsoft.com/en-us/library/aa970060.aspx

Microsoft has yet another technology that also uses XAML. It's called Silverlight, and you can read about it here: http://www.microsoft.com/silverlight/

Did you find an error in this appendix? Please submit it using the Errata page for Head First C# (3rd edition) so we can fix it as quickly as possible!

http://www.oreilly.com/catalog/errata.csp?isbn=0636920027812

How to use this appendix

This appendix contains complete replacements for pages in *Head First C# (3rd edition)*. We've divided this appendix up into individual guides for each chapter, starting with an overview page that has specific instructions for how to work through that chapter: what pages to replace in the chapter, what to read in it, and any specific instructions to help you get the best learning experience.

NOTE

If you're using an old version of Visual Studio, you'll be able to do these projects... but things will be a little harder for you.

The team at Microsoft did a really good job of improving the user interface of Visual Studio 2013, especially when it comes to editing XAML. One important feature of *Head First C#* is its use of the Visual Studio IDE as a tool for teaching, learning, and exploration. This is why we strongly recommend that you use the latest version of Visual Studio if possible.

However, we do understand that some readers cannot install Visual Studio 2013. (For example, a lot of our readers are using a computer provided by an employer, and do not have administrative privileges to install new software.) We still want you to be able to use our book, even if you're stuck using an old version of Visual Studio! We'll do our best to give you as much guidance as we can. But we also need to strike a balance here, because we're being careful not to compromise the learning for the majority of our readers who are using the latest version of Visual Studio.

If you're using Visual Studio 2010 or earlier, and you find yourself stuck because the IDE's user interface doesn't look right or menu options aren't where you expect them to be, **we recommend that you enter the XAML and C# code by hand** — or even better, copy it and paste it into Visual Studio. Once the XAML is correct, it's often easier to track down the feature in the IDE that generated it.

NOTE

One more thing. This appendix has replacements for pages that you'll find in the printed or PDF version this book, and you can find those pages using their page numbers. However, if you're using a Kindle or another eBook reader, you might not be able to use the page numbers. Instead, just use the section heading to look up the section to replace. For example, this appendix has replacements for pages 72 and 73 section called "Build an app from the ground up," which you can find in your eBook reader's Table of Contents underneath Chapter 2. (Exercises like the one on page 83 and the solution on page 85 might not show up in your reader's Table of Contents, but you'll get to the exercises as you go through each chapter.) This will be much easier for you if you download the PDF of this appendix from the book's website.

Chapter 1

Build a game, and get a feel for the IDE.

The first project in the book walks you through building a complete — and fun! — video game. The goal of the project is to help you get used to creating user interfaces and writing C# code using the Visual Studio IDE.

We recommend that you read through Note in the main part of the book, and then flip to the next page in this appendix. We designed Start with a blank application–Add a splash screen and a tile in this appendix so that they can be 100% replacements for the corresponding pages in the book. Once you've finished building the WPF version of *Save the Humans*, you can go on to Chapter 2 in the book.

NOTE

The screenshots in this chapter are from Visual Studio 2013 for Windows Desktop, the latest version of Visual Studio available at this time. If you're using Visual Studio 2010, some of the menu options and windows in the IDE will be different. We'll give you guidance to help you find the right menu options.

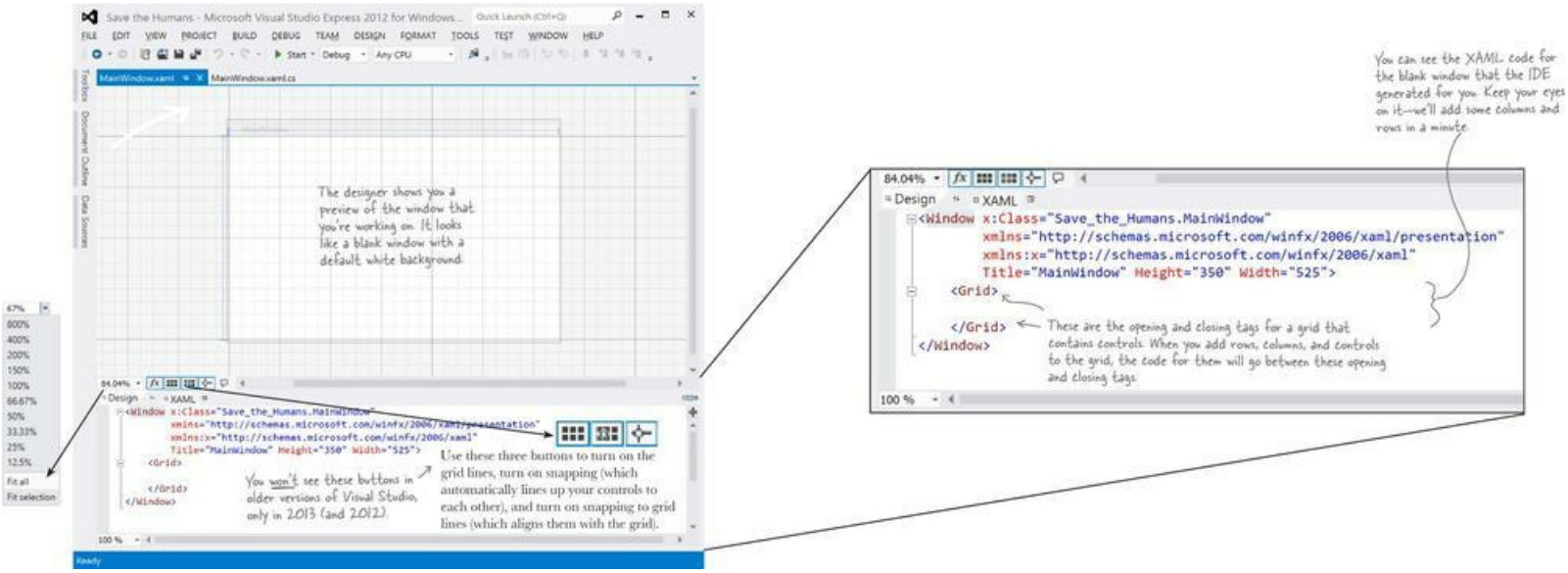
We worked really hard to keep the page flipping to a minimum, because by reducing distractions we make it easier for you to learn important C# concepts. After you read the first 11 pages of Chapter 1, you won't have to flip back to the main part of the book at all for the rest of the chapter. Then there are just five pages that you need in this appendix for Chapter 2. After that, the book concentrates on building desktop applications, which you can build with any version of Windows. You won't need this appendix again until you get to Chapter 10.

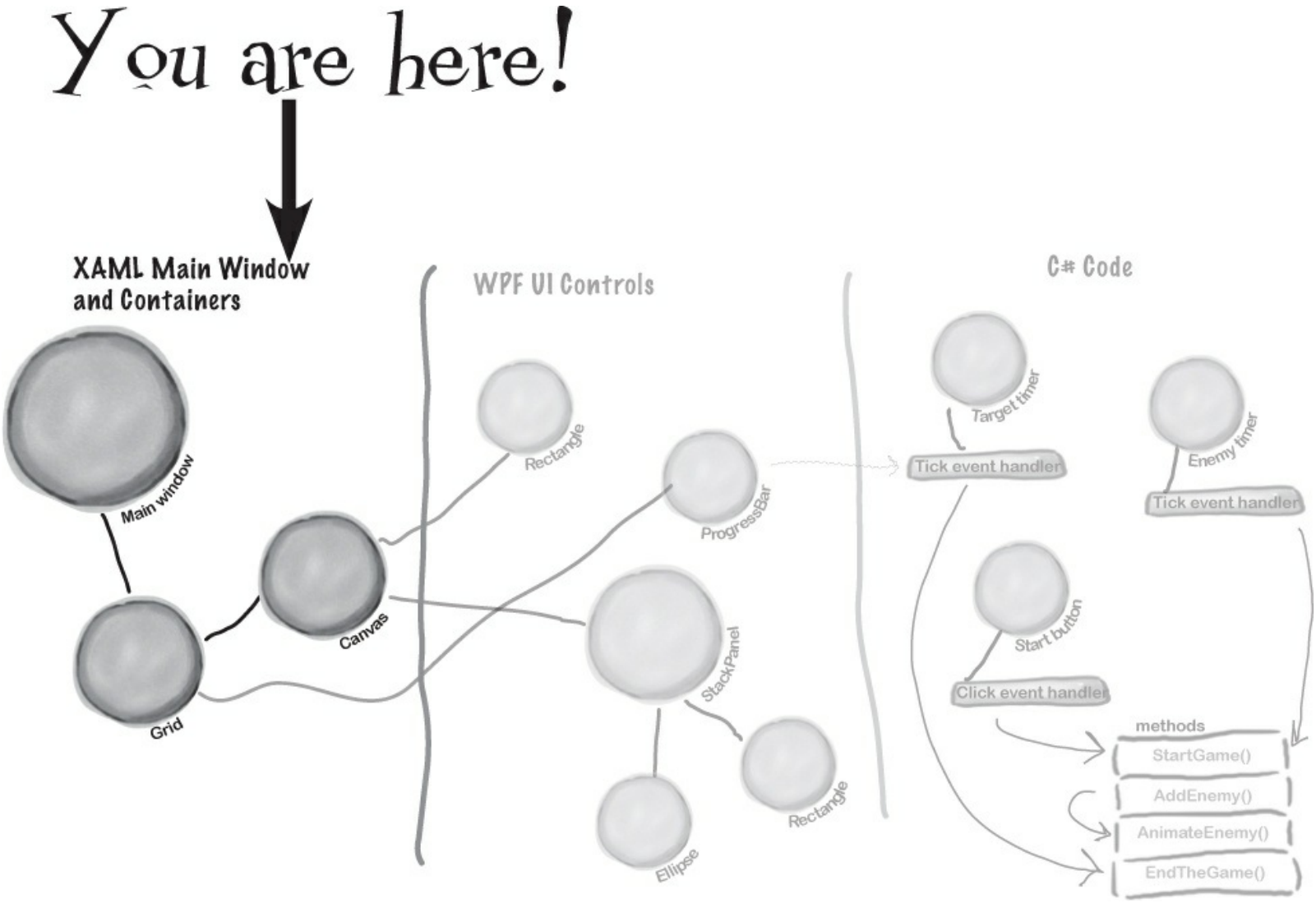
Start with a blank application

Every great app starts with a new project. Choose New Project from the File menu. Make sure you have Visual C#→Windows selected and choose **WPF Application** as the project type. Type "Save the Humans" as the project name.

> **NOTE**
>
> If your code filenames don't end in ".cs" you may have accidentally created a JavaScript, Visual Basic, or Visual C++ program. You can fix this by closing the solution and starting over. If you want to keep the project name "Save the Humans," then you'll need to delete the previous project folder.

- ❶ Your starting point is the **Designer window**. Double-click on *MainWindow.xaml* in the Solution Explorer to bring it up (if it's not already displayed). Find the zoom drop-down in the lower-left corner of the designer and choose "Fit all" to zoom it out.





The bottom half of the Designer window shows you the XAML code. It turns out your "blank" window isn't blank at all — it contains a **XAML grid**. The grid works a lot like a table in an HTML page or Word document. We'll use it to lay out our windows in a way that lets them grow or shrink to different screen sizes and shapes.

This part of the project has steps numbered ① to ③ . Flip the page to keep going!

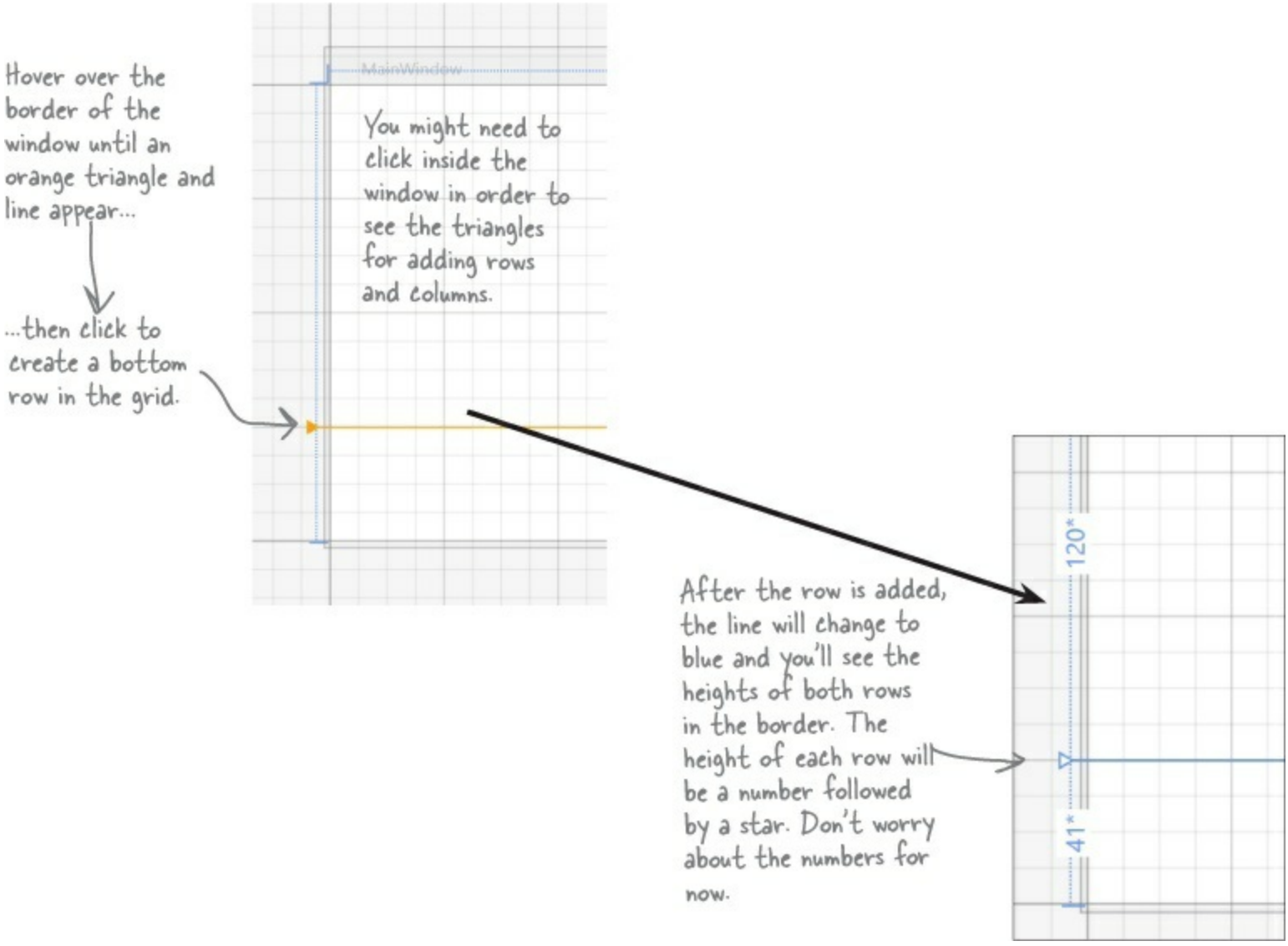
THIS PROJECT CLOSELY FOLLOWS CHAPTER 1.

We want to give you a solid learning foundation, so we've designed this project so that it can replace Start with a blank application-Publish your app of *Head First C#*. Other projects in this appendix will give you all the information that you need to adapt the material in the book. So even when we don't give you oneto-one page replacements, we'll make sure you get all the information you need to do the projects.

- ❷ Your app will be a grid with two rows and three columns, with one big cell in the middle that will contain the play area. Start defining rows by hovering over the border of the window until a line and triangle appear:

Over the next few pages you'll explore a lot of different features in the Visual Studio IDE, because we'll be using the IDE as a powerful tool for learning and teaching. You'll use the IDE throughout the book to explore C#. That's a really effective way to get it into your brain!

WPF apps often need to adapt to different window sizes displayed at different screen resolutions.



Laying out the window using a grid's columns and rows allows your program to automatically adjust to the window size.

THERE ARE NO DUMB QUESTIONS

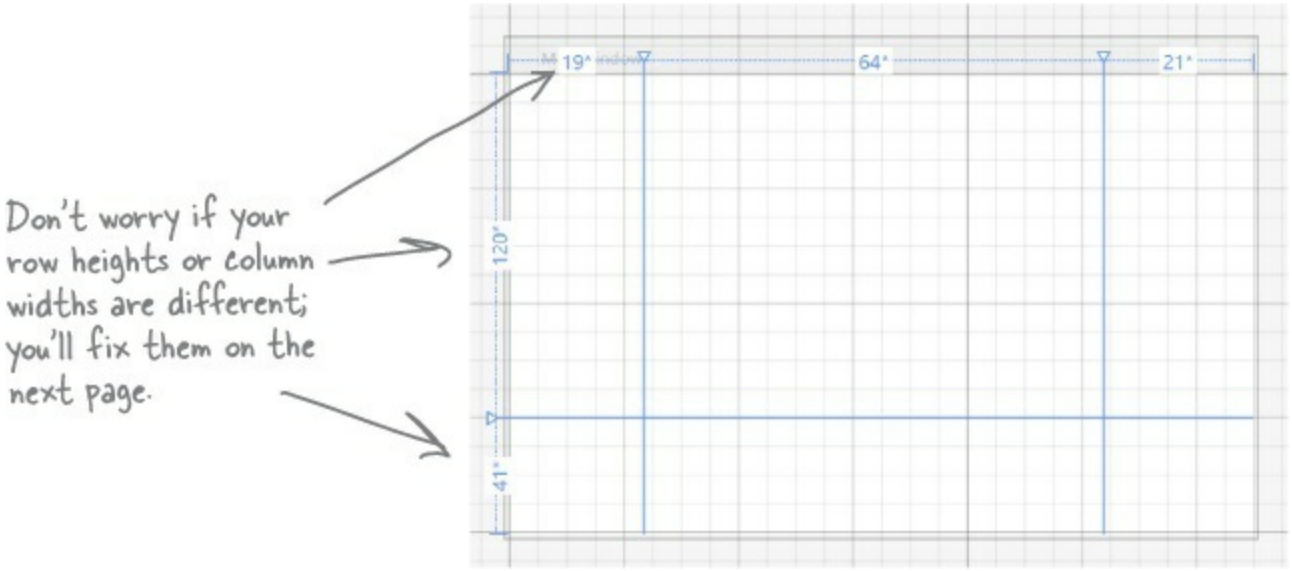
Q: **Q: But it looks like I already have many rows and columns in the grid. What are those gray lines?**

A: **A:** The gray lines are just Visual Studio giving you a grid of guidelines to help you lay your controls out evenly in the window. You can turn them on and off with the button. None of the lines you see in the designer show up when you run the app outside of Visual Studio. But when you clicked and created a new row, you actually altered the XAML, which will change the way the app behaves when it's compiled and executed.

Q: **Q: Wait a minute. I wanted to learn about C#. Why am I spending all this time learning about XAML?**

A: **A:** Because WPF apps built in C# almost always start with a user interface that's designed in XAML. That's also why Visual Studio has such a good XAML editor — to give you the tools you need to build stunning user interfaces. Throughout the book, you'll learn how to build other types of programs with C#: Windows Store apps, which use XAML, and desktop applications and console applications, which don't. Seeing all of these different technologies will give you a deeper understanding of programming with C#.

- ❸ Do the same thing along the top border of the window — except this time create two columns, a small one on the left-hand side and another small one on the right-hand side. Don't worry about the row heights or column widths — they'll vary depending on where you click. We'll fix them in a minute.



When you're done, look in the XAML window and go back to the same grid from the previous page. Now the column widths and row heights match the numbers on the top and side of your window.

```
84.04%
Design   XAML
<Window x:Class="Save_the_Humans.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="19*"/>
            <ColumnDefinition Width="64*"/>
            <ColumnDefinition Width="21*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="120*"/>
            <RowDefinition Height="41*"/>
        </Grid.RowDefinitions>

    </Grid>
</Window>
100 %
```

Here's the width of the left column you created in step 3—the width matches the width that you saw in the designer. That's because the IDE generated this XAML code for you.

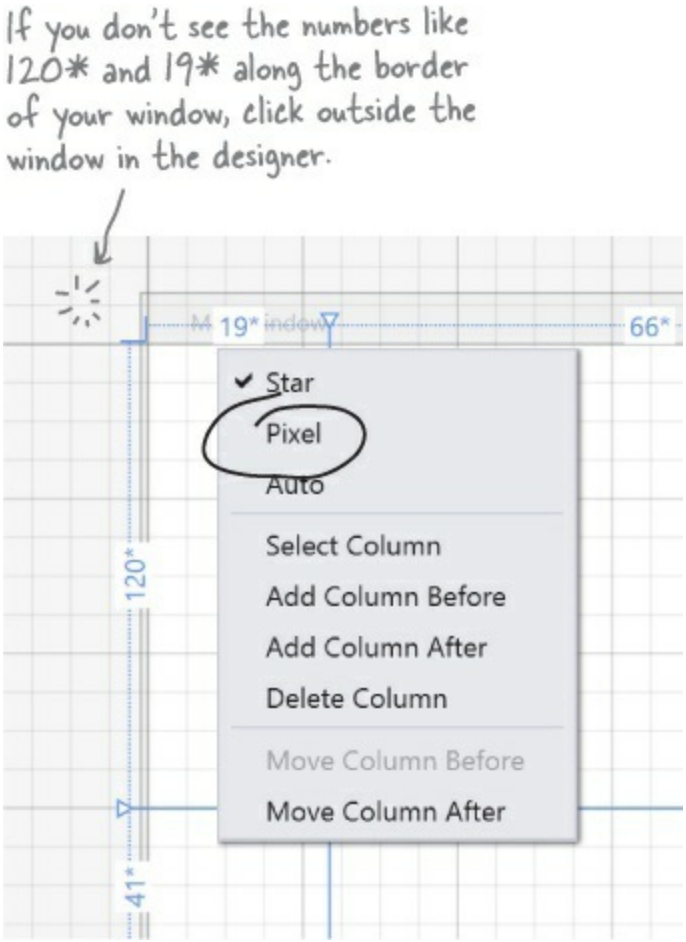
Your grid rows and columns are now added!

XAML grids are **container controls**, which means they hold other controls. Grids consist of rows and columns that define cells, and each cell can hold other XAML controls that show buttons, text, and shapes. A grid is a great way to lay out a window, because you can set its rows and columns to resize themselves based on the size of the screen.

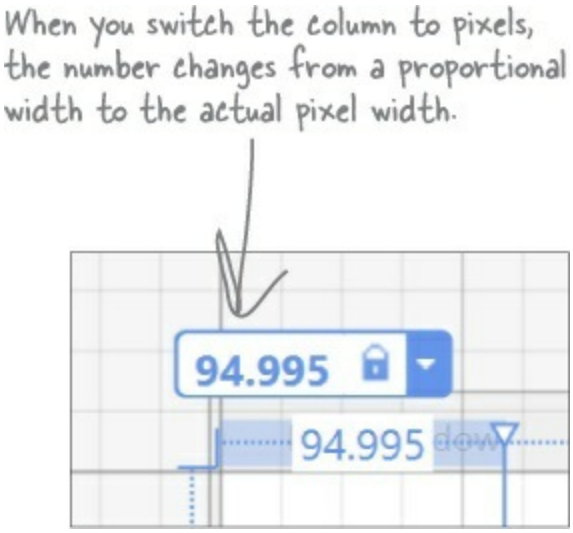
Set up the grid for your window

Your program needs to be able to work on different sized windows, and using a grid is a great way to do that. You can set the rows and columns of a grid to a specific pixel height. But you can also use the **Star** setting, which keeps them the same size proportionally — to one another and also to the window — no matter how big the window or resolution of the display.

- **❶ SET THE WIDTH OF THE LEFT COLUMN.**
  Hover over the number above the *leftmost* column until a drop-down menu appears. Choose Pixel to change the star to a lock, and then click on the number to change it to `140`. Your column's number should now look like this:



- **❷ REPEAT FOR THE RIGHT COLUMN AND THE BOTTOM ROW.**
  Make the right column 160 pixels and the bottom row 150 by choosing Pixel and typing `160` or `150` into the box.



Set your columns or rows to Pixel to give them a fixed width or height. The Star setting lets a row or column grow or shrink proportionally to the rest of the grid. Use this setting in the designer to alter the Width or Height property in the XAML. If you remove the Width or Height property, it's the same as setting the property to 1*.

RELAX

It's OK if you're not a pro at app design...*yet*.

We'll talk a lot more about what goes into designing a good app later on. For now, we'll walk you through building this game. By the end of the book, you'll understand exactly what all of these things do!

- **❸ MAKE THE CENTER COLUMN THE DEFAULT SIZE.**
  Make sure that the center column width is set to 1*. If it isn't, click on the number above the center column and enter 1. Don't use the drop-down (leave it star) so it looks like the picture below. Then make sure to look back at the other columns to make sure the IDE didn't resize them. If it did, just change them back to the widths you set in steps 1 and 2.

NOTE

XAML and C# are case sensitive! Make sure your uppercase and lowercase letters match example code.

- **❹ LOOK AT YOUR XAML CODE!**
  Click on the grid to make sure it's selected, then look in the XAML window to see the code that you built.

```
<Window x:Class="Save_the_Humans.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="140"/>
            <ColumnDefinition/>
            <ColumnDefinition Width="160"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition Height="150"/>
        </Grid.RowDefinitions>
    </Grid>
</Window>
```

The <Grid> line at the top means everything that comes after it is part of the grid.

This is how a column is defined for a XAML grid. You added three columns and two rows, so there are three ColumnDefinition tags and two RowDefinition tags.

You used the designer to set the height of the bottom row to 150 pixels.

In a minute, you'll be adding controls to your grid, which will show up here, after the row and column definitions.

You used the column and row drop-downs to set the Width and Height properties.

NOTE

If you're using Visual Studio 2010, the IDE looks different. When you hover over a column size, you'll see this box to select pixel or star:

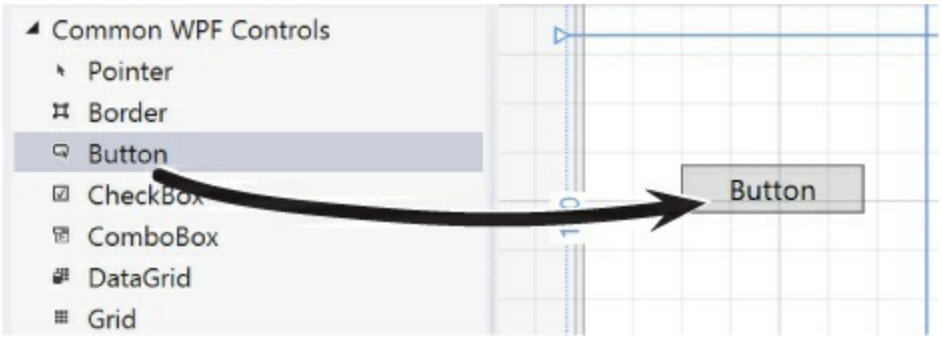
It's possible to edit the column sizes in the designer using the older versions of the IDE, but it's not nearly as easy to do. We recommend that if you're using an older version of the IDE, you create the columns and rows, and then edit the XAML row and column definitions by hand.

Add controls to your grid

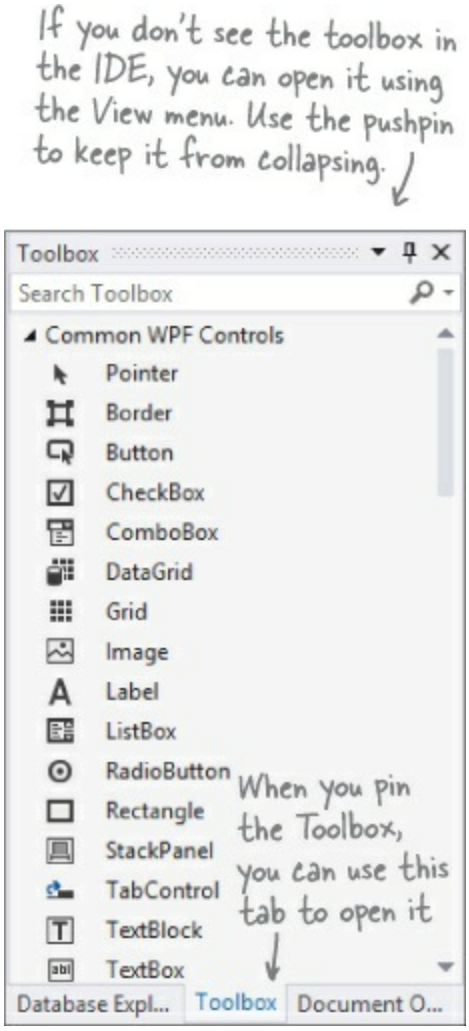
Ever notice how programs are full of buttons, text, pictures, progress bars, sliders, drop-downs, and menus? Those are called **controls**, and it's time to add some of them to your app — *inside* the cells defined by your grid's rows and columns.

- ❶ Expand the **Common WPF Controls** section of the toolbox and drag a

into the **bottom-left cell** of the grid.



Then look at the bottom of the Designer window and have a look at the **XAML tag** that the IDE generated for you. You'll see something like this — your margin numbers will be different depending on where in the cell you dragged it, and the properties might be in a different order.

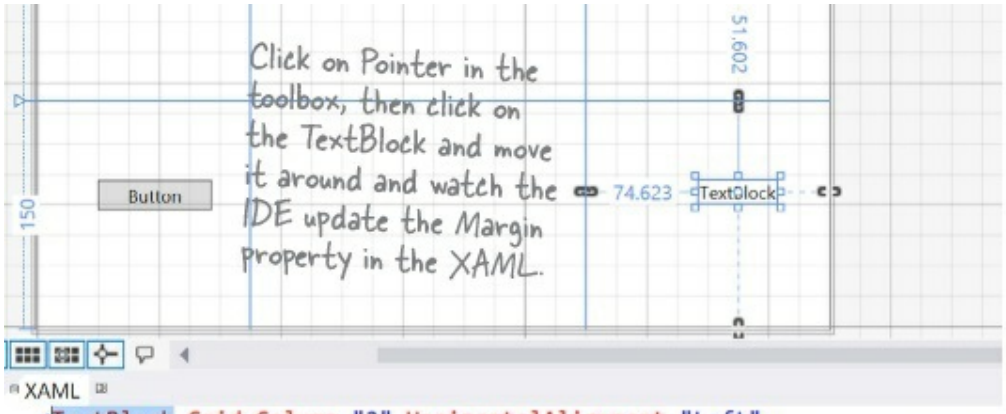


The XAML for the button starts here, with the opening tag.

These are properties. Each property has a name, followed by an equals sign, followed by its value.

```
<Button Content="Button" HorizontalAlignment="Left"
        Margin="40,52,0,0" Grid.Row="1" VerticalAlignment="Top" Width="75" />
```

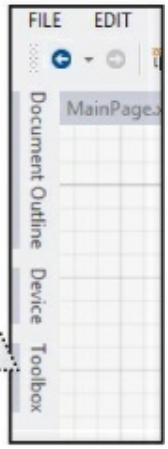
- ❷ Drag a TextBlock into the **lower-right cell** of the grid. Your XAML will look something like this. See if you can figure out how it determines which row and column the controls are placed in.



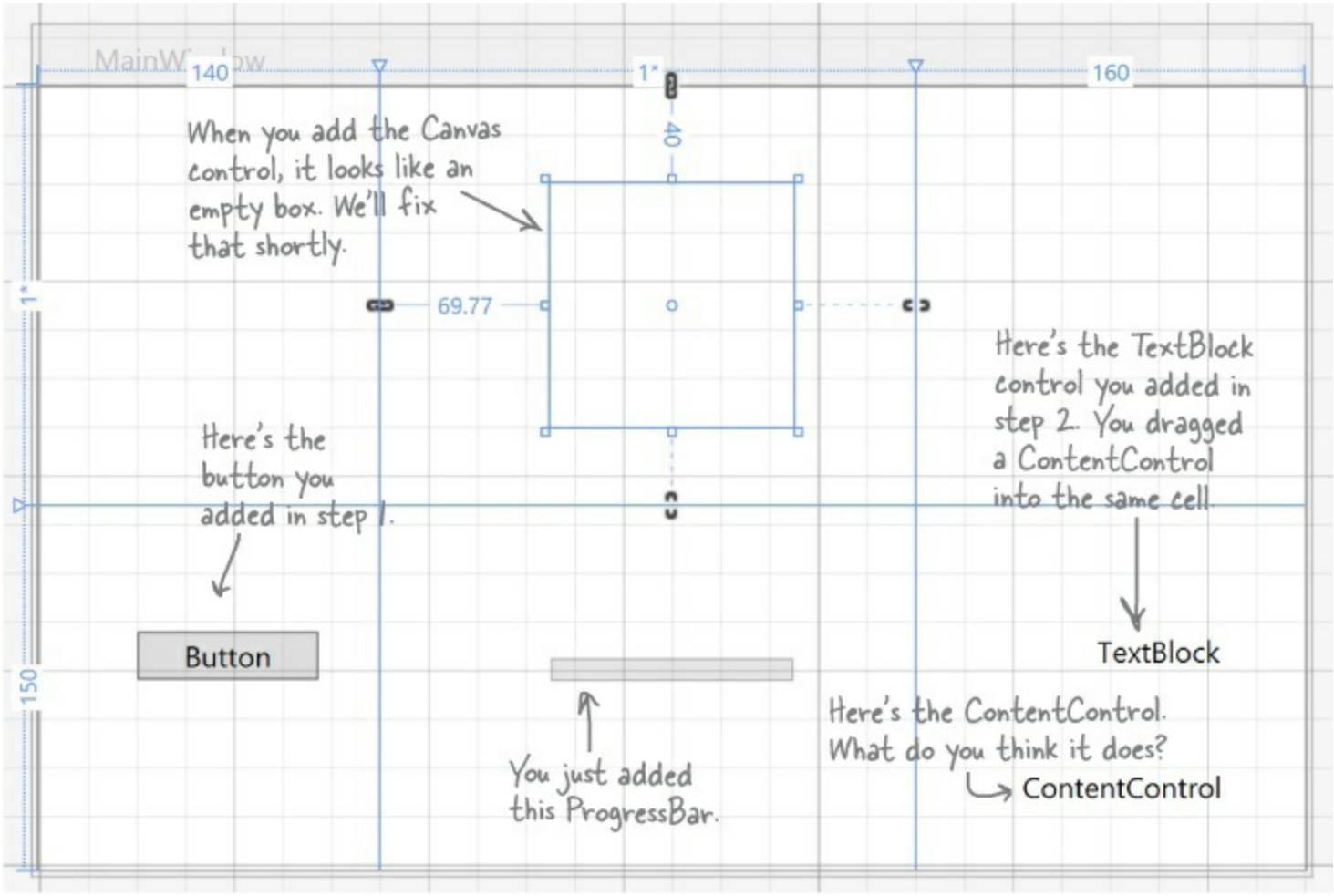

```
<TextBlock Grid.Column="2" HorizontalAlignment="Left"
           Margin="74.623,51.602,0,0" Grid.Row="1" TextWrapping="Wrap"
           Text="TextBlock" VerticalAlignment="Top"/>
```

We added line breaks to make the XAML easier to read. You can add line breaks, too. Give it a try!

If you don't see the toolbox, try clicking on the word "Toolbox" that shows up in the upper-left corner of the IDE. If it's not there, select Toolbox from the View menu to make it appear.



- ❸ Next, expand the **All WPF Controls** section of the toolbox. Drag a ProgressBar into the bottom-center cell, a ContentControl into the bottom-right cell (make sure it's **below** the TextBlock you already put in that cell), and a Canvas into the top center cell. Your window should now have controls on it (don't worry if they're placed differently than the picture below; we'll fix that in a minute):



- ❹ You've got the Canvas control currently selected, since you just added it. (If not, use the pointer to select it again.) Look in the XAML window:

```
<Canvas Grid.Column="1" HorizontalAlignment="Left" Height="100"...
```

It's showing you the XAML tag for the Canvas control. It starts with `<Canvas` and ends with `/>`, and between them it has properties like `Grid.Column="1"` (to put the Canvas in the center column) and `Height="100"` (to set its height in pixels). Try clicking *in both the grid and the XAML window* to select different controls.

When you drag a control out of the toolbox and onto your window, the IDE automatically generates XAML to put it where you dragged it.

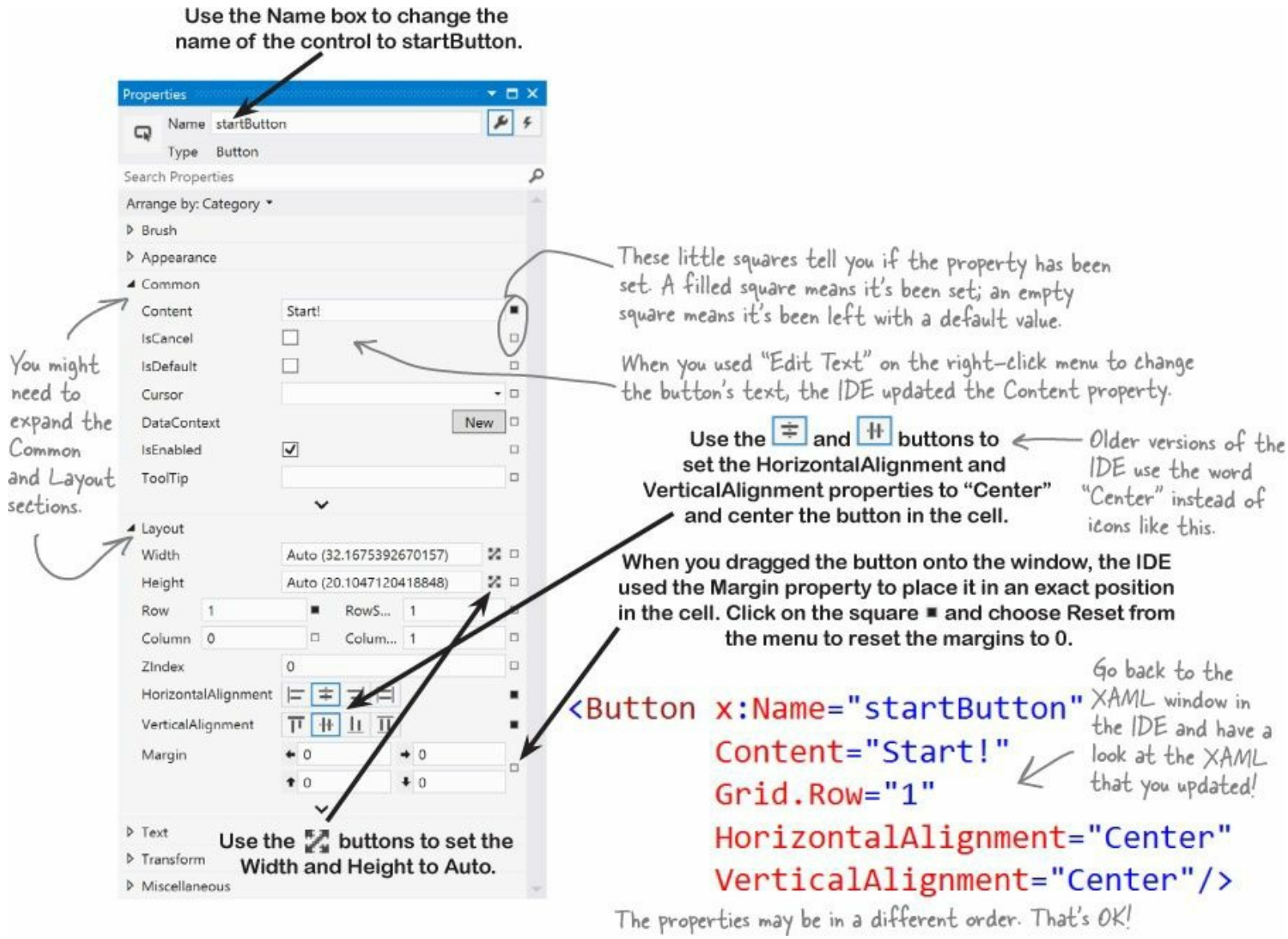
Use properties to change how the controls look

The Visual Studio IDE gives you fine control over your controls. The **Properties window** in the IDE lets you change the look and even the behavior of the controls on your window.

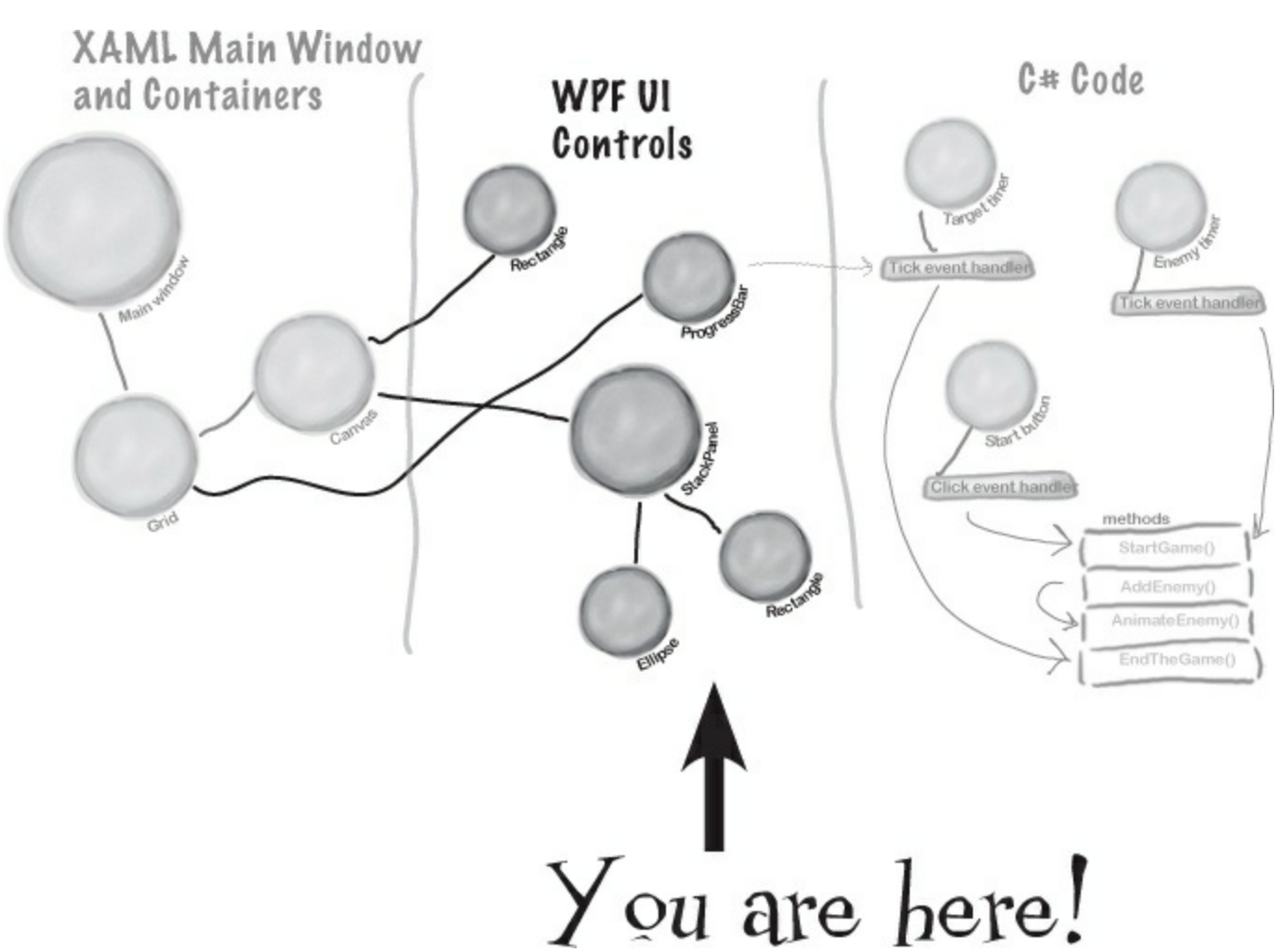
- **❶ Change the text of the button.**
  Right-click on the button control that you dragged onto the grid and choose **Edit Text** from the menu. Change the text to: `Start!` and see what you did to the button's XAML:

> **NOTE**
>
> When you're editing text, use the Escape key to finish. This works for other things in the IDE, too.

- **❷ Use the Properties window to modify the button.**
  Make sure the button is selected in the IDE, and then look at the Properties window in the lower-right corner of the IDE. Use it to change the name of the control to `startButton` and center the control in the cell. Once you've got the button looking right, **right-click on it and choose View Source** to jump straight to the `<Button>` tag in the XAML window.



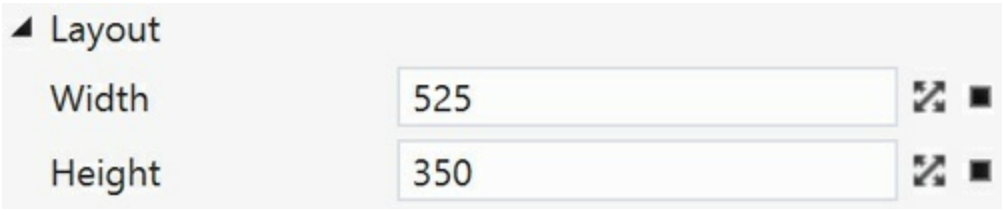
You can use Edit→Undo (or Ctrl-Z) to undo the last change. Do it several times to undo the last few changes. If you selected the wrong thing, you can choose Select None from the Edit menu to deselect. You can also hit Escape to deselect the control. If it's living inside a container like a StackPanel or Grid, hitting Escape will select the container, so you may need to hit it a few times.



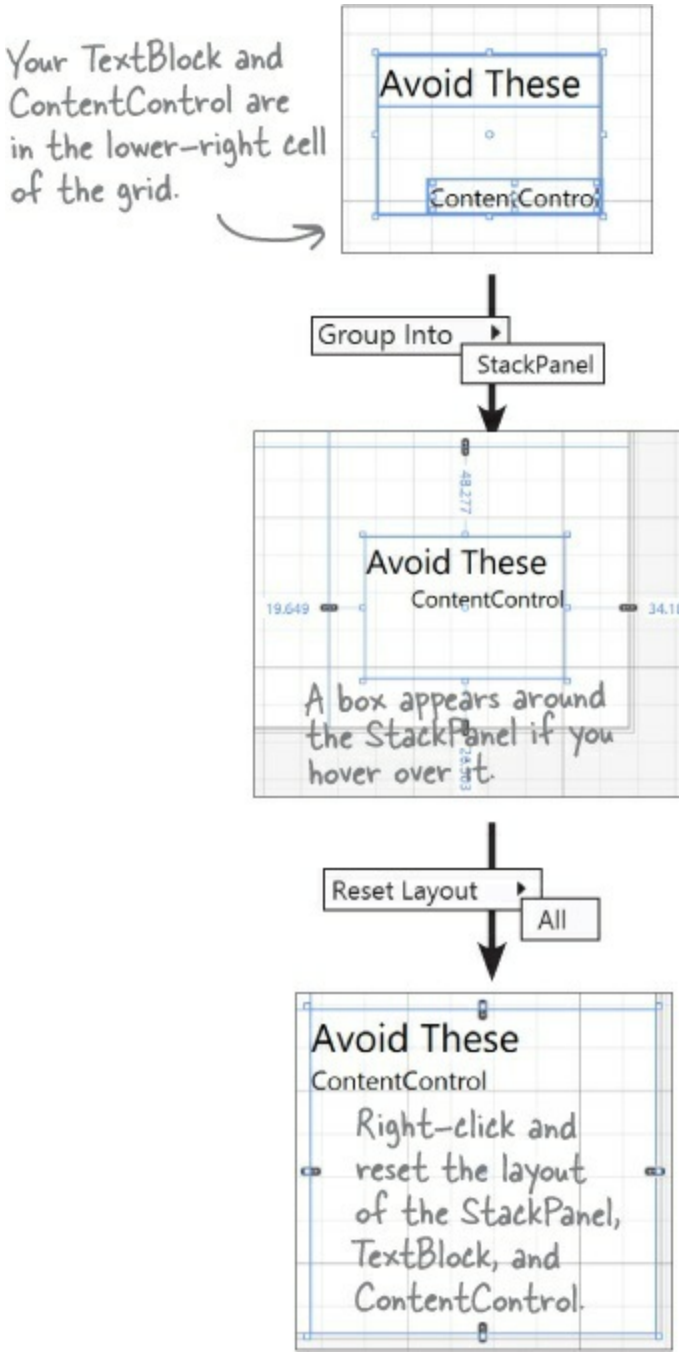
- **❸ Change the size and title of the window.**
  Select any of the controls. Then hit Escape, and keep hitting Escape until the outer <Window> tag is displayed in the XAML editor:

```
<Window x:Class="Save_the_Humans.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
```

  Click `<Window` in the XAML editor. The <Window> tag has properties for Height and Width. Look for their corresponding values in the Properties window in the IDE:



  Set the width to 1000 and height to 700, and the window immediately resizes itself to the new size. You can use the "Fit all" option in the Zoom drop-down to show the whole window in the designer. Notice how the center column and top row resized themselves to fit the new window, while the other rows and columns kept their pixel sizes. Then expand the Common section in the Properties window and **set the Title property to `Save the Humans`**. You'll see the window title get updated.



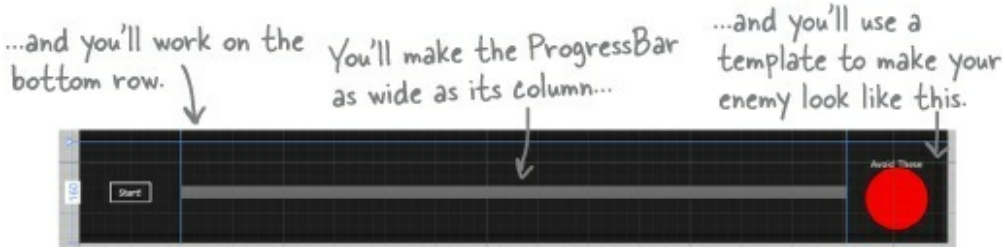
- **❹ Update the TextBlock to change its text and its font size.**
  Use the Edit Text right-mouse menu option to change the TextBlock so it says `Avoid These` (hit Escape to finish editing the text). Then expand the Text section of the Properties window and change the font size to `18 px`. This may cause the text to wrap and expand to two lines. If it does, drag the TextBlock to make it wider.
- **❺ Use a StackPanel to group the TextBlock and ContentControl.**
  Make sure that the TextBlock is near the top of the cell, and the ContentControl is near the bottom. **Click and drag to select both the TextBlock and ContentControl, and then right-click**. Choose Group Into from the pop-up menu, then choose StackPanel. This adds a new control to your form: a **StackPanel control**. You can select the StackPanel by clicking between the two controls.
  The StackPanel is a lot like the Grid and Canvas: its job is to hold other controls (it's called a "container"), so it's not visible on the form. But since you dragged the TextBlock to the top of the cell and the ContentControl to the bottom, the IDE created the StackPanel so it fills up most of the cell. Click in the middle of the StackPanel to select it, then right-click and choose Layout and Reset All to quickly reset its properties, which will set its vertical and horizontal alignment to

Stretch. Right-click on the TextBox and ContentControl to reset their properties as well. While you have the ContentControl selected, set its vertical and horizontal alignments to Center.

The user interface for editing colors in earlier versions of Visual Studio is not as advanced, but you should still be able to set the colors so they look correct. The Document Outline window is also a little more primitive, but it still works. However, **there is *not* an easy way to visually create a template in Visual Studio 2010.** The easiest way to do this in the old version of the IDE is to copy the entire `<Window.Resources>` section (up through the closing `</Window.Resources>` tag) from the downloadable source code and paste it into your XAML just above the opening `<Grid>` tag. **Make sure you download the code from the WPF folder!** Then you can select the ContentControl and use the Properties window to set the Template property to EnemyTemplate. Your enemies will already look like evil aliens, so make sure you still read Dragging humans onto enemies ends the game and Your game is now playable.

Controls make the game work

Controls aren't just for decorative touches like titles and captions. They're central to the way your game works. Let's add the controls that players will interact with when they play your game. Here's what you'll build next:



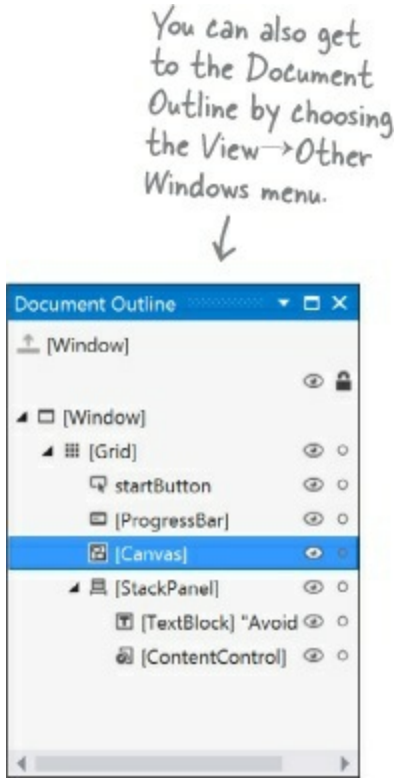
- **❶ Update the ProgressBar.**
  Right-click on the ProgressBar in the bottom-center cell of the grid, choose the **Layout** menu option, and then choose **Reset All** to reset all the properties to their default values. Use the Height box in the Layout section of the Properties window to set the `Height` to `20`. The IDE stripped all of the properties from the XAML, and then added the new `Height`:

```
<ProgressBar Grid.Column="1" Grid.Row="2" Height="20"/>
```

- **❷ Turn the Canvas control into the gameplay area.**
  Remember that Canvas control that you dragged into the center square? It's hard to see it right now because a Canvas control is invisible when you first drag it out of the toolbox, but there's an easy way to find it. Click the very small button above the XAML window to bring up the **Document Outline**. Click on [Canvas] to select the Canvas control.

Make sure the Canvas control is selected, then **use the Name box** in the Properties window to set the name to `playArea`.

> **NOTE**
>
> Once you change the name, it'll show up as playArea instead of [Canvas] in the Document Outline window.

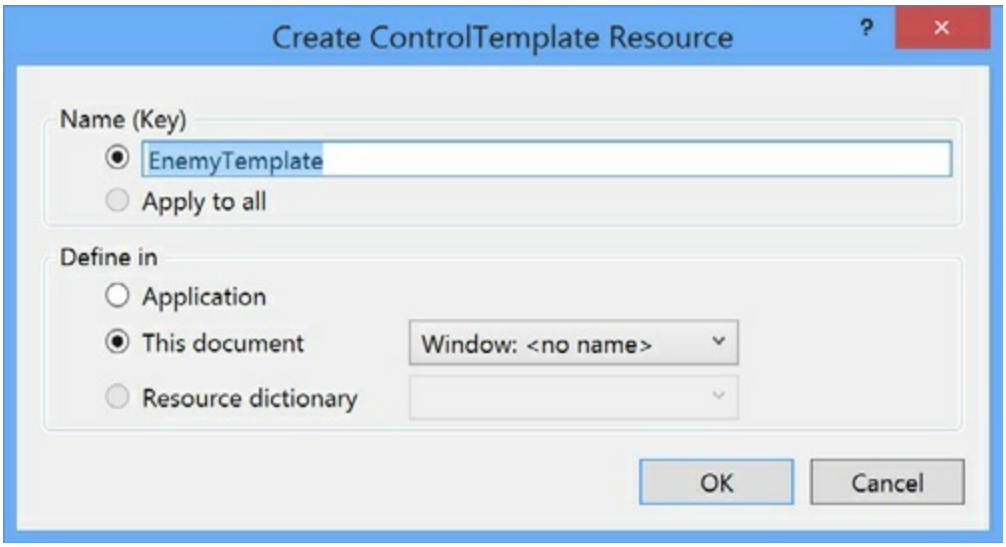


After you've named the Canvas control, you can close the Document Outline window. Then use the and buttons in the Properties window to set its vertical and horizontal alignments to Stretch, reset the margins, and click both buttons to set the `Width` and `Height` to Auto. Then set its Column to 0, and its ColumnSpan (next to Column) to 3.

Finally, open the **Brush** section of the Properties window and use the button to give it a **gradient**. Choose the starting and ending colors for the gradient by clicking each of the tabs at the

bottom of the color editor and then clicking a color.

- **❸ Create the enemy template.**
  Your game will have a lot of enemies bouncing around the screen, and you're going to want them all to look the same. Luckily, XAML gives us **templates**, which are an easy way to make a bunch of controls look alike.
  Next, right-click on the ContentControl in the Document Outline window. Choose **Edit Template**, then choose **Create Empty...** from the menu. Name it `EnemyTemplate`. The IDE will add the template to the XAML.



  Your newly created template is currently selected in the IDE. Collapse the Document Outline window so it doesn't overlap the Toolbox. *Your template is **still invisible**, but you'll change that in the next step. If you accidentally click out of the control template, **you can always get back to it** by opening the Document Outline, right-clicking on the Content Control, and choosing Edit Template→Edit Current.*

> **NOTE**
>
> You're "flying blind" for this next bit — the designer won't display anything for the template until you add a control and set its height and width so it shows up. Don't worry; you can always undo and try again if something goes wrong.

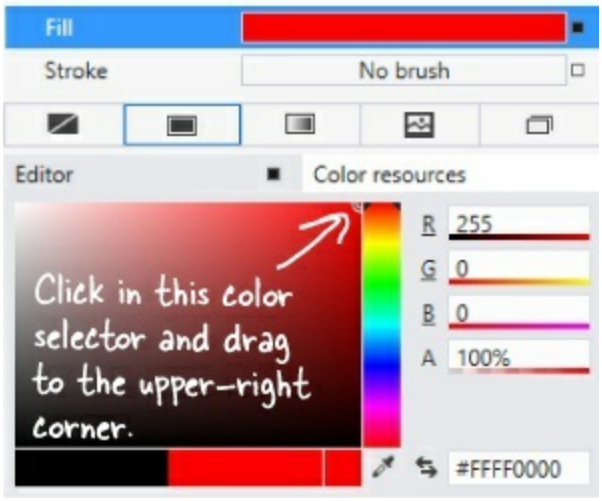
> **NOTE**
>
> You can also use the Document Outline window to select the grid if it gets deselected.

- **❹ Edit the enemy template.**
  Add a red circle to the template:

> **NOTE**
>
> Make sure you don't click anywhere else in the designer until you see the ellipse. That will keep the template selected.

  Double-click on Ellipse in the Toolbox to add an ellipse.
  Set the ellipse's `Height` and `Width` properties to `100`, which will cause the ellipse to be displayed in the cell.

Reset the `Margin`, `HorizontalAlignment`, and `VerticalAlignment` properties by clicking their squares and choosing Reset.
Go to the Brush section of the Properties window and click on ▭ to select a solid-color brush. Color your ellipse red by clicking in the color selector and dragging to the upper-right corner.



The XAML for your ContentControl now looks like this:

```
<ContentControl Content="ContentControl" Template="{DynamicResource EnemyTemplate}"
                VerticalAlignment="Center" HorizontalAlignment="Center"/>
```

Scroll around your window's XAML window and see if you can find where EnemyTemplate is defined. It should be right below the AppName resource.

- **❺ Use the Document Outline to modify the StackPanel, TextBlock, and Grid controls.**
Go back to the Document Outline (if you see EnemyTemplate (ContentControl Template) at the top of the Document Outline window, just click ▭ to get back to the Window outline). Select the StackPanel control, make sure its vertical and horizontal alignments are set to center, and clear the margins. Then do the same for the TextBlock, and use the Properties window to set the **Foreground** property to **white** using the color selector.

Click here and use the color selector to make the TextBlock white.

Finally, select the Grid, then open the Brush section of properties and click ▭ to give it a black Background.
You're almost done laying out the form! Flip the page for the last steps... →

- **❻ Add the human to the Canvas.**
You've got two options for adding the human. The first option is to follow the next three paragraphs. The second, quicker option is to just type the four lines of XAML into the IDE. It's your choice!
Select the Canvas control, and then open the All XAML Controls section of the toolbox and double-click on Ellipse to add an Ellipse control to the Canvas. Select the Canvas control again and double-click on Rectangle. The Rectangle will be added right on top of the Ellipse, so drag the Rectangle below it.
Hold down the Shift key and click on the Ellipse so both controls are selected. Right-click on the Ellipse, choose **Group Into**, and then **StackPanel**. Select the Ellipse, use the solid brush property

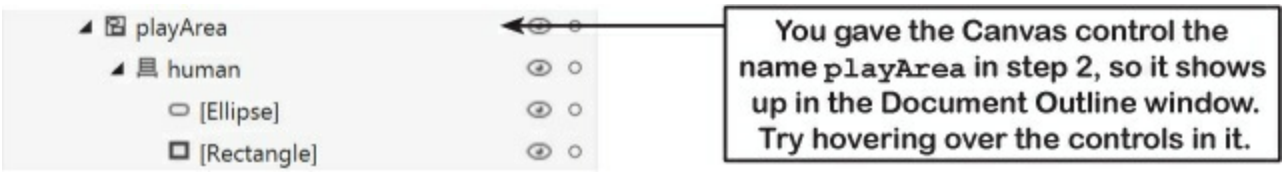
to change its color to white, and set its `Width` and `Height` properties to 10. Then select the Rectangle, make it white as well, and change its `Width` to 10 and its `Height` to 25.
Use the Document Outline window to select the Stack Panel (make sure you see Type StackPanel at the top of the Properties window). Reset its margins, **then click both ⛶ buttons to set the `Width` and `Height` to Auto**. Then use the Name box at the top of the window to set its name to `human`.
Here's the XAML you generated:

> **NOTE**
>
> If you choose to type this into the XAML window of the IDE, make sure you do it directly above the </Canvas> tag. That's how you indicate that the human is contained in the Canvas.

If you used the designer to create your human, make sure its source matches this XAML.

```
<StackPanel x:Name="human" Orientation="Vertical">
    <Ellipse Fill="White" Height="10" Width="10"/>
    <Rectangle Fill="White" Height="25" Width="10"/>
</StackPanel>
```

You might also see a Stroke property on the Ellipse and Rectangle set to `"Black"`. (If you don't see one, try adding it. What happens?)
Go back to the Document Outline window to see how your new controls appear:



If `human` isn't indented underneath `playArea`, click and drag `human` onto it.

- **❼ Add the Game Over text.**

When your player's game is over, the game will need to display a Game Over message. You'll do it by adding a TextBlock, setting its font, and giving it a name:

Select the Canvas, and then drag a TextBlock out of the toolbox and onto it.
Use the Name box in the Properties window to change its name to `gameOverText`.
Use the Text section of the Properties window to change the font to Arial, change the size to 100 px, and make it Bold and Italic.
Click on the TextBlock and drag it to the middle of the Canvas.
Edit the text so it says `Game Over`.

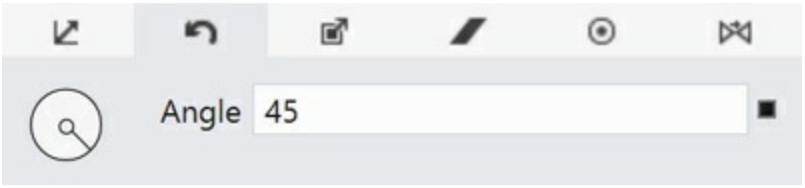
> **When you drag a control around a Canvas, its Left and Top properties are changed to set its position. If you change the Left and Top properties, you move the control.**

- **❽ Add the target portal that the player will drag the human onto.**

There's one last control to add to the Canvas: the target portal that your player will drag the human into. (It doesn't matter where in the Canvas you drag it.)
Select the Canvas control, and then drag a Rectangle control onto it. Use the ▭ button in the

Brushes section of the Properties window to give it a gradient. Set its `Height` and `Width` properties to **`50`**.

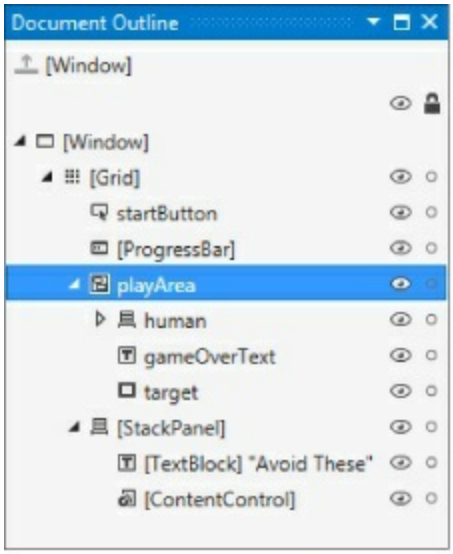
Turn your rectangle into a diamond by rotating it 45 degrees. Open the Transform section of the Properties window to rotate the Rectangle 45 degrees by clicking and setting the angle to **`45`**.



Finally, use the Name box in the Properties window to give it the name `target`.

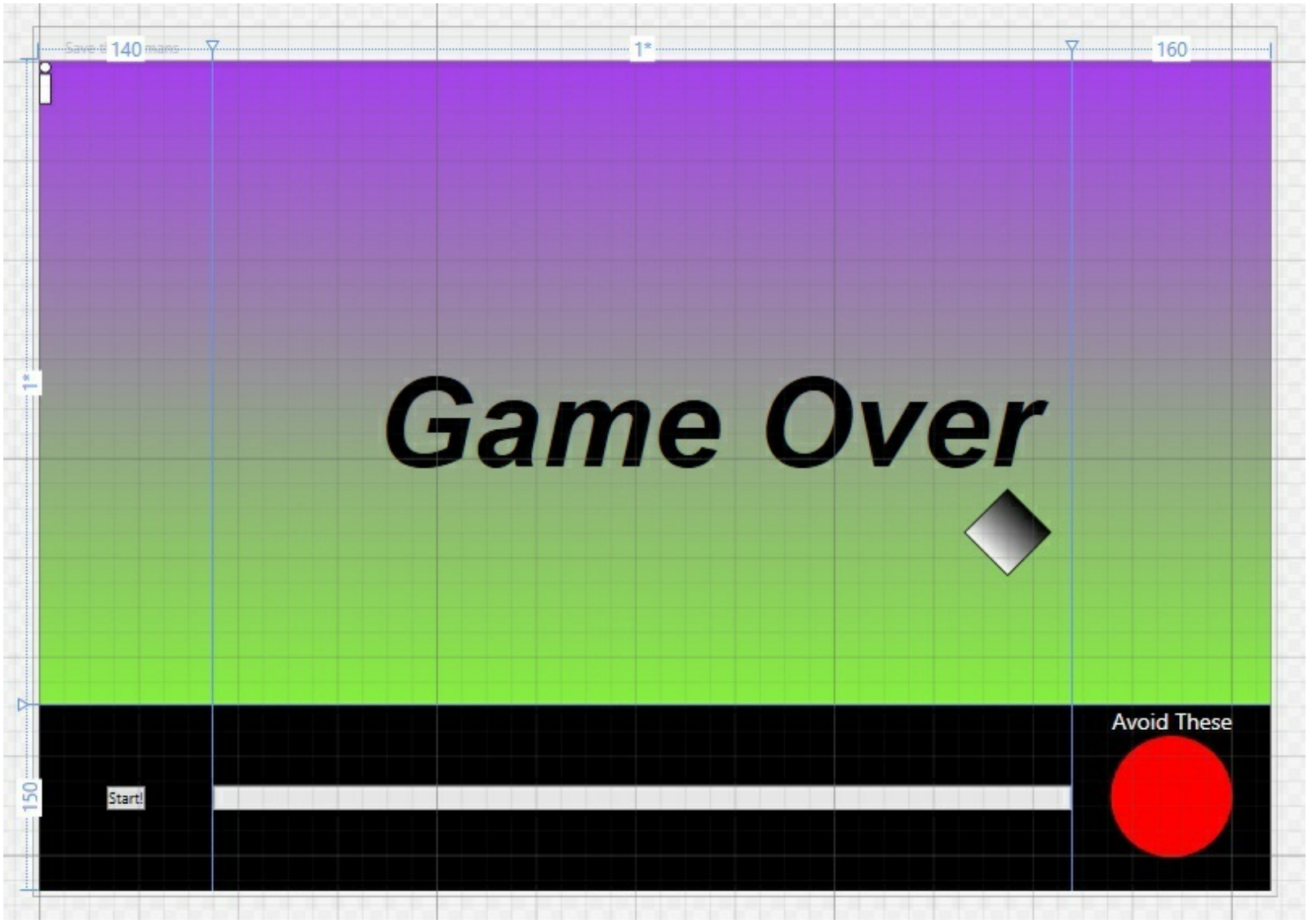
- **❾ Take a minute and double-check a few things.**

  Open the Document Outline window and make sure that the `human` StackPanel, `gameOverText` TextBlock, and `target` Rectangle are indented underneath the `playArea` Canvas control, which is indented under the second [Grid]. Select the `playArea` Canvas control and make sure its Height and Width are set to Auto. These are all things that **could cause bugs** in your game that will be difficult to track down. Your Document Outline window should look like this:



We collapsed human to make it obvious that it's indented underneath playArea, along with gameOverText and target. It's okay if the controls are in a different order (you can even drag them up an down!), as long as the indenting is correct— that's how you know which controls are inside other container controls.

Congratulations — you've finished building the window for your app!



WHO DOES WHAT?

Now that you've built a user interface, you should have a sense of what some of the controls do, and you've used a lot of different properties to customize them. See if you can work out which property does what, and where in the Properties window in the IDE you find it.

XAML property	Where to find it in the Properties window in the IDE	What it does
	At the top	Determines how tall the control should be
Content	▷ Brush	
Height	▷ Appearance	Sets the angle that the control is turned
Rotation	▷ Common	You use this in your C# code to manipulate a specific control
Fill	▷ Layout	The color of the control
x:Name	▷ Transform	Use this when you want to change text displayed inside your control

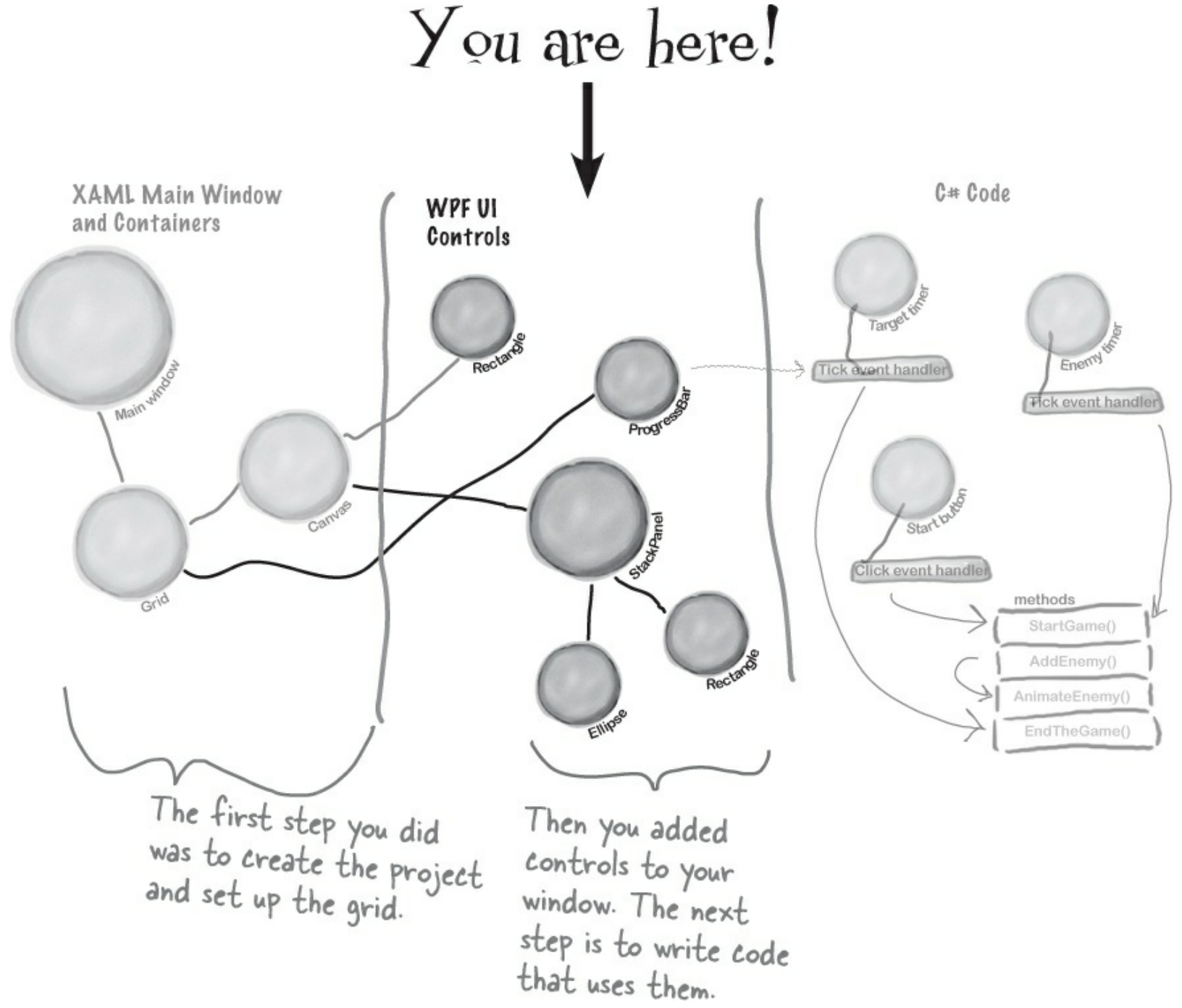
Solution in Note →

NOTE

Here's a hint: you can use the Search box in the Properties window to find properties — but some of these properties aren't on every type of control.

You've set the stage for the game

Your window is now all set for coding. You set up the grid that will serve as the basis of your window, and you added controls that will make up the elements of the game.



Visual Studio gave you useful tools for laying out your window, but all it really did was help you create XAML code. You're the one in charge!

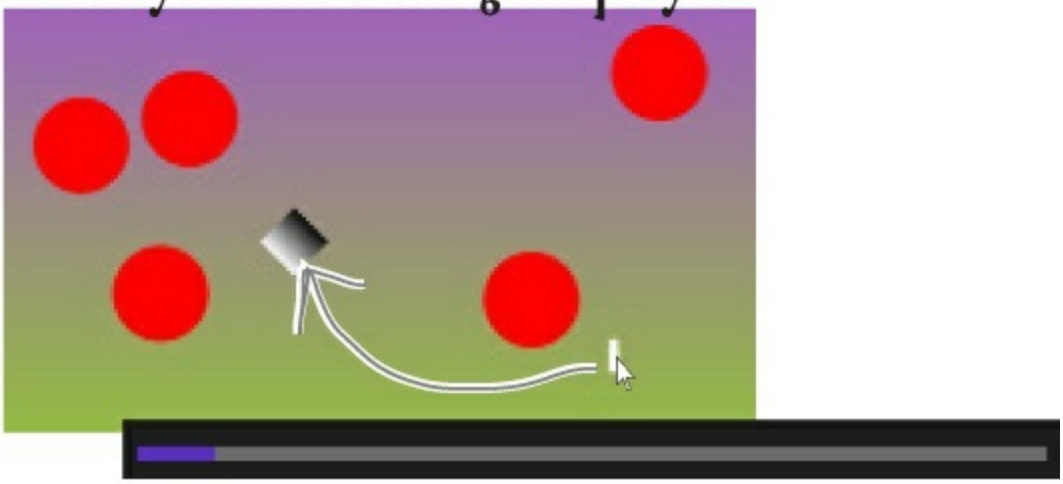
What you'll do next

Now comes the fun part: adding the code that makes your game work. You'll do it in three stages: first you'll animate your enemies, then you'll let your player interact with the game, and finally you'll add polish to make the game look better.

First you'll animate the enemies...

The first thing you'll do is add C# code that causes enemies to shoot out across the play area every time you click the Start button.

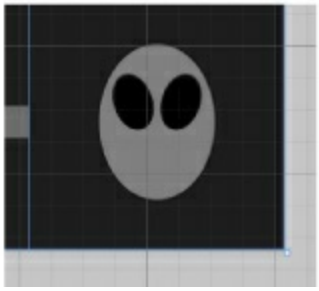
...then you'll add the gameplay...



To make the game work, you'll need the progress bar to count down, the human to move, and the game to end when the enemy gets him or time runs out.

You used a template to make the enemies look like red circles. Now you'll update the template to make them look like evil alien heads.

...and finally, you'll make it look good.



A lot of programmers build their code in small increments, making sure one piece works before moving on to the next one. That's how you'll build the rest of this program. You'll start by creating a method called `AddEnemy()` that adds an animated enemy to the Canvas control. First you'll hook it up to the Start button so you can fill your window up with bouncing enemies. That will lay the groundwork to build out the rest of the game.

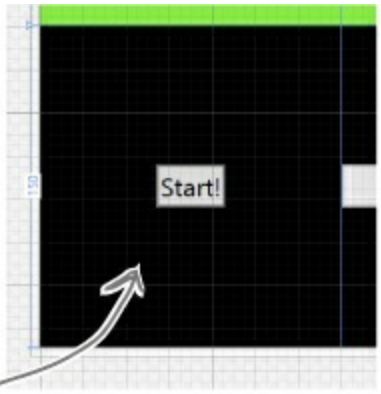
```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    AddEnemy();
}
```

The red squiggly line is the IDE telling you there's a problem, and the blue box is the IDE telling you that it might have a solution.

Add a method that does something

It's time to start writing some C# code, and the first thing you'll do is add a **method** — and the IDE can give you a great starting point by generating code.

When you're editing a window in the IDE, double-clicking on any of the toolbox controls causes the IDE to automatically add code to your project. Make sure you've got the window designer showing in the IDE, and then double-click on the Start button. The IDE will add code to your project that gets run anytime a user clicks on the button. You should see some code pop up that looks like this:



When you double-clicked the button control, the IDE created this method. It will run when a user clicks the "Start!" button in the running application.

```
private void startButton_Click(object sender, RoutedEventArgs e)
{

}
```

```
Click="startButton_Click"
```

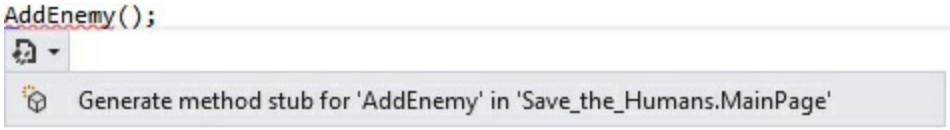
The IDE also added this to the XAML. See if you can find it. You'll learn more about what this is in Chapter 2.

Use the IDE to create your own method

Click between the { } brackets and type this, including the parentheses and semicolon:

Notice the red squiggly line underneath the text you just typed? That's the IDE telling you that something's wrong. If you click on the squiggly line, a blue box appears, which is the IDE's way of telling you that it might be able to help you fix the error.

Hover over the blue box and click the icon that pops up. You'll see a box asking you to generate a method stub. What do you think will happen if you click it? Go ahead and click it to find out!



THERE ARE NO DUMB QUESTIONS

Q: **Q: What's a method?**

A: **A:** A **method** is just a *named block of code*. We'll talk a lot more about methods in Chapter 2.

Q: **Q: And the IDE generated it for me?**

A: **A:** Yes...for now. A method is one of the basic building blocks of programs — you'll write a lot of them, and you'll get used to writing them by hand.

WATCH IT!

C# code must be added exactly as you see it here.

It's really easy to throw off your code. When you're adding C# code to your program, the capitalization has to be exactly right, and make sure you get all of the parentheses, commas, and semicolons. If you miss one, your program won't work!

Fill in the code for your method

It's time to make your program ***do something***, and you've got a good starting point. The IDE generated a **method stub** for you: the starting point for a method that you can fill in with code.

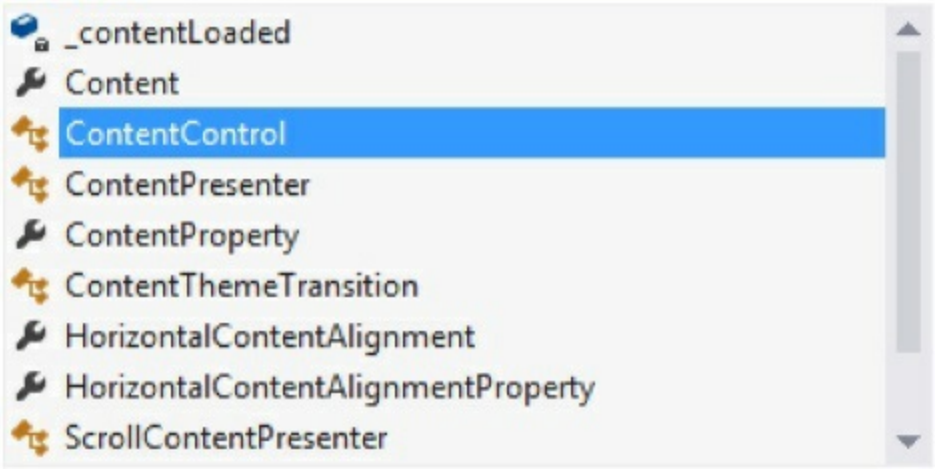
- ❶ Delete the contents of the method stub that the IDE generated for you.

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

Select this and delete it. You'll learn about exceptions in Chapter 12.

- ❷ Start adding code. Type the word "Content" into the method body. The IDE will pop up a window called an **IntelliSense Window** with suggestions. Choose `ContentControl` from the list.

```
private void AddEnemy()
{
    Content
}
```



- ❸ Finish adding the first line of code. You'll get another IntelliSense window after you type `new`.

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
}
```

This line creates a new ContentControl object. You'll learn about objects and the new keyword in Chapter 3, and reference variables like enemy in Chapter 4.

- ❹ Before you fill in the `AddEnemy()` method, you'll need to add a line of code near the top of the file. Find the line that says **`public partial class MainWindow : Window`** and add this line after the bracket (`{`):

```
/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow : Window
{
    Random random = new Random();
```

This is called a field. You'll learn more about how it works in Chapter 4.

- ❺ Finish adding the method. You'll see some squiggly red underlines. The ones under

`AnimateEnemy()` will go away when you generate its method stub.

Do you see a squiggly underline under playArea? Go back to the XAML editor and make sure you set the name of the Canvas control to playArea.

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
    enemy.Template = Resources["EnemyTemplate"] as ControlTemplate;
    AnimateEnemy(enemy, 0, playArea.ActualWidth - 100, "(Canvas.Left)");
    AnimateEnemy(enemy, random.Next((int)playArea.ActualHeight - 100),
        random.Next((int)playArea.ActualHeight - 100), "(Canvas.Top)");
    playArea.Children.Add(enemy);
}
```

This line adds your new enemy control to a collection called Children. You'll learn about collections in Chapter 8.

If you need to switch between the XAML and C# code, use the tabs at the top of the window.

- ❻ Use the blue box and the button to generate a method stub for `AnimateEnemy()`, just like you did for `AddEnemy()`. This time it added four **parameters** called `enemy`, `p1`, `p2`, and `p3`. Edit the top line of the method to change the last three parameters. Change the property `p1` to **`from`**, the property `p2` to **`to`**, and the property `p3` to **`propertyToAnimate`**. Then change any `int` types to **`double`**.

```
private void AnimateEnemy(ContentControl enemy, int p1, double p2, string p3)
{
    throw new NotImplementedException();
}
```

You'll learn about methods and parameters in Chapter 2.

```
private void AnimateEnemy(ContentControl enemy, double from, double to, string propertyToAnimate)
```

The IDE may generate the method stub with "int" types. Change them to "double". You'll learn about types in Chapter 4.

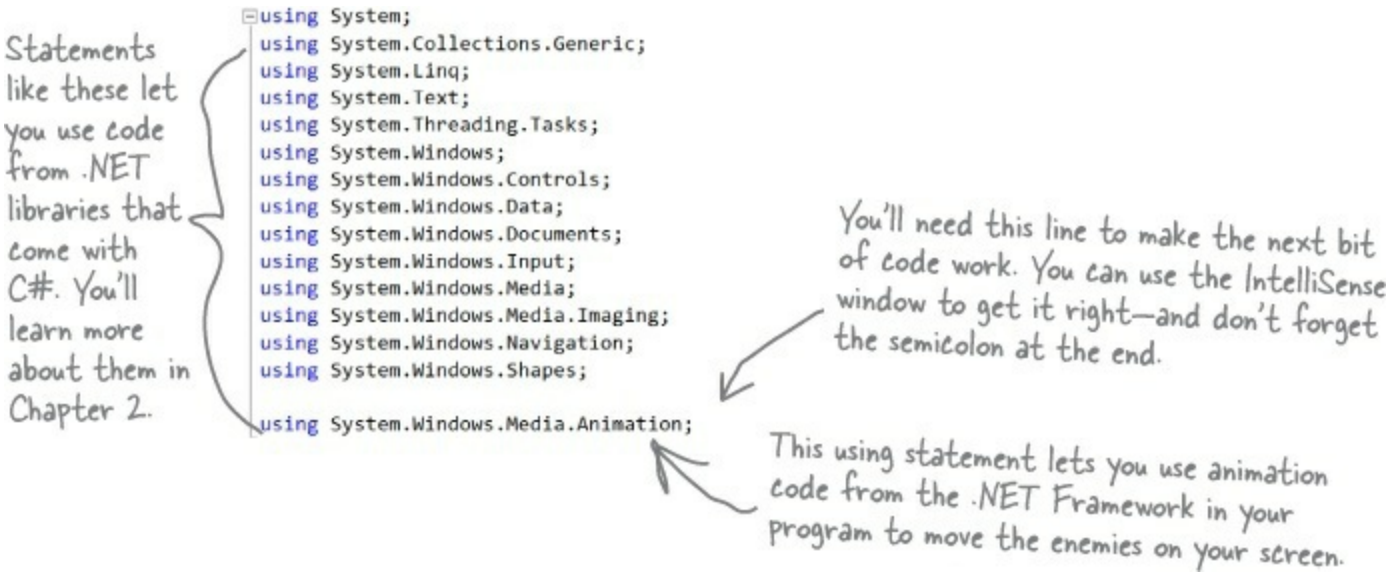
Flip the page to see your program run!

Finish the method and run your program

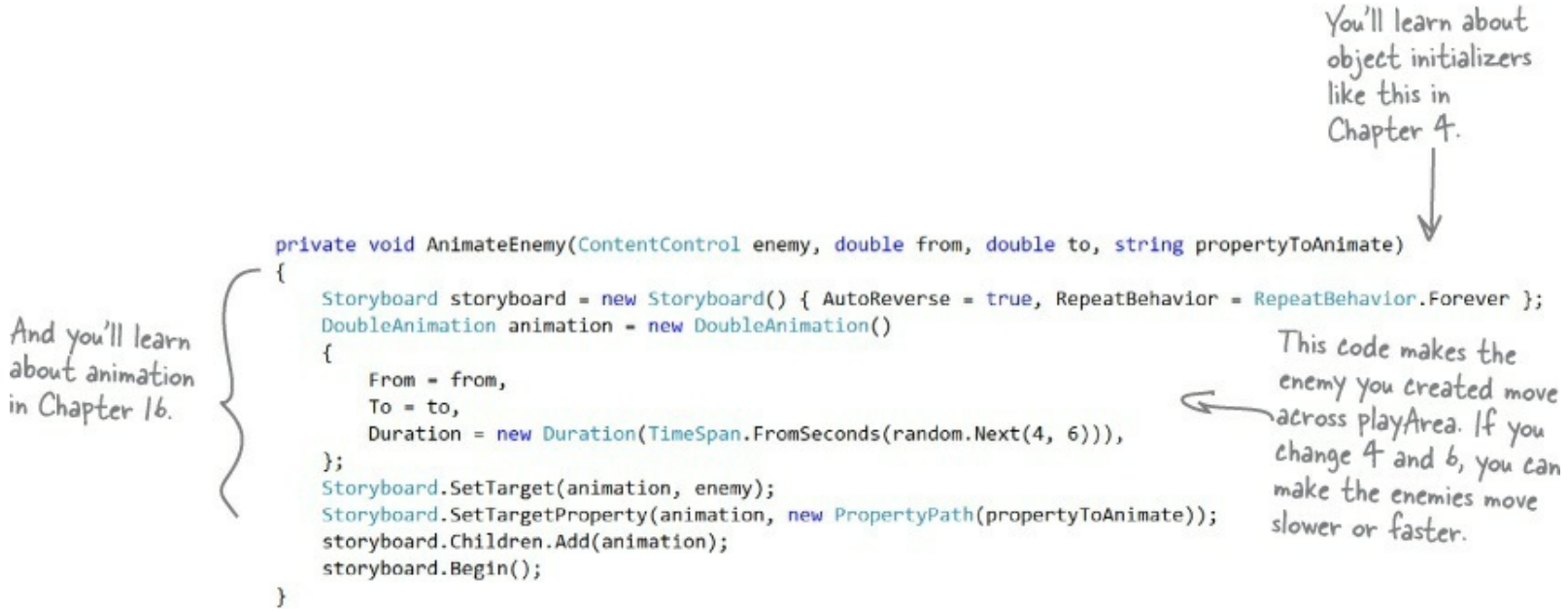
Your program is almost ready to run! All you need to do is finish your `AnimateEnemy()` method. Don't panic if things don't quite work yet. You may have missed a comma or some parentheses — when you're programming, you need to be really careful about those things!

- **❶ Add a using statement to the top of the file.**
  Scroll all the way to the top of the file. The IDE generated several lines that start with `using`. Add one more to the bottom of the list:

> **RELAX**
>
> **Still seeing red? The IDE helps you track down problems.**
>
> If you still have some of those red squiggly lines, don't worry! You probably just need to track down a typo or two. If you're still seeing squiggly red underlines, it just means you didn't type in some of the code correctly. We've tested this chapter with a lot of different people, and we didn't leave anything out. All the code you need to get your program working is in these pages.



- **❷ Add code that creates an enemy bouncing animation.**
  You generated the method stub for the `AnimateEnemy()` method on the previous page. Now you'll add its code. It makes an enemy start bouncing across the screen.

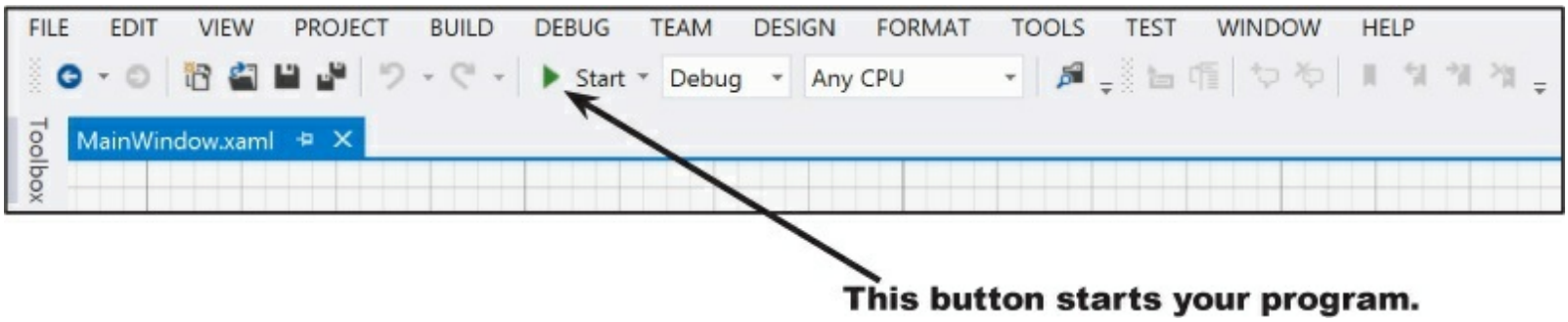


- **❸ Look over your code.**
  You shouldn't see any errors, and your Error List window should be empty. If not, double-click on the error in the Error List. The IDE will jump your cursor to the right place to help you track down the problem.

> **NOTE**
>
> If you can't see the Error List window, choose Error List from the View menu to show it. You'll learn more about using the error window and debugging your code in Chapter 2.

> **Here's a hint: if you move too many windows around your IDE, you can always reset by choosing Reset Window Layout from the Window menu.**

- **❹ Start your program.**
  Find the ▶ button at the top of the IDE. This starts your program running.



- **❺ Now your program is running!**
  When you start your program, the main window will be displayed. Click the "Start!" button a few times. Each time you click it, a circle is launched across your canvas.

You built something cool! And it didn't take long, just like we promised. But there's more to do to get it right.

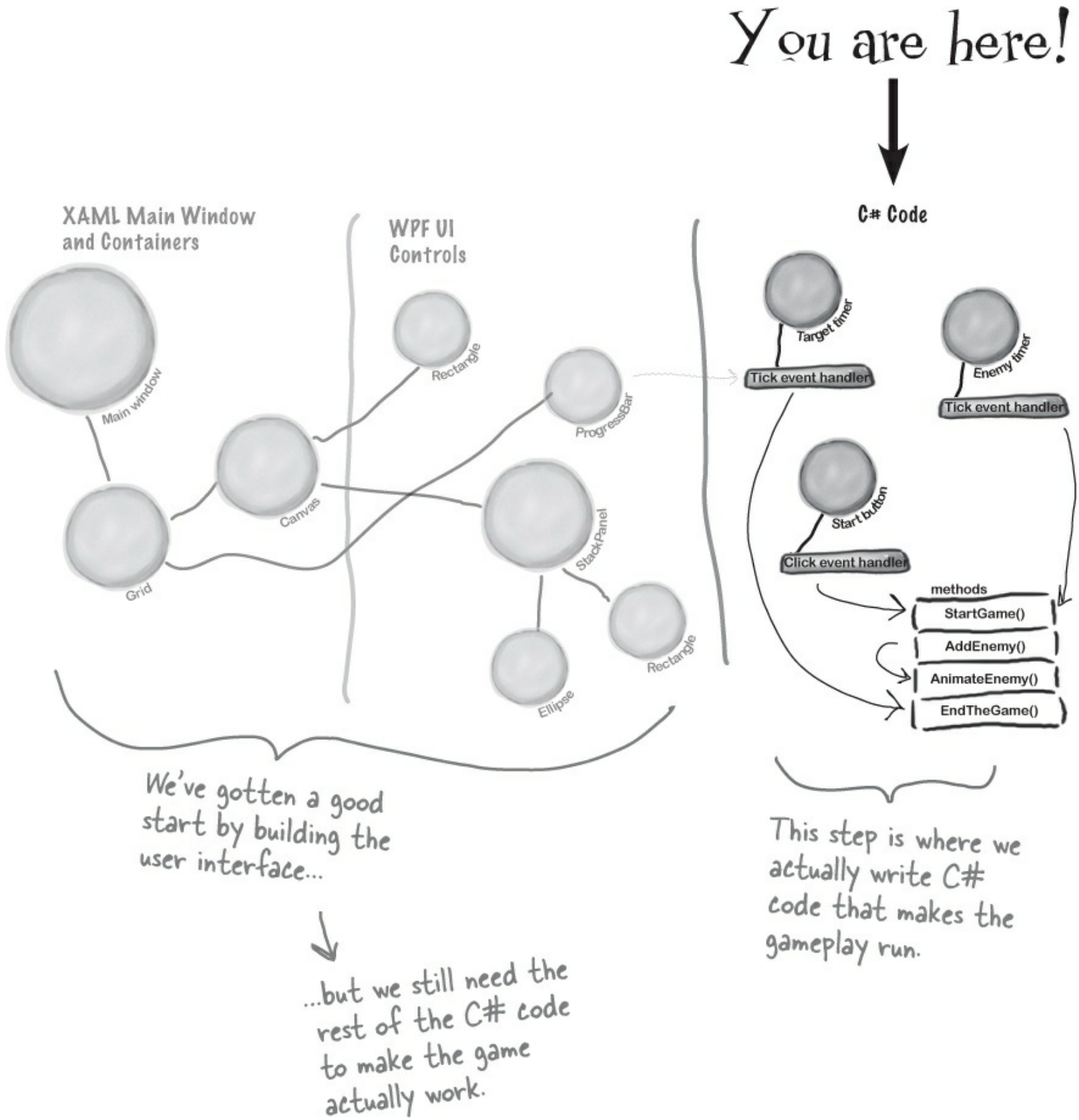
If the enemies aren't bouncing, or if they leave the play area, double-check the code. You may be missing parentheses or keywords.

- **❻ Stop your program.**

Press Alt-Tab to switch back to the IDE. The button in the toolbar has been replaced with to break, stop, and restart your program. Click the square to stop the program running.

Here's what you've done so far

Congratulations! You've built a program that actually does something. It's not quite a playable game, but it's definitely a start. Let's look back and see what you built.



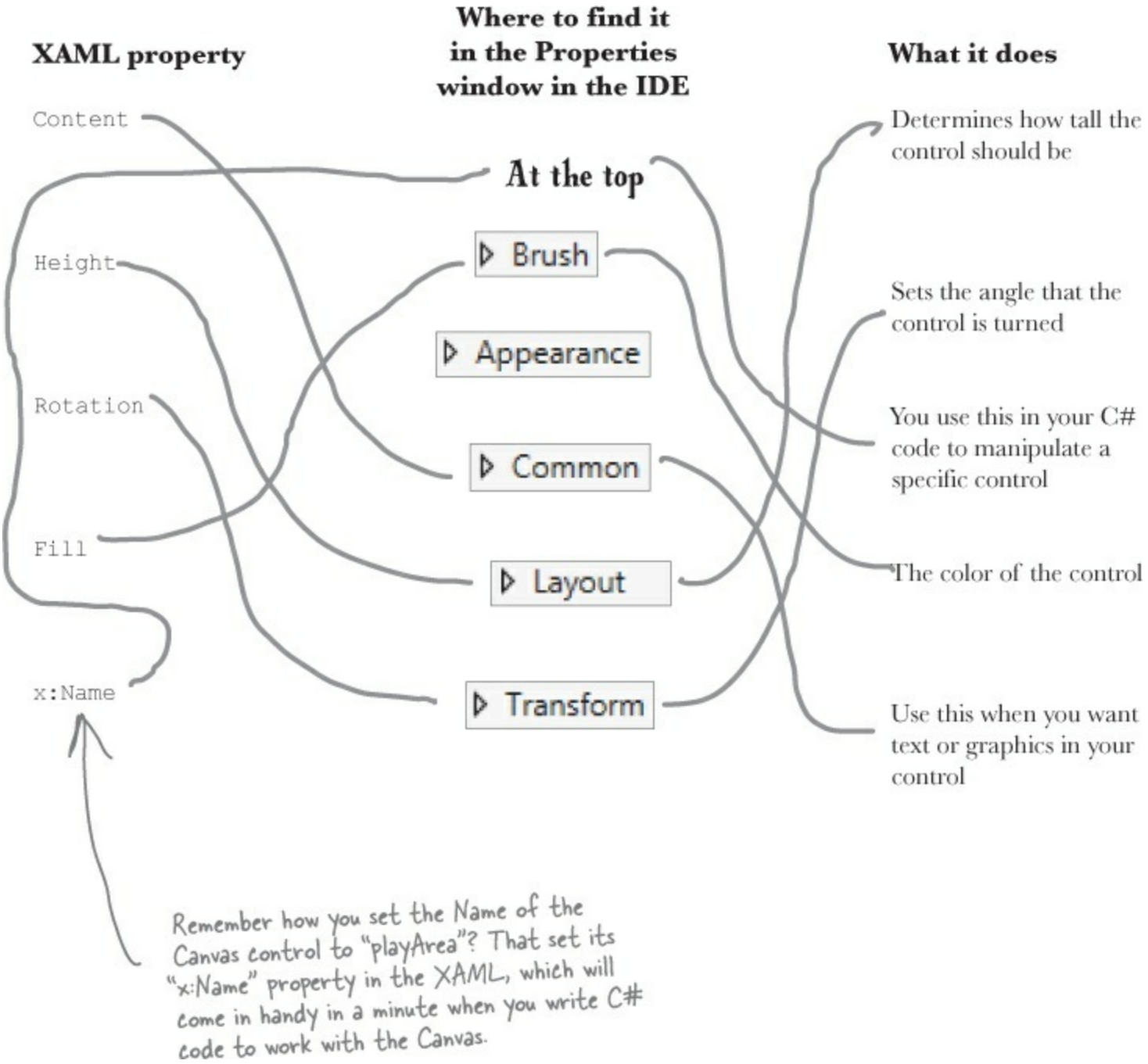
Visual Studio can generate code for you, but you need to know what you want to build <u>BEFORE</u> you start building it. It won't do that for you!

Here's the solution for the "Who Does What" exercise in Who Does What?. We'll give you the answers to the pencil-and-paper puzzles and exercises, but they won't always be on the next page.

WHO DOES WHAT?

solution

Now that you've built a user interface, you should have a sense of what some of the controls do, and you've used a lot of different properties to customize them. See if you can work out which property does what, and where in the Properties window in the IDE you find it.



Add timers to manage the gameplay

> **NOTE**
>
> The MainWindow.Xaml.cs file you've been editing contains the code for a class called MainWindow. You'll learn about classes in Chapter 3.

Let's build on that great start by adding working gameplay elements. This game adds more and more enemies, and the progress bar slowly fills up while the player drags the human to the target. You'll use **timers** to manage both of those things.

- **❶ ADD ANOTHER LINE TO THE TOP OF YOUR C# CODE.**
  You'll need to add one more `using` line right below the one you added a few pages ago:

```
using System.Windows.Media.Animation;
using System.Windows.Threading;
```

This using statement lets you use DispatcherTimers.

  Then go up to the top of the file where you added that `Random` line. Add three more lines:

```
public partial class MainWindow : Window
{
    Random random = new Random();
    DispatcherTimer enemyTimer = new DispatcherTimer();
    DispatcherTimer targetTimer = new DispatcherTimer();
    bool humanCaptured = false;
```

Add these three lines below the one you added before. These are fields, and you'll learn about them in Chapter 4.

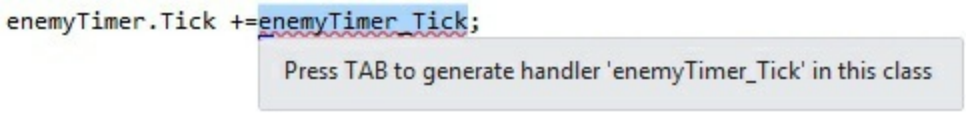
- **❷ ADD A METHOD FOR ONE OF YOUR TIMERS.**
  Find this code that the IDE generated:

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

  Put your cursor right after the semicolon, hit Enter two times, and type `enemyTimer.` (including the period). As soon as you type the dot, an IntelliSense window will pop up. Choose `Tick` from the IntelliSense window and type the following text. As soon as you enter += the IDE pops up a box:

Timers "tick" every time interval by calling methods over and over again. You'll use one timer to add enemies every few seconds, and the other to end the game when time expires.

Press the Tab key. The IDE will pop up another box:



Press Tab one more time. Here's the code the IDE generated for you:

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

    enemyTimer.Tick += enemyTimer_Tick;
}

void enemyTimer_Tick(object sender, EventArgs e)
{
    throw new NotImplementedException();
}
```

The IDE generated a method for you called an event handler. You'll learn about event handlers in Chapter 15.

- **❸ FINISH THE MAINWINDOW() METHOD.**

> **NOTE**
>
> It's normal to add parentheses () when writing about a method.

You'll add another `Tick` event handler for the other timer, and you'll add two more lines of code. Here's what your finished `MainWindow()` method and the two methods the IDE generated for you should look like:

> **BRAIN POWER**
>
> Right now your Start button adds bouncing enemies to the play area. What do you think you'll need to do to make it start the game instead?

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

    enemyTimer.Tick += enemyTimer_Tick;
    enemyTimer.Interval = TimeSpan.FromSeconds(2);

    targetTimer.Tick += targetTimer_Tick;
    targetTimer.Interval = TimeSpan.FromSeconds(.1);
}

void targetTimer_Tick(object sender, EventArgs e)
{
    throw new NotImplementedException();
}

void enemyTimer_Tick(object sender, EventArgs e)
{
    throw new NotImplementedException();
}
```

Try changing these numbers once your game is finished. How does that change the gameplay?

The IDE generated these lines as placeholders when you pressed Tab to add the Tick event handlers. You'll replace them with code that gets run every time the timers tick.

- **❹ ADD THE ENDTHEGAME() METHOD.**
  Go to the new `targetTimer_Tick()` method, delete the line that the IDE generated, and add the following code. Type `EndTheGame()` and generate a method stub for it, just like before:

```
void targetTimer_Tick(object sender, object e)
{
    progressBar.Value += 1;
    if (progressBar.Value >= progressBar.Maximum)
        EndTheGame();
}
```

Did the IDE keep trying to capitalize the P in progressBar? That's because there was no lowercase-P progressBar, and the closest match it could find was the type of the control.

Notice how `progressBar` has an error? That's OK. We did this on purpose (and we're not even sorry about it!) to show you what it looks like when you try to use a control that doesn't have a name, or has a typo in the name. Go back to the XAML code (it's in the other tab in the IDE), find the ProgressBar control that you added to the bottom row, and change its name to `progressBar`. Next, go back to the code window and generate a method stub for `EndTheGame()`, just like you did a few pages ago for `AddEnemy()`. Here's the code for the new method:

NOTE

If you closed the Designer tab that had the XAML code, double-click on MainWindow.xaml in the Solution Explorer window to bring it up.

This method ends the game by stopping the timers, making the Start button visible again, and adding the GAME OVER text to the play area.

```
private void EndTheGame()
{
    if (!playArea.Children.Contains(gameOverText))
    {
        enemyTimer.Stop();
        targetTimer.Stop();
        humanCaptured = false;
        startButton.Visibility = Visibility.Visible;
        playArea.Children.Add(gameOverText);
    }
}
```

If gameOverText comes up as an error, it means you didn't set the name of the "Game Over" TextBlock. Go back and do it now.

READY BAKE CODE

We're giving you a lot of code to type in.

By the end of the book, you'll know what all this code does — in fact, you'll be able to write code just like it on your own.

For now, your job is to make sure you enter each line accurately and to follow the instructions exactly. This will get you used to entering code and will help give you a feel for the ins and outs of the IDE..

Make the Start button work

Remember how you made the Start button fire circles into the Canvas? Now you'll fix it so it actually starts the game.

- **❶ Make the Start button start the game.**
  Find the code you added earlier to make the Start button add an enemy. Change it so it looks like this:

```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    StartGame();
}
```

When you change this line, you make the Start button start the game instead of just adding an enemy to the playArea Canvas.

- **❷ Add the StartGame() method.**
  Generate a method stub for the `StartGame()` method. Here's the code to fill into the stub method that the IDE added:

```
private void StartGame()
{
    human.IsHitTestVisible = true;
    humanCaptured = false;
    progressBar.Value = 0;
    startButton.Visibility = Visibility.Collapsed;
    playArea.Children.Clear();
    playArea.Children.Add(target);
    playArea.Children.Add(human);
    enemyTimer.Start();
    targetTimer.Start();
}
```

You'll learn about IsHitTestVisible in Chapter 15.

Did you forget to set the names of the target Rectangle or the human StackPanel? You can look a few pages back to make sure you set the right names for all the controls.

- **❸ Make the enemy timer add the enemy.**
  Find the `enemyTimer_Tick()` method that the IDE added for you and replace its contents with this:

```
void enemyTimer_Tick(object sender, object e)
{
    AddEnemy();
}
```

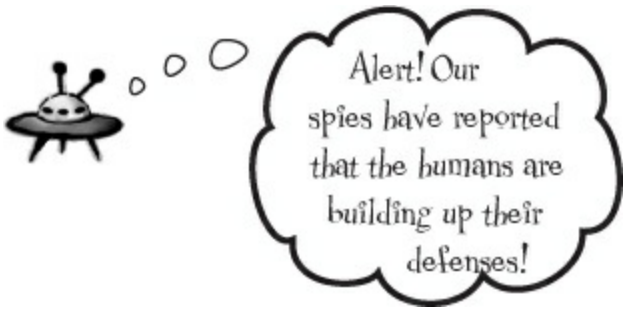
Are you seeing errors in the Error List window that don't make sense? One misplaced comma or semicolon can cause two, three, four, or more errors to show up. Don't waste your time trying to track down every typo! Just go to the Head First Labs web page — we made it really easy for you to copy and paste all the code in this program.
There's also a link to the Head First C# forum, which you can check for tips to get this game working!

NOTE

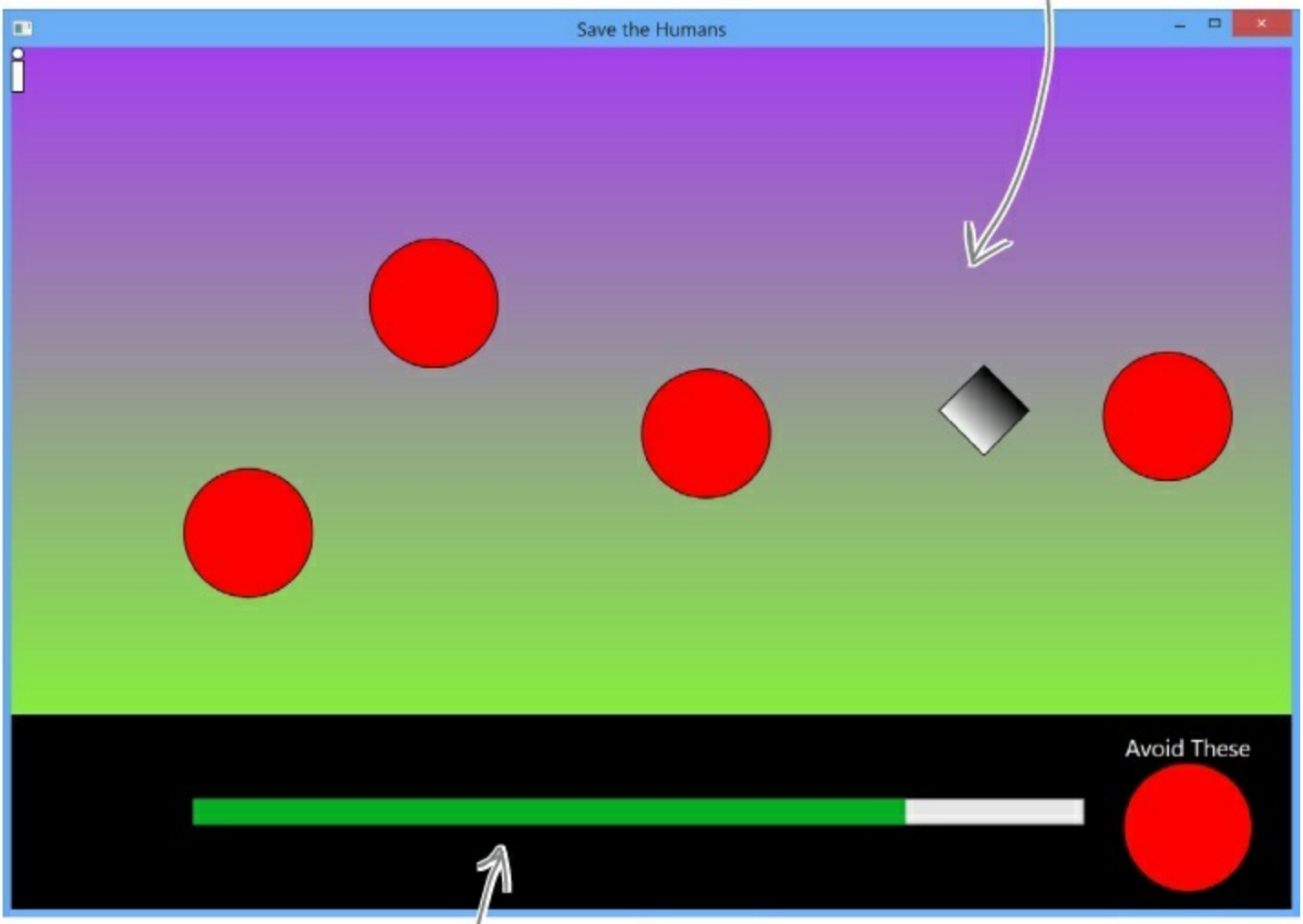
Once you're used to working with code, you'll be good at spotting those missing parentheses, semicolons, etc.

Run the program to see your progress

Your game is coming along. Run it again to see how it's shaping up.



The play area slowly starts to fill up with bouncing enemies.



When the progress bar at the bottom fills up, the game ends and the Game Over text is displayed.

The target timer should fill up slowly, and the enemies should appear every two seconds. If the timing is off, make sure you added all the lines to the MainWindow() method.

BRAIN POWER

What do you think you'll need to do to get the rest of your game working?

Flip the page to find out!

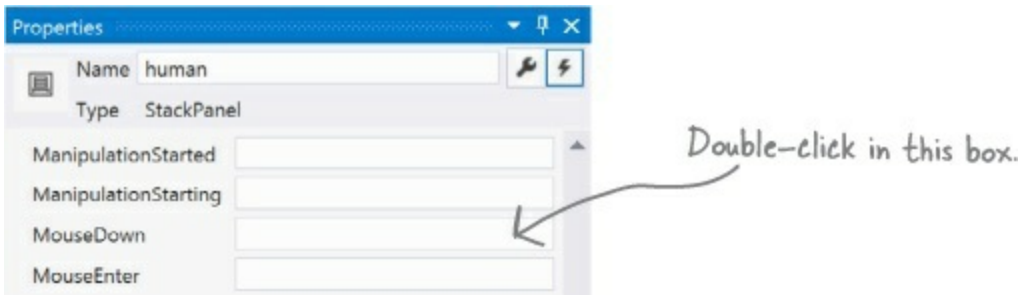
Add code to make your controls interact with the player

Make sure you switch back to the IDE and stop the app before you make more changes to the code.

You've got a human that the player needs to drag to the target, and a target that has to sense when the human's been dragged to it. It's time to add code to make those things work.

- ❶ Go to the XAML designer and use the Document Outline window to select `human` (remember, it's the StackPanel that contains a Circle and a Rectangle). Then go to the Properties window and press the ⚡ button to switch it to show event handlers. Find the MouseDown row and double-click in the empty box.

> **NOTE**
>
> You'll learn more about the event handlers in the Properties window in Chapter 4.



Now go back and check out what the IDE added to your XAML for the StackPanel:

```
<StackPanel x:Name="human" Orientation="Vertical" MouseDown="human_MouseDown">
```

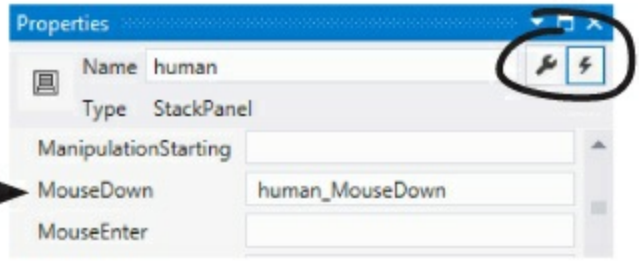
It also generated a method stub for you. Right-click on `human_MouseDown` in the XAML and choose "Navigate to Event Handler" to jump straight to the C# code:

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

}
```

- ❷ Fill in the C# code:

```
private void human_MouseDown(object sender, MouseButtonEventArgs e)
{
    if (enemyTimer.IsEnabled)
    {
        humanCaptured = true;
        human.IsHitTestVisible = false;
    }
}
```

You can use these buttons to switch between showing properties and event handlers in the Properties window.



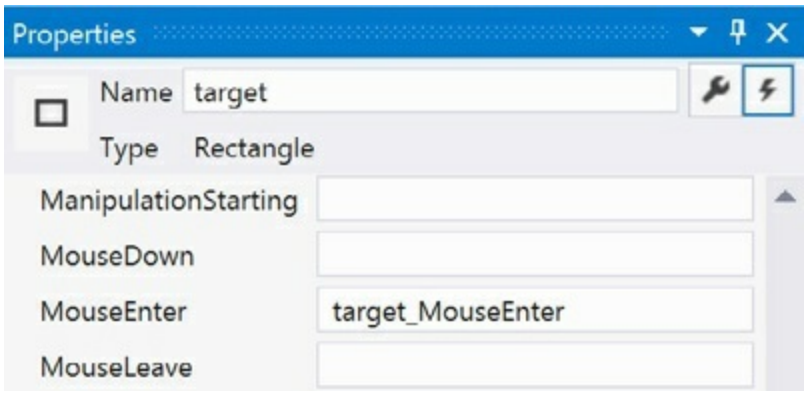
If you go back to the designer and click on the StackPanel again, you'll see that the IDE filled in the name of the new event handler method. You'll be adding more event handler methods the same way.

- ❸ Use the Document Outline window to select the Rectangle named `target`, and then use the event handlers view of the Properties window to add a `MouseEnter` event handler. Here's the code for the method:

> **NOTE**
>
> Make sure you add the right event handler! You added a MouseDown event handler to the human, but now you're adding a MouseEnter event handler to the target.

> **When the Properties window is in the mode where it displays event handlers, double-clicking on an empty event handler box causes the IDE to add a method stub for it.**

```
private void target_MouseEnter(object sender, MouseEventArgs e)
{
    if (targetTimer.IsEnabled && humanCaptured)
    {
        progressBar.Value = 0;
        Canvas.SetLeft(target, random.Next(100, (int)playArea.ActualWidth - 100));
        Canvas.SetTop(target, random.Next(100, (int)playArea.ActualHeight - 100));
        Canvas.SetLeft(human, random.Next(100, (int)playArea.ActualWidth - 100));
        Canvas.SetTop(human, random.Next(100, (int)playArea.ActualHeight - 100));
        humanCaptured = false;
        human.IsHitTestVisible = true;
    }
}
```



- ❹ Now you'll add two more event handlers, this time to the `playArea` Canvas control. You'll need to find the `[Grid]` in the Document Outline, select it, and set its name to `grid`. Then you can add these methods to handle the MouseMove and MouseLeave event handlers for the Canvas:

> **NOTE**
>
> You'll need to switch your Properties window back to show properties instead of event handlers.

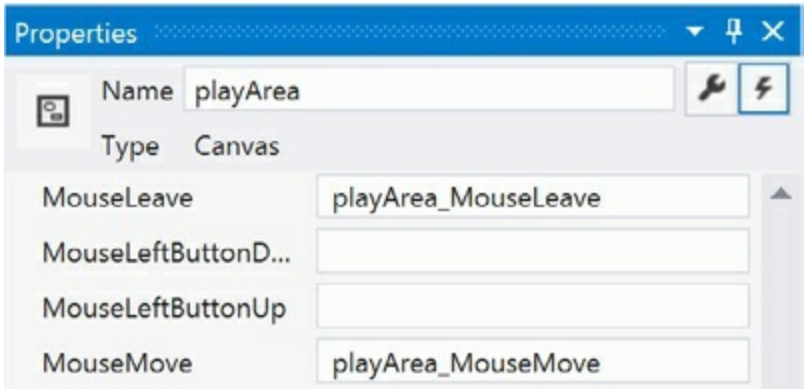
```
private void playArea_MouseMove(object sender, MouseEventArgs e)
{
    if (humanCaptured)
    {
        Point pointerPosition = e.GetPosition(null);
        Point relativePosition = grid.TransformToVisual(playArea).Transform(pointerPosition);
        if ((Math.Abs(relativePosition.X - Canvas.GetLeft(human)) > human.ActualWidth * 3)
            || (Math.Abs(relativePosition.Y - Canvas.GetTop(human)) > human.ActualHeight * 3))
        {
            humanCaptured = false;
            human.IsHitTestVisible = true;
        }
        else
        {
            Canvas.SetLeft(human, relativePosition.X - human.ActualWidth / 2);
            Canvas.SetTop(human, relativePosition.Y - human.ActualHeight / 2);
        }
    }
}

private void playArea_MouseLeave(object sender, MouseEventArgs e)
{
    if (humanCaptured)
        EndTheGame();
}
```

That's a lot of parentheses! Be really careful and get them right.

These two vertical bars are a logical operator. You'll learn about them in Chapter 2.

You can make the game more or less sensitive by changing these 3s to a lower or higher number.

> **NOTE**
>
> Make sure you put the right code in the correct event handler! Don't accidentally swap them.


Properties
Name playArea
Type Canvas
MouseLeave
playArea_MouseLeave
MouseLeftButtonD...
MouseLeftButtonUp
MouseMove
playArea_MouseMove

Dragging humans onto enemies ends the game

When the player drags the human into an enemy, the game should end. Let's add the code to do that. Go to your `AddEnemy()` method and add one more line of code to the end. Use the IntelliSense window to fill in `enemy.PointerEntered` from the list:

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
    enemy.Template = Resources["EnemyTemplate"] as ControlTemplate;
    AnimateEnemy(enemy, 0, playArea.ActualWidth - 100, "(Canvas.Left)");
    AnimateEnemy(enemy, random.Next((int)playArea.ActualHeight - 100),
        random.Next((int)playArea.ActualHeight - 100), "(Canvas.Top)");
    playArea.Children.Add(enemy);

    enemy.Mouse
}
```

MouseDoubleClick
MouseDown
MouseEnter
MouseLeave
MouseLeftButtonDown
MouseLeftButtonUp
MouseMove
MouseRightButtonDown
MouseRightButtonUp

MouseEventHandler UIElement.MouseEnter
Occurs when the mouse pointer enters the bounds of this element.

Here's the last line of your AddEnemy() method. Put your cursor at the end of the line and hit Enter to add the new line of code.

Start typing this line of code. As soon as you enter the dot, an IntelliSense window will pop up. Keep typing "Enter" to jump down to the right entry in the list.

Choose `MouseEnter` from the list. (If you choose the wrong one, don't worry — just backspace over it to delete everything past the dot. Then enter the dot again to bring up the IntelliSense window.)

Next, add an event handler, just like you did before. Type += and then press Tab:

```
enemy.MouseEnter +=
```

enemy_MouseEnter; (Press TAB to insert)

You'll learn all about how event handlers like this work in Chapter 15.

Then press Tab again to generate the stub for your event handler:

```
enemy.MouseEnter +=enemy_MouseEnter;
```

Press TAB to generate handler 'enemy_MouseEnter' in this class

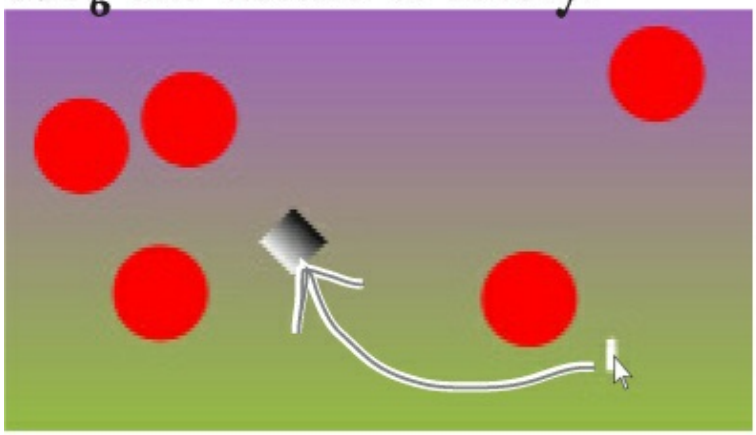
Now you can go to the new method that the IDE generated for you and fill in the code:

```
void enemy_MouseEnter(object sender, MouseEventArgs e)
{
    if (humanCaptured)
        EndTheGame();
}
```

Your game is now playable

Run your game — it's almost done! When you click the Start button, your play area is cleared of any enemies, and only the human and target remain. You have to get the human to the target before the progress bar fills up. Simple at first, but it gets harder as the screen fills with dangerous alien enemies!

Drag the human to safety!

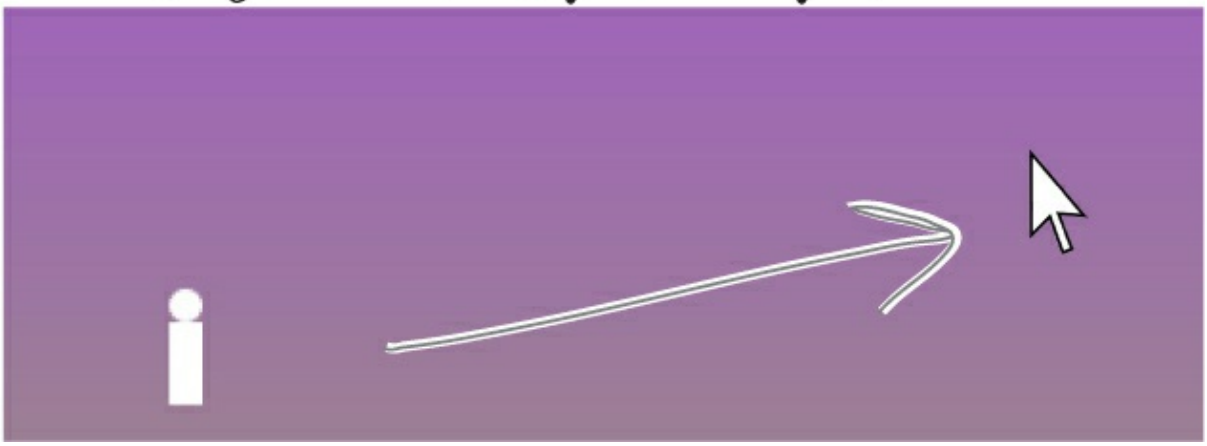


The aliens spend their time patrolling for moving humans, so the game ends only if you drag a human onto an enemy. Once you release the human, he's temporarily safe from aliens.

Look through the code and find where you set the IsHitTestVisible property on the human. When it's on, the human intercepts the PointerEntered event because the human's StackPanel control is sitting between the enemy and the pointer.

Get him to the target before time's up...

...but drag too fast, and you'll lose your human!



Make your enemies look like aliens

Red circles aren't exactly menacing. Luckily, you used a template. All you need to do is update it.

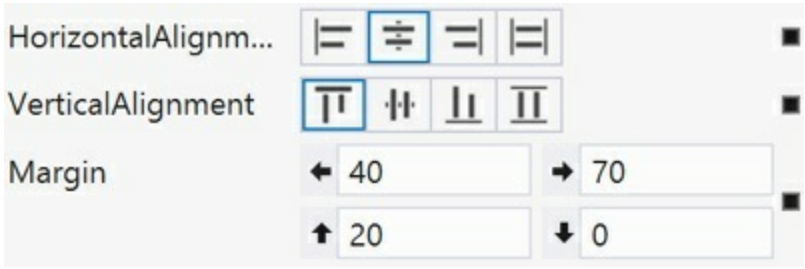
- ❶ Go to the Document Outline, right-click on the ContentControl, choose Edit Template, and then Edit Current to edit the template. You'll see the template in the XAML window. Edit the XAML code for the ellipse to set the width to `75` and the fill to `Gray`. Then add `Stroke="Black"` to add a black outline. Here's what it should look like (you can delete any additional properties that may have inadvertently been added while you worked on it):

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

> **WATCH IT!**
>
> **Seeing events instead of properties?**
>
> *You can toggle the Properties window between displaying properties or events for the selected control by clicking the wrench or lightning bolt icons.*

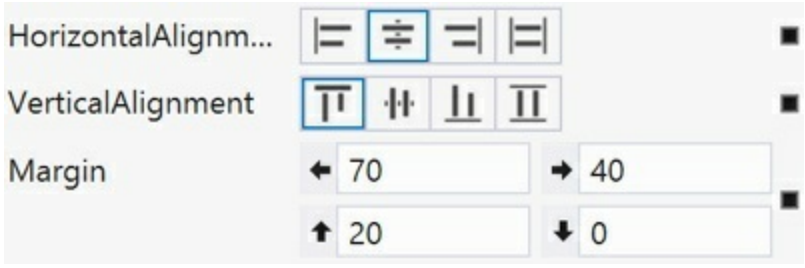
- ❷ Drag another Ellipse control out of the toolbox on top of the existing ellipse. Change its **Fill** to black, set its width to `25`, and its height to `35`. Set the alignment and margins like this:

> **NOTE**
>
> You can also "eyeball" it (excuse the pun) by using the mouse or arrow keys to drag the ellipse into place. Try using Copy and Paste in the Edit menu to copy the ellipse and paste another one on top of it.

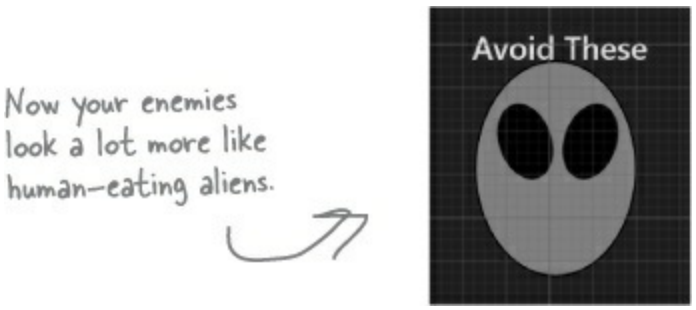


- ❸ Use the button in the Transforms section of the Properties window to add a Skew transform:

- ❹ Drag one more Ellipse control out of the toolbox on top of the existing ellipse. Change its fill to `Black`, set its width to `25`, and set its height to `35`. Set the alignment and margins like this: and add a skew like this:



and add a skew like this:



Here's the final XAML for the updated enemy ControlTemplate you created:

```
<ControlTemplate x:Key="EnemyTemplate" TargetType="{x:Type ContentControl}">
    <Grid>
        <Ellipse Fill="Gray" Stroke="Black" Height="100" Width="75"/>
        <Ellipse Fill="Black" Stroke="Black" Height="35"  Width="25"
                 VerticalAlignment="Top" HorizontalAlignment="Center"
                 Margin="40,20,70,0" RenderTransformOrigin="0.5,0.5">
            <Ellipse.RenderTransform>
                <TransformGroup>
                    <ScaleTransform/>
                    <SkewTransform AngleX="10"/>
                    <RotateTransform/>
                    <TranslateTransform/>
                </TransformGroup>
            </Ellipse.RenderTransform>
        </Ellipse>
        <Ellipse Fill="Black" Stroke="Black" Height="35"  Width="25"
                 VerticalAlignment="Top" HorizontalAlignment="Center"
                 Margin="70,20,40,0" RenderTransformOrigin="0.5,0.5">
            <Ellipse.RenderTransform>
                <TransformGroup>
                    <ScaleTransform/>
                    <SkewTransform AngleX="-10"/>
                    <RotateTransform/>
                    <TranslateTransform/>
                </TransformGroup>
            </Ellipse.RenderTransform>
        </Ellipse>
    </Grid>
</ControlTemplate>
```

NOTE

See if you can get creative and change the way the human, target, play area, and enemies look.

NOTE

And don't forget to step back and really appreciate what you built. Good job!

THERE'S JUST ONE MORE THING YOU NEED TO DO... PLAY YOUR GAME!

Chapter 2

Start diving into code with WPF projects.

The second chapter gets you started writing C# code, and most of the chapter is focused around building Windows Store apps.

We recommend that you do the following:

- Read Chapter 2 in the main part of the book through C# uses familiar math symbols.
- We provide a replacement for Use the debugger to see your variables change in this appendix. After that, you can read IDE Tip: , Loops perform an action over and over, and if/else statements make decisions in the book.
- Then there are replacements for Build an app from the ground up and Exercise Solution, where you build a program from scratch. You can follow the rest of the project in the book.
- The book will work just fine for you through Code Magnets.
- There's an exercise in Note, and its solution is in Exercise Solution. We provide replacements for those pages in this PDF.

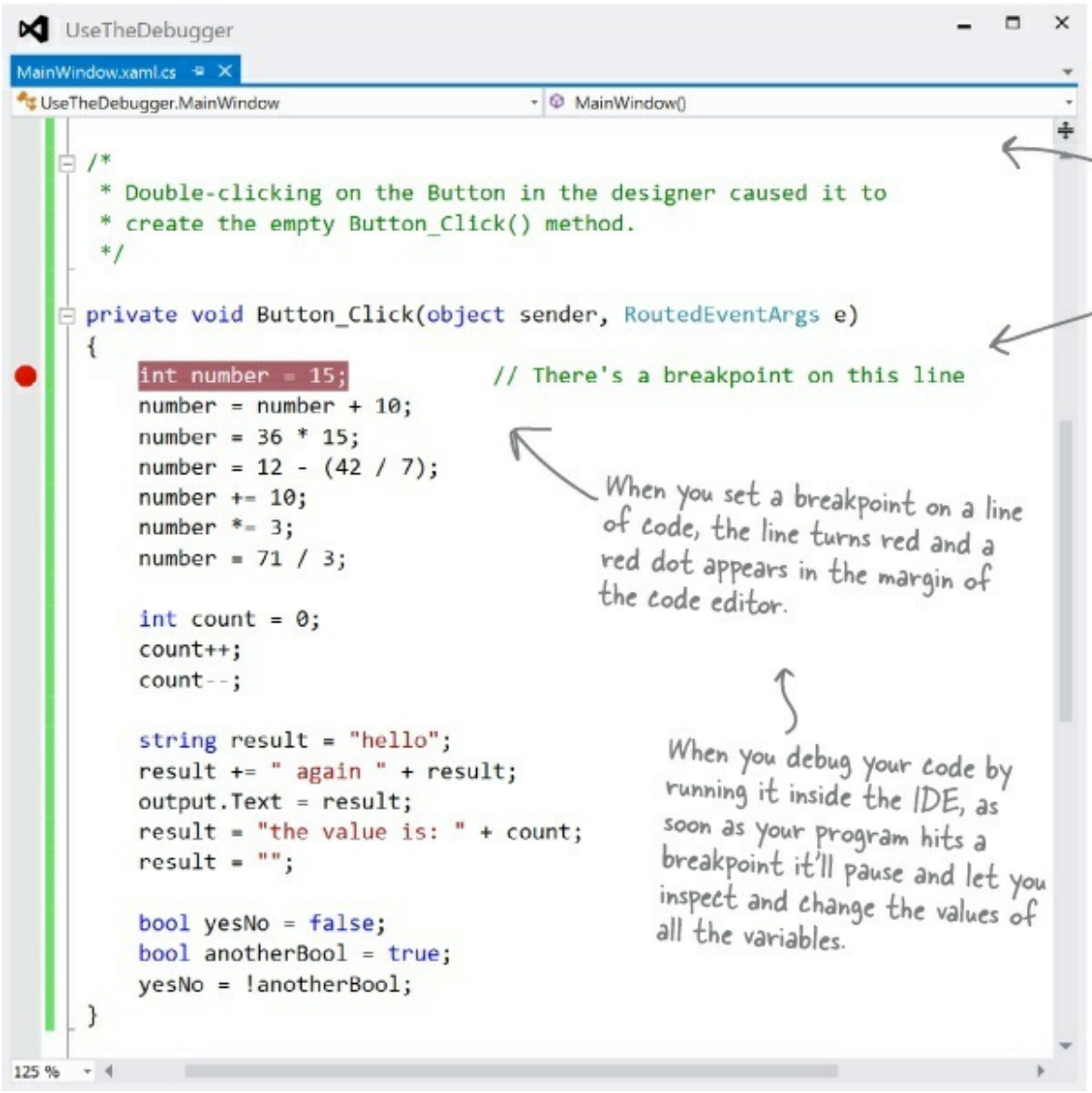
Once you finish that exercise, the chapter no longer requires any Windows Store apps or Windows 8. You'll be able to continue on in the book through Chapter 9, and you can do the first and second labs.

Use the debugger to see your variables change

The debugger is a great tool for understanding how your programs work. You can use it to see the code on the previous page in action.

DEBUG THIS!

- **❶ CREATE A NEW WPF APPLICATION PROJECT.**
  Drag a TextBlock onto your page and give it the name `output`. Then add a button and double-click it to add a method called `Button_Click()`. The IDE will automatically open that method in the code editor. Enter all the code on the previous page into the method.
- **❷ INSERT A BREAKPOINT ON THE FIRST LINE OF CODE.**
  Right-click on the first line of code (`int number = 15;`) and choose Insert Breakpoint from the Breakpoint menu. (You can also click on it and choose Debug→Toggle Breakpoint or press F9.)



Creating a new WPF Application project will tell the IDE to create a new project with a blank window. You might want to name it something like UseTheDebugger (to match the header of this page). You'll be building a whole lot of programs throughout the book, and you may want to go back to them later.
Flip back to IDE Tip: in the book and keep going!

NOTE

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Build an app from the ground up

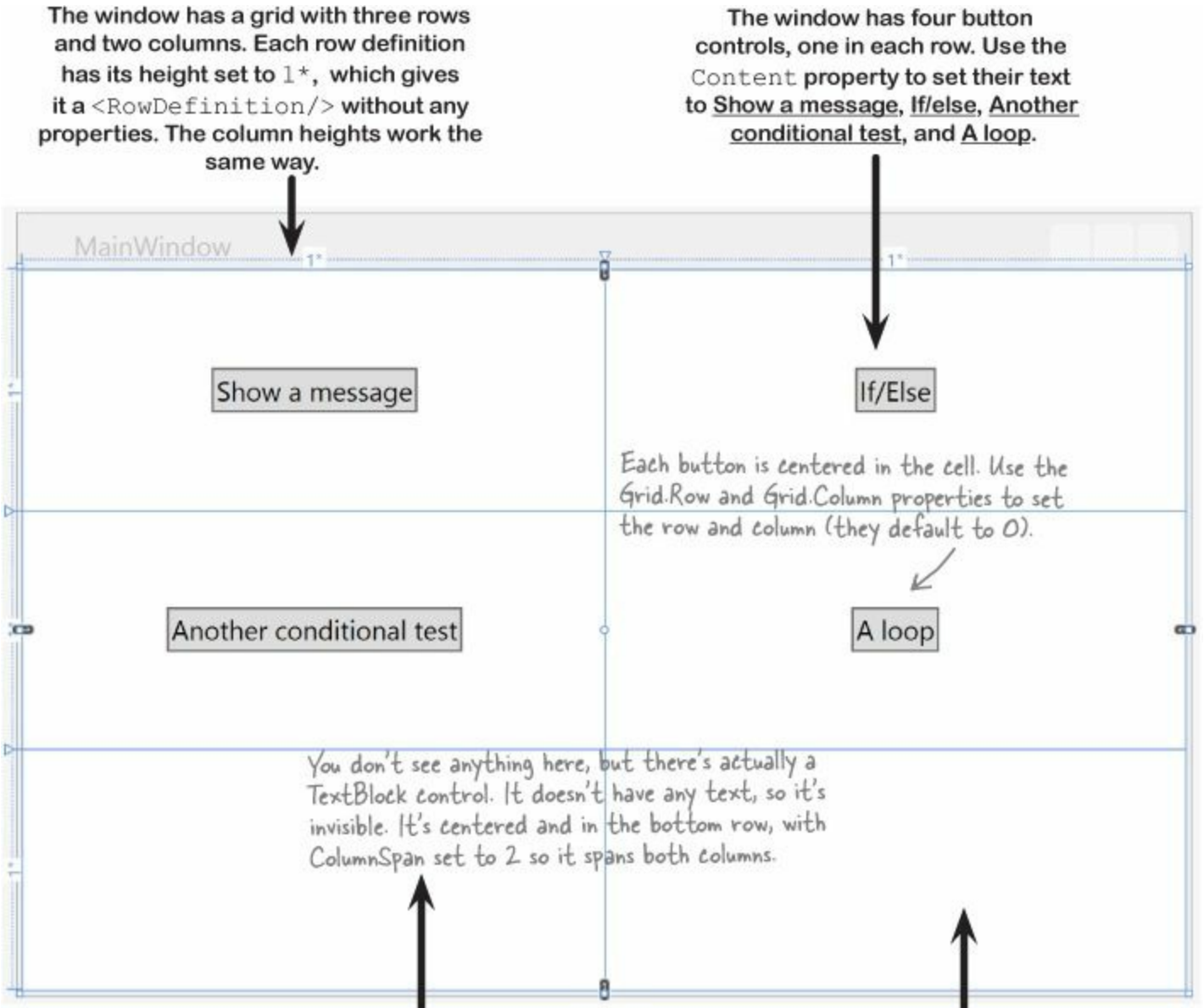
> **NOTE**
>
> Make sure you choose a sensible name for this project, because you'll refer back to it later in the book.

The real work of any program is in its statements. You've already seen how statements fit into a window. Now let's really dig into a program so you can understand every line of code. Start by **creating a new Visual C# WPF Application project**. Open the main window and use the IDE to modify it by adding three rows and two columns to the grid, and then adding four button controls and a TextBlock to the cells.

EXERCISE

Build this window

> **NOTE**
>
> When you see these sneakers, it means that it's time for you to come up with code on your own.

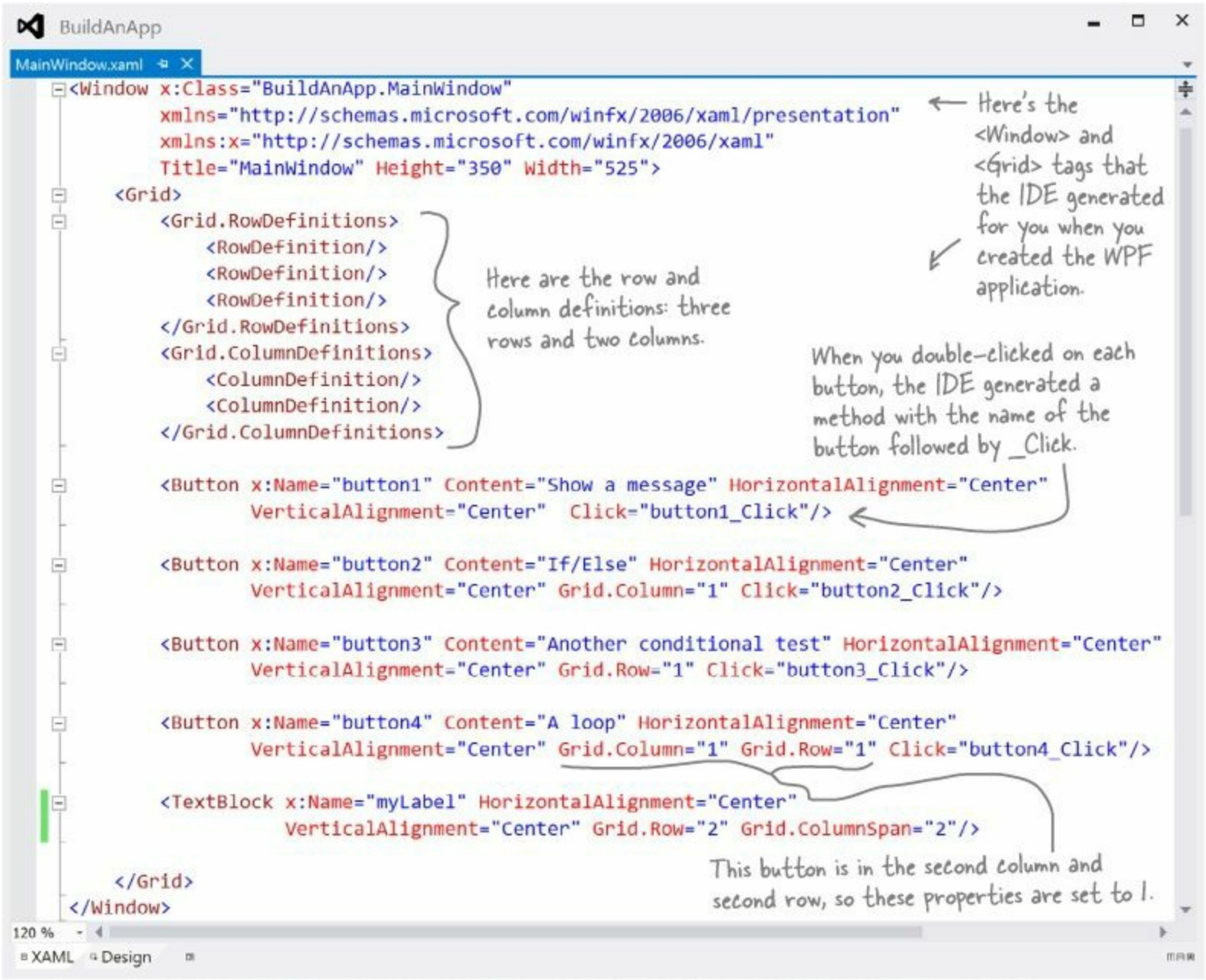


EXERCISE SOLUTION

NOTE

Here's our solution to the exercise. Does your solution look similar? Are the line breaks different, or the properties in a different order? If so, that's OK!

A lot of programmers don't use the IDE to create their XAML — they build it by hand. If we asked you to type in the XAML by hand instead of using the IDE, would you be able to do it?



Try removing the HorizontalAlignment or VerticalAlignment property from one of the buttons. It expands to fill the entire cell horizontally or vertically if the alignment isn't set.

BRAIN POWER

Why do you think the left column and top row are given the number 0, not 1? Why is it OK to leave out the `Grid.Row` and `Grid.Column` properties for the top-left cell?

EXERCISE

NOTE

We'll give you a lot of exercises like this throughout the book. We'll give you the answer in a couple of pages. If you get stuck, don't be afraid to peek at the answer — it's not cheating!

NOTE

You'll be creating a lot of applications throughout this book, and you'll need to give each one a different name. We recommend naming this one "PracticeUsingIfElse". It helps to put programs from a chapter in the same folder.

Time to get some practice using if/else statements. Can you build this program?

If you create two rows and set one row's height to 1* in the IDE, it seems to disappear because it's collapsed to a tiny size. Just set the other row to 1* and it'll show up again.

Build this window.
It's got a grid with two rows and two columns, it's 150 pixels tall and 450 pixels wide, and it's got the window title *Fun with if/else statements.*

Add a button and a checkbox.
You can find the checkbox control in the toolbox, just below the button control. Set the Button's name to `changeText` and the checkbox's name to `enableCheckbox`. Use the Edit Text right-click menu option to set the text for both controls (hit Escape to finish editing the text). Right-click on each control and choose Reset Layout→All, then make sure both of them have their VerticalAlignment and HorizontalAlignment set to `Center`.

Add a TextBlock.
It's almost identical to the one you added to the bottom of the window in the last project. This time, name it `labelToChange` and set its `Grid.Row` property to `"1"`.

Set the TextBlock to this message if the user clicks the button but the box IS NOT checked.

Text changing is disabled

Here's the conditional test to see if the checkbox is checked:

```
enableCheckbox.IsChecked == true
```

If that test is **NOT** true, then your program should execute two statements:

Hint: you'll put this code in the else block.

```
labelToChange.Text = "Text changing is disabled";
labelToChange.HorizontalAlignment = HorizontalAlignment.Center;
```

If the user clicks the button and the box IS checked, change the TextBlock so it either shows on the left-hand side or on the right-hand side.

If the label's `Text` property is currently equal to `"Right"` then the program should change the text to `"Left"` and set its `HorizontalAlignment` property to `HorizontalAlignment.Left`. Otherwise, set its text to `"Right"`and its `HorizontalAlignment` property to `HorizontalAlignment.Right`. This should cause the program to flip the label back and forth when the user presses the button — but only if the checkbox is checked.

EXERCISE SOLUTION

Time to get some practice using if/else statements. Can you build this program?

Here's the XAML code for the grid:

> **NOTE**
>
> We added line breaks as usual to make it easier to read on the window.

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

    <Button x:Name="changeText" Content="Change the label if checked"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Click="changeText_Click"/>

    <CheckBox x:Name="enableCheckbox" Content="Enable label changing"
        HorizontalAlignment="Center"  VerticalAlignment="Center"
        IsChecked="true" Grid.Column="1"/>

    <TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
            Text="Press the button to set my text"
            HorizontalAlignment="Center" VerticalAlignment="Center"
            Grid.ColumnSpan="2"/>

</Grid>
```

If you double-clicked the button in the designer before you set its name, it may have created a Click event handler method called Button_Click_1() instead of changeText_Click().

And here's the C# code for the button's event handler method:

```
private void changeText_Click(object sender, RoutedEventArgs e)
{
    if (enableCheckbox.IsChecked == true)
    {
        if (labelToChange.Text == "Right")
        {
            labelToChange.Text = "Left";
            labelToChange.HorizontalAlignment = HorizontalAlignment.Left;
        }
        else
        {
            labelToChange.Text = "Right";
            labelToChange.HorizontalAlignment = HorizontalAlignment.Right;
        }
    }
    else
    {
        labelToChange.Text = "Text changing is disabled";
        labelToChange.HorizontalAlignment = HorizontalAlignment.Center;
    }
}
```

You won't use XAML for the next part of the book.

The rest of Chapter 2 doesn't require Windows 8 and can be done with Visual Studio 2010, or using a Windows operating system as early as Windows 2003. You won't need to replace any pages in the book until you get to Chapter 10. That's because the next part of the book uses Windows Forms Application (or WinForms) projects. These C# projects use an older technology for building desktop apps. You'll use WinForms as a teaching and learning tool, just like you've been using the IDE to learn and explore C# and XAML.

NOTE

Have a look at Windows Desktop apps are easy to build, which explains why switching to WinForms is a good tool for getting C# concepts into your brain.

NOTE

This applies to WPF, too! Building these WinForms projects will help get core C# concepts into your brain faster, and that's the quickest route to learning WPF.

Sometimes older technologies make great learning tools.

If you want to build a desktop app, WPF is a superior tool for doing it. But if you want to learn C#, a simpler technology can make it easier to make concepts stick. And there's another important reason for using WinForms. When you see the same thing done in more than one way, you learn a lot from seeing what they have in common, and also what's different between them — like in Rebuild your app for Windows Desktop, when you rebuild the WPF you just built using WinForms. We'll get back to XAML in Chapter 10, and by that time you'll have laid down a solid foundation that will make it much easier for those WPF concepts to stick.

WATCH IT!

Some chapters use C# features introduced in .NET 4.0 that are not supported by Visual Studio 2008.

If you're using Visual Studio 2008, you may run into a few problems once you reach the end of Chapter 3. That's because the latest version of the .NET Framework available in 2008 was 3.5. And that's a problem, because the book uses features of C# that were only introduced in .NET 4.0. In Chapter 3 we'll teach you about ***object initializers****, and in Chapter 8 you'll learn about* ***collection initializers*** *and* ***covariance*** *— and if you're using Visual Studio 2008, the code for those examples* ***won't compile*** *because in 2008 those things hadn't been added to C# yet! If you absolutely can't install a newer version of Visual Studio, you'll still be able to do almost all the exercises, but you won't be able to use these features of C#.*

NOTE

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Chapter 10

You can port your WinForms apps to WPF.

If you've completed Chapter 3–Chapter 9 and finished all the exercises and labs so far, then you've **written a lot of code**. In this chapter, you'll revisit some of that code and use it as a springboard for learning WPF.

Here's how we recommend that you work through Chapter 10:

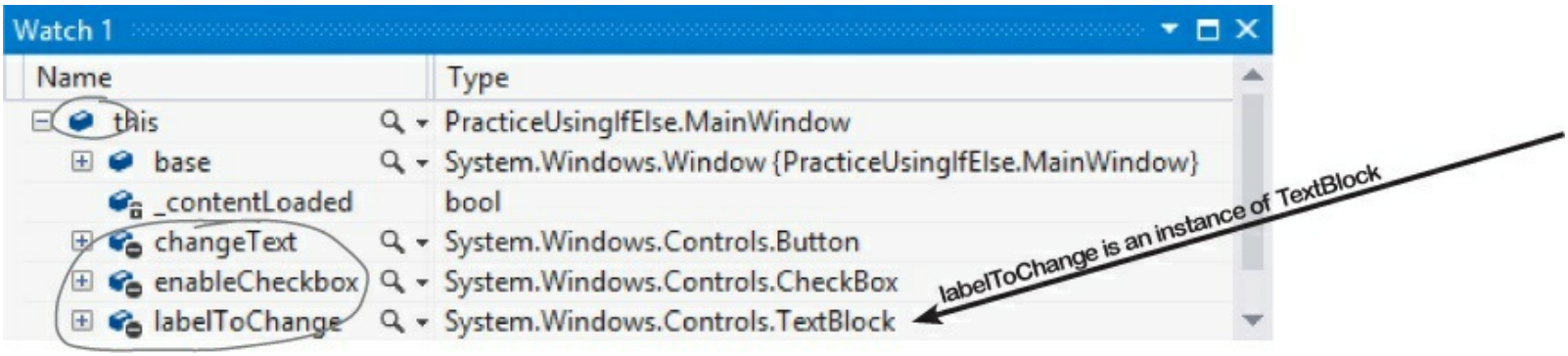
- We recommend that you follow the chapter in the main part of the book through Use the IDE to explore the object graph. This includes doing everything in Behind the scenes, the "Sharpen your Pencil" exercises, and the "Do this!" exploration project in Use the IDE to explore the object graph.
- This appendix has replacement pages for Windows Store apps use XAML to create UI objects–505, so use those instead.
- Use the grid system to lay out app pages **applies only to Windows Store projects**, so you can read it but it won't help you with WPF.
- After that, use Exercise Solution–Brain Power from this appendix.
- Finally, read XAML controls can contain text...and more and There are no Dumb Questions in the book. Once you've read them, you can replace the rest of the chapter (Use data binding to build Sloppy Joe a better menu–Exercise Solution) with pages in this appendix.

WPF applications use XAML to create UI objects

Do this!

When you use XAML to build the user interface for a WPF application, you're building out an object graph. And just like with WinForms, you can explore it with IDE's Watch window. **Open the "fun with if-else statements" program from Chapter 2**. Then open *MainWindow.xaml.cs*, place a breakpoint in the constructor on the call to `InitializeComponent()`, and **use the IDE to explore the app's UI objects**.

- ❶ Start debugging, then press F10 to step over the method. Open a Watch window using the Debug menu. Start by choosing **Debug→Windows→Watch→Watch 1**, and add a watch for `this`:

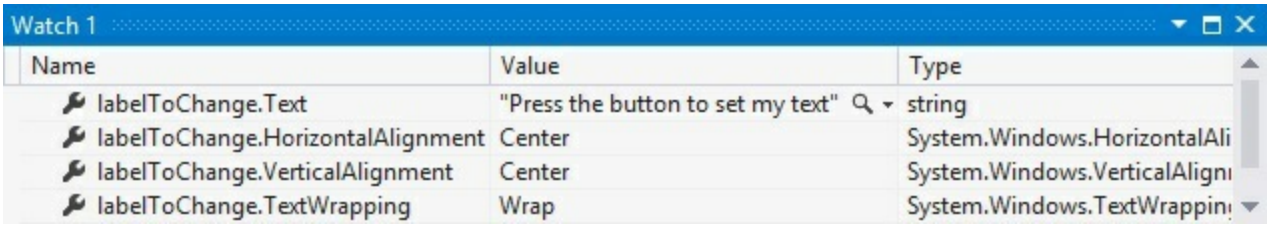


- ❷ Now have another look at the XAML that defines the page:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Button x:Name="changeText" Content="Change the label if checked"
            HorizontalAlignment="Center" Click="changeText_Click"/>

    <CheckBox x:Name="enableCheckbox" Content="Enable label changing"
              HorizontalAlignment="Center" IsChecked="true"
              Grid.Column="1"/>

    <TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
               Text="Press the button to set my text"
               HorizontalAlignment="Center" VerticalAlignment="Center"
               Grid.ColumnSpan="2"/>
</Grid>
```

The XAML that defines the controls on a page is turned into a Page object with fields and properties that contain references to UI controls.

- ❸ Add some of the `labelToChange` properties to the Watch window:



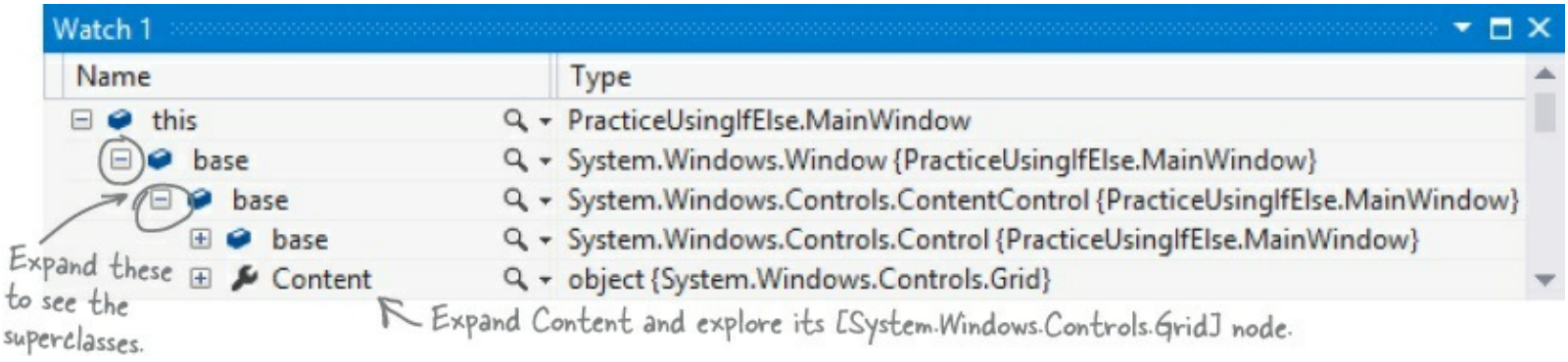
The app automatically sets the properties based on your XAML:

```
<TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
           Text="Press the button to set my text" →
           HorizontalAlignment="Center" VerticalAlignment="Center"
           Grid.ColumnSpan="2"/>
```

But try putting `labelToChange.Grid` or `labelToChange.ColumnSpan` into the Watch window. The control is a `Windows.UI.Controls.TextBlock` object, and that object doesn't have those properties. Can you guess what's going on with those XAML properties?

- ❹ Stop your program, open *MainWindow.xaml.cs*, and find the class declaration for `MainWindow`. Take a look at the declaration — it's a subclass of `Window`. Hover over `Window` so the IDE shows you its full class name:

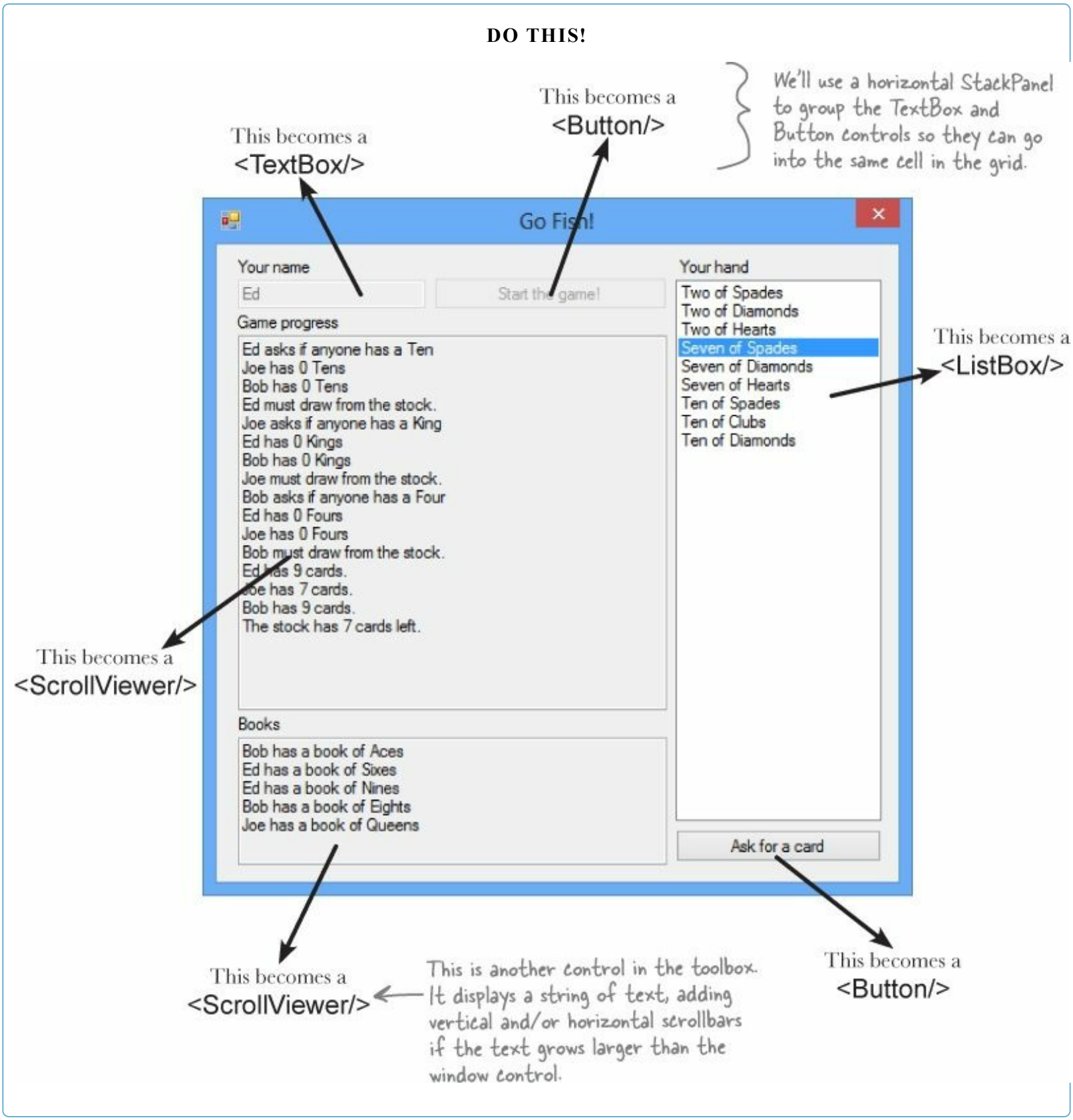
Now start your program again and press F10 to step over the call to `InitializeComponent()`. Go back to the Watch window and expand `this` >> `base` >> `base` to traverse back up the inheritance hierarchy.



Take a little time and explore the objects that your XAML generated. We'll dig into some of these objects later on in the book. For now, just poke around and get a sense of how many objects are behind your app.

Redesign the Go Fish! form as a WPF application

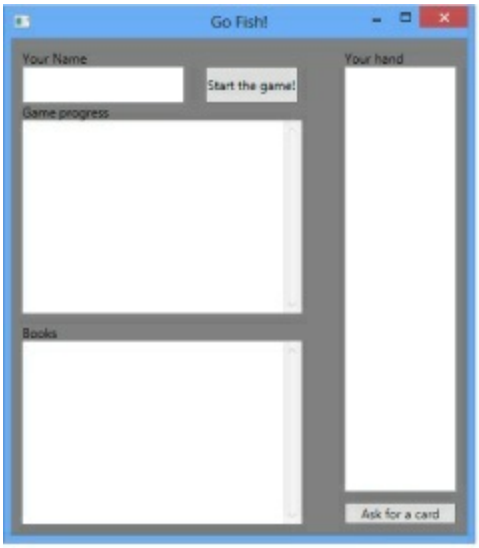
The Go Fish! game that you built in Chapter 8 would make a great WPF application. Open Visual Studio and **create a new WPF Application project** (just like you did for Save the Humans). Over the next few pages, you'll redesign it in XAML, with a main window that adjusts its content as it's resized. Instead of using Windows Forms controls on a form, you'll use WPF XAML controls.



Here's how those controls will look on the app's main window:

Most of the code to manage the gameplay will remain the same, but the UI code will change.

The controls will be contained in a grid, with rows and columns that expand or contract based on the size of the window. This will allow the game to shrink or grow if the user resizes the window:



The game will be playable no matter what the window dimensions are.

Page layout starts with controls

WPF apps and WinForms have one thing in common: they both rely on controls to lay out your page. The Go Fish! page has two buttons, a ListBox to show the hand, a TextBox for the user to enter the name, and four TextBlock labels. It also has two ScrollViewer controls with a white background to display the game progress and books.

The XAML for the main window starts with an opening `<Window>` tag. The title property sets the title of the window to "Go Fish!" Setting the Height and Width property changes the window size — and you'll see the size change in the designer as soon as you change those properties. Use the Background property to give it a gray background.

Here's the updated `<Window>` opening tag. We named our project `GoFish` — if you use a different name, the first line will have that name in its `x:Class` property.

The window title and starting width and height are set using properties in the <Window> tag.

```
<Window x:Class="GoFish.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Go Fish!" Height="500" Width="525" Background="Gray">
    <Grid Margin="10" >
```

We'll use a StackPanel to put the TextBox for the player's name and the Start button in one cell:

- ❶

```
<TextBlock Text="Your Name" />

<StackPanel Orientation="Horizontal" Grid.Row="1">
    <TextBox x:Name="playerName" FontSize="12" Width="150" />
```

- ❷

```
        <Button x:Name="startButton" Margin="5,0"
    Content="Start the game!"/>
</StackPanel>
```

This Margin property sets the left and right margins for the button to 5, and the top and bottom margins to 0. We could also have set it to 5,0,0,0 to set just the left margin and left the right margin zero.

Each label on the page ("Your name," "Game progress," etc.) is a TextBlock. Use the Margin property to add a 10-pixel margin above the label:

```
<TextBlock Text="Game progress" Grid.Row="2"
           Margin="0,10,0,0"/>
```

A ScrollViewer control displays the game progress, with scrollbars that appear if the text is too big for the window:

- ❸

```
<ScrollViewer Grid.Row="3" FontSize="12"
    Background="White" Foreground="Black" />
```

Here's another TextBlock and ScrollViewer to display the books. The default vertical and horizontal alignment for the ScrollViewer is `Stretch`, and that's going to be really useful. We'll set up the rows and columns so the ScrollViewer controls expand to fit any screen size.

- ❹

```
<TextBlock Text="Books"
       Margin="0,10,0,0" Grid.Row="4"/>

<ScrollViewer FontSize="12" Background="White" Foreground="Black"
          Grid.Row="5" Grid.RowSpan="2" />
```

We used a small 40-pixel column to add space, so the ListBox and Button controls need to go in the third column. The ListBox spans rows 2–6, so we gave it `Grid.Row="1"` and `Grid.RowSpan="5"` — this will also let the ListBox grow to fill the page.

- ❺

Remember, rows and columns start at zero, so a control in the third column has Grid.Column="2".

```
<TextBlock Text="Your hand" Grid.Row="0" Grid.Column="2" />

<ListBox x:Name="cards" Background="White" FontSize="12"
         Height="Auto" Margin="0,0,0,10"
         Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"/>
```

The "Ask for a card" button has its horizontal and vertical alignment set to `Stretch` so that it fills up the cell. The 20-pixel margin at the bottom of the ListBox adds a small gap.

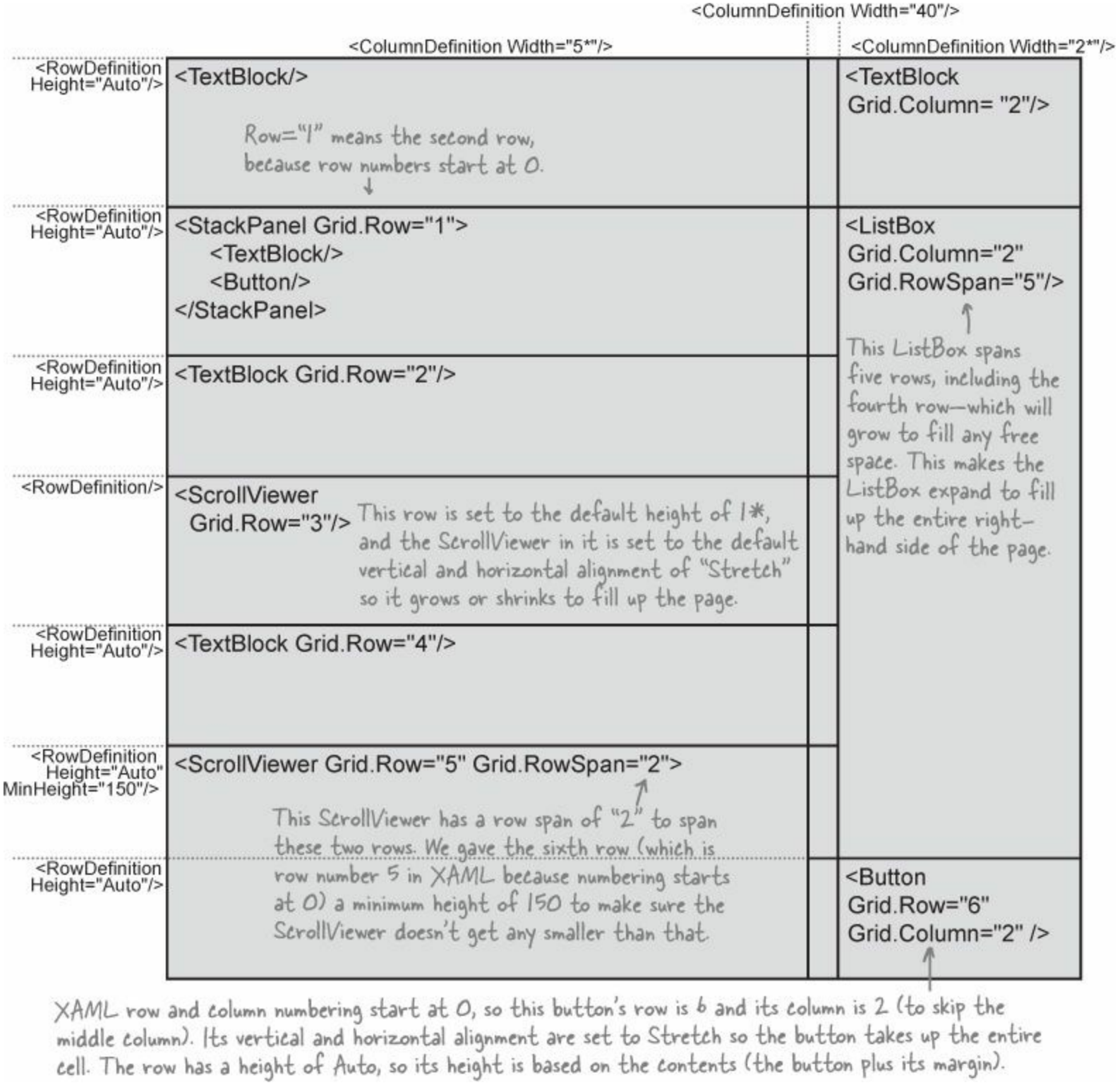
- ❻

```
<Button x:Name="askForACard" Content="Ask for a card"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
    Grid.Row="6" Grid.Column="2"/>
```

We'll finish this grid on the next page

Rows and columns can resize to match the page size

Grids are very effective tools for laying out windows because they help you design pages that can be displayed on many different devices. Heights or widths that end in * **adjust automatically** to different screen geometries. The Go Fish! window has three columns. The first and third have widths of 5* and 2*, so they will **grow or shrink proportionally** and always keep a 5:2 ratio. The second column has a fixed width of 40 pixels to keep them separated. Here's how the rows and columns for the window are laid out (including the controls that live inside them):



Here's how the row and column definitions make the window layout work:

```
  <Grid.ColumnDefinitions>
      <ColumnDefinition Width="5*"/>
      <ColumnDefinition Width="40"/>
      <ColumnDefinition Width="2*"/>
   </Grid.ColumnDefinitions>
   <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto" MinHeight="150" />
      <RowDefinition Height="Auto"/>
   </Grid.RowDefinitions>
```

The first column will always be 2.5 times as wide as the third (a 5:2 ratio), with a 40-pixel column to add space between them. The ScrollViewer and ListBox controls that display data have HorizontalAlignment set to "Stretch" to fill up the columns.

The fourth row has the default height of 1* to make it grow or shrink to fill up any space that isn't taken up by the other rows. The ListBox and first ScrollViewer span this row, so they will grow and shrink, too.

Almost all the row heights are set to `Auto`. There's only one row that will grow or shrink, and any control that spans this row will also grow or shrink.

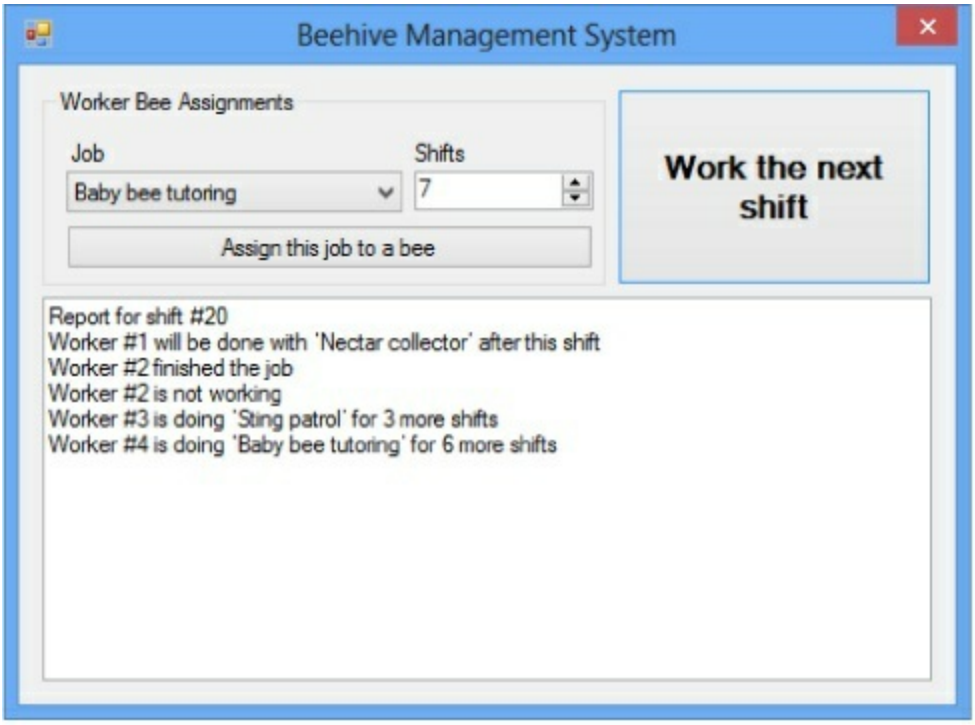
You can add the row and column definitions above or below the controls in the grid. We added them below this time.

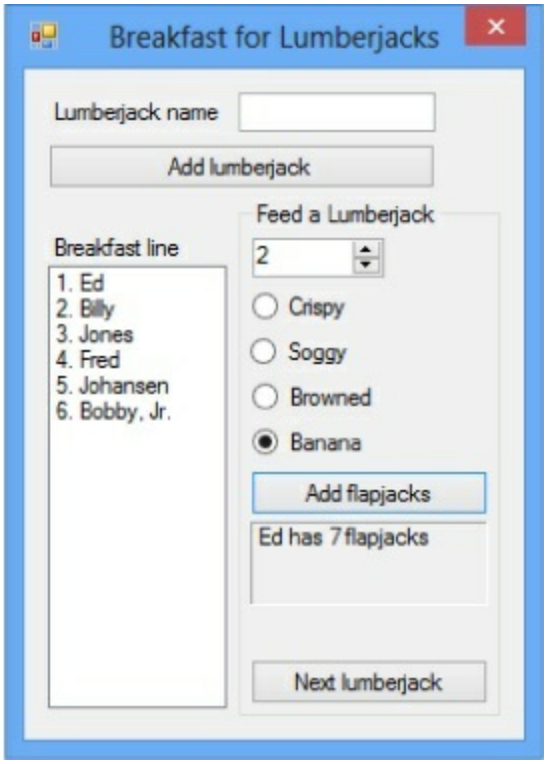
```
   </Grid>

</Window>
```

Here's the closing tag for the grid, followed by the closing tab for the window. You'll bring this all together at the end of the chapter when you finish porting the Go Fish! game to a WPF app.

EXERCISE

Use XAML to redesign each of these Windows desktop forms as WPF applications. Create a new WPF Application project for each of them, and modify each page by updating or replacing the grid and adding controls. You don't need to get them working. Just create the XAML so they match the screenshots.





NOTE

Use a Border control to draw a border around ScrollViewers.

If you look in the Properties window or look at the IntelliSense window, you'll see that the ScrollViewer control has BorderBrush and BorderThickness properties. This is a little misleading, because these properties don't actually do anything. ScrollViewer is a subclass of ContentControl, and it inherits those properties from ContentControl but doesn't actually do anything with them.

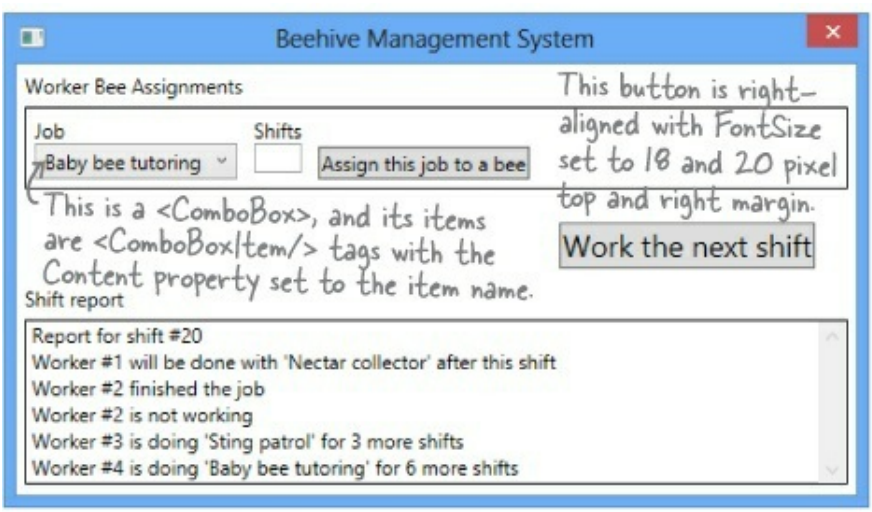
Luckily, there's an easy way to draw a border around a ScrollViewer, or any other control, by using a Border control. Here's XAML code that you can use in the Breakfast for Lumberjacks window:

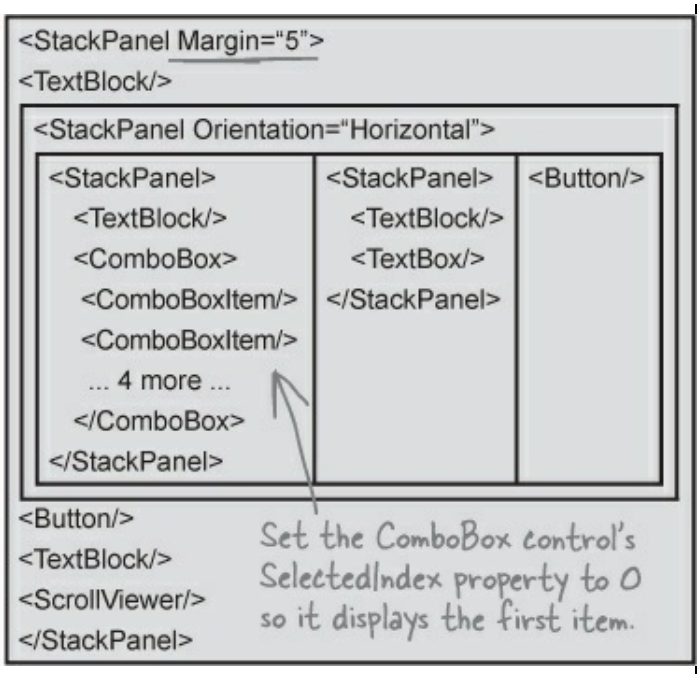
```
<Border Grid.Row="6" BorderThickness="1"
        Margin="0,5,0,0" BorderBrush="Gray">
    <ScrollViewer Content="Ed has 7 flapjacks"
                  BorderThickness="2" BorderBrush="White" MinHeight="50"/>
</Border>
```

Use the BorderThickness and BorderBrush properties to set the thickness and color of the border. You can also add a background, round the corners, and make other visual changes.

The Border control can contain one other control. If you want to put more than one control inside it, use a StackPanel, Grid, Canvas, or other container.

Use StackPanels to design this window. Its height is set to `300`, its width is `525`, and its ResizeMode property is set to `NoResize`. It uses two `<Border>` controls, one to draw a border around the top StackPanel and one to draw a border around the ScrollViewer.

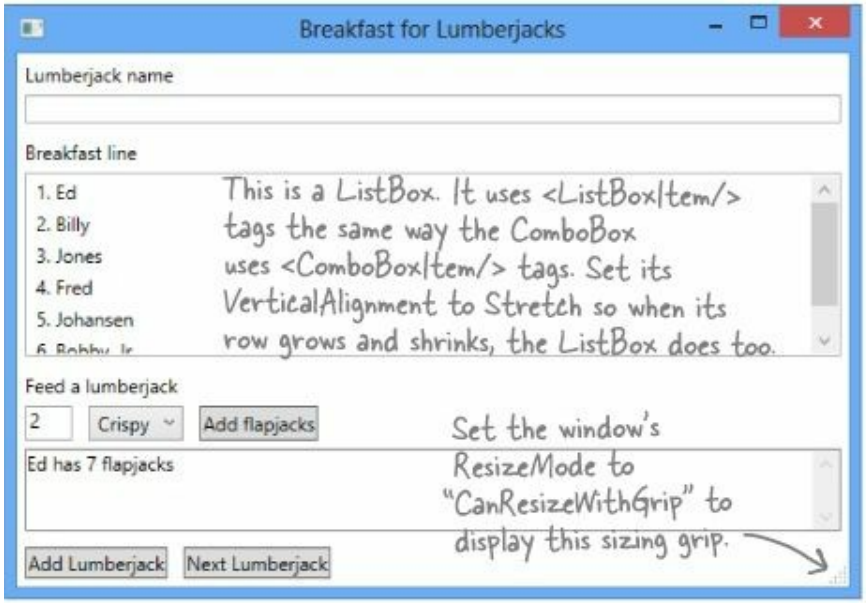




NOTE

Use the Content property to add text to this ScrollViewer.
 will add line breaks. Give it a 2-pixel white border using BorderThickness and BorderBrush, and a height of 250.

Use a Grid to design this form. It has seven rows with `height` set to `Auto` so they expand to fit their contents, and one with the default height (which is the same as `1*`) so that row expands with the grid. Use StackPanels to put multiple controls in the same row. Each TextBlock has a 5-pixel margin below it, and the bottom two TextBlocks each have a 10-pixel margin above them. Use the `<Window>` properties



Use these <Window> properties to set the initial and minimum size for the window, then resize the window to make sure they work: Height="400" MinHeight="350" Width="525" MinWidth="300"

```
<Grid Grid.Row="1" Margin="5">
  <TextBlock/>
  <TextBox/>
  <TextBlock/>
  <ListBox VerticalAlignment="Stretch">
    <ListBoxItem/>
    <ListBoxItem/>
    ... 4 more ...
  </ListBox>
```

Set this row to the default height 1* and make all the other row heights "Auto" so this row grows and shrinks when the window is resized.

```
  <TextBlock>
  <StackPanel Orientation="Horizontal">
    <TextBox/>
    <ComboBox> ... 4 items ... </ComboBox>
    <Button/>
  </StackPanel>
  <ScrollViewer/>
  <StackPanel Orientation="Horizontal">
    <Button/>
    <Button/>
  </StackPanel>
```

Get your pages to look just like these screenshots by adding **dummy data** to the controls that would normally be filled in using the methods and properties in your classes.

EXERCISE SOLUTION

Use XAML to redesign each of these Windows desktop forms as WPF applications. Create a new WPF Application project for each of them, and modify each page by updating or replacing the grid and adding controls. You don't need to get them working. Just create the XAML so they match the screenshots.

```
<Window x:Class="BeehiveManagementSystem.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Beehive Management System"
        Height="300" Width="525"
        ResizeMode="NoResize">

    <StackPanel Margin="5">
        <TextBlock Text="Worker Bee Assignments" Margin="0,0,0,5" />
        <Border BorderThickness="1" BorderBrush="Black">
            <StackPanel Orientation="Horizontal" Margin="5">
                <StackPanel Margin="0,0,10,0">
                    <TextBlock Text="Job"/>
                    <ComboBox SelectedIndex="0" >
                        <ComboBoxItem Content="Baby bee tutoring"/>
                        <ComboBoxItem Content="Egg care"/>
                        <ComboBoxItem Content="Hive maintenance"/>
                        <ComboBoxItem Content="Honey manufacturing"/>
                        <ComboBoxItem Content="Nectar collector"/>
                        <ComboBoxItem Content="Sting patrol"/>
                    </ComboBox>
                </StackPanel>
                <StackPanel>
                    <TextBlock Text="Shifts" />
                    <TextBox/>
                </StackPanel>
                <Button Content="Assign this job to a bee"
                        VerticalAlignment="Bottom" Margin="10,0,0,0" />
            </StackPanel>
        </Border>

        <Button Content="Work the next shift" Margin="0,20,20,0"
                FontSize="18"
                HorizontalAlignment="Right" />

        <TextBlock Text="Shift report" Margin="0,10,0,5"/>
        <Border BorderBrush="Black" BorderThickness="1" Height="100">
            <ScrollViewer
                  Content="
Report for shift #20&#13;
Worker #1 will be done with 'Nectar collector' after this shift&#13;
Worker #2 finished the job&#13;
Worker #2 is not working&#13;
Worker #3 is doing 'Sting patrol' for 3 more shifts&#13;
Worker #4 is doing 'Baby bee tutoring' for 6 more shifts
                        "/>
        </Border>
    </StackPanel>
</Window>
```

Here's the margin we gave you. Specifying just one number (5) sets the top, left, bottom, and right margins to the same value.

Does your XAML code look different from ours? There are many ways to display very similar (or even identical) pages in XAML.

And don't forget that XAML is very flexible about tag order. You can put many of these tags in a different order and still create the same object graph for your window.

This Border control draws a border around the ScrollViewer.

Here's the dummy data we used to populate the shift report. The Content property ignores line breaks—we added them to make the solution easier to read.

```
<Window x:Class="BreakfastForLumberjacks.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Breakfast for Lumberjacks"
        Width="525" Height="400"
        MinWidth="300" MinHeight="350"
        ResizeMode="CanResizeWithGrip" >

    <Grid Grid.Row="1" Margin="5">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition />
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <TextBlock Text="Lumberjack name" Margin="0,0,0,5" />
        <TextBox Grid.Row="1"/>

        <TextBlock Grid.Row="2" Text="Breakfast line" Margin="0,10,0,5" />
        <ListBox Grid.Row="3" VerticalAlignment="Stretch">
            <ListBoxItem Content="1. Ed"/>
            <ListBoxItem Content="2. Billy"/>
            <ListBoxItem Content="3. Jones"/>
            <ListBoxItem Content="4. Fred"/>
            <ListBoxItem Content="5. Johansen"/>
            <ListBoxItem Content="6. Bobby, Jr."/>
        </ListBox>

        <TextBlock Grid.Row="4" Text="Feed a lumberjack" Margin="0,10,0,5" />
        <StackPanel Grid.Row="5" Orientation="Horizontal">
            <TextBox Text="2" Margin="0,0,10,0" Width="30"/>
            <ComboBox SelectedIndex="0" Margin="0,0,10,0">
                <ComboBoxItem Content="Crispy"/>
                <ComboBoxItem Content="Soggy"/>
                <ComboBoxItem Content="Browned"/>
                <ComboBoxItem Content="Banana"/>
            </ComboBox>
            <Button Content="Add flapjacks" />
        </StackPanel>

        <Border BorderThickness="1"  BorderBrush="Gray" Grid.Row="6" Margin="0,5,0,0">
            <ScrollViewer Content="Ed has 7 flapjacks"
                      BorderThickness="2" BorderBrush="White"
                          MinHeight="50"/>
        </Border>

        <StackPanel Grid.Row="7" Orientation="Horizontal" Margin="0,10,0,0">
            <Button Content="Add Lumberjack" Margin="0,0,10,0" />
            <Button Content="Next Lumberjack" />
        </StackPanel>

    </Grid>
</Window>
```

Here are the Window properties that set the initial window size to 525x400, and set a minimum size of 300x350.

You can set the ResizeMode property to NoResize to prevent all resizing, CanMinimize to allow only minimizing, CanResize to allow all resizing, or CanResizeWithGrip to display a sizing grip in the lower right-hand corner of the window.

Just to be 100% clear, we asked you to add these dummy items **as part of the exercise**, to make the form look like it's being used. You're about to learn how to bind controls like this ListBox to properties in your classes.

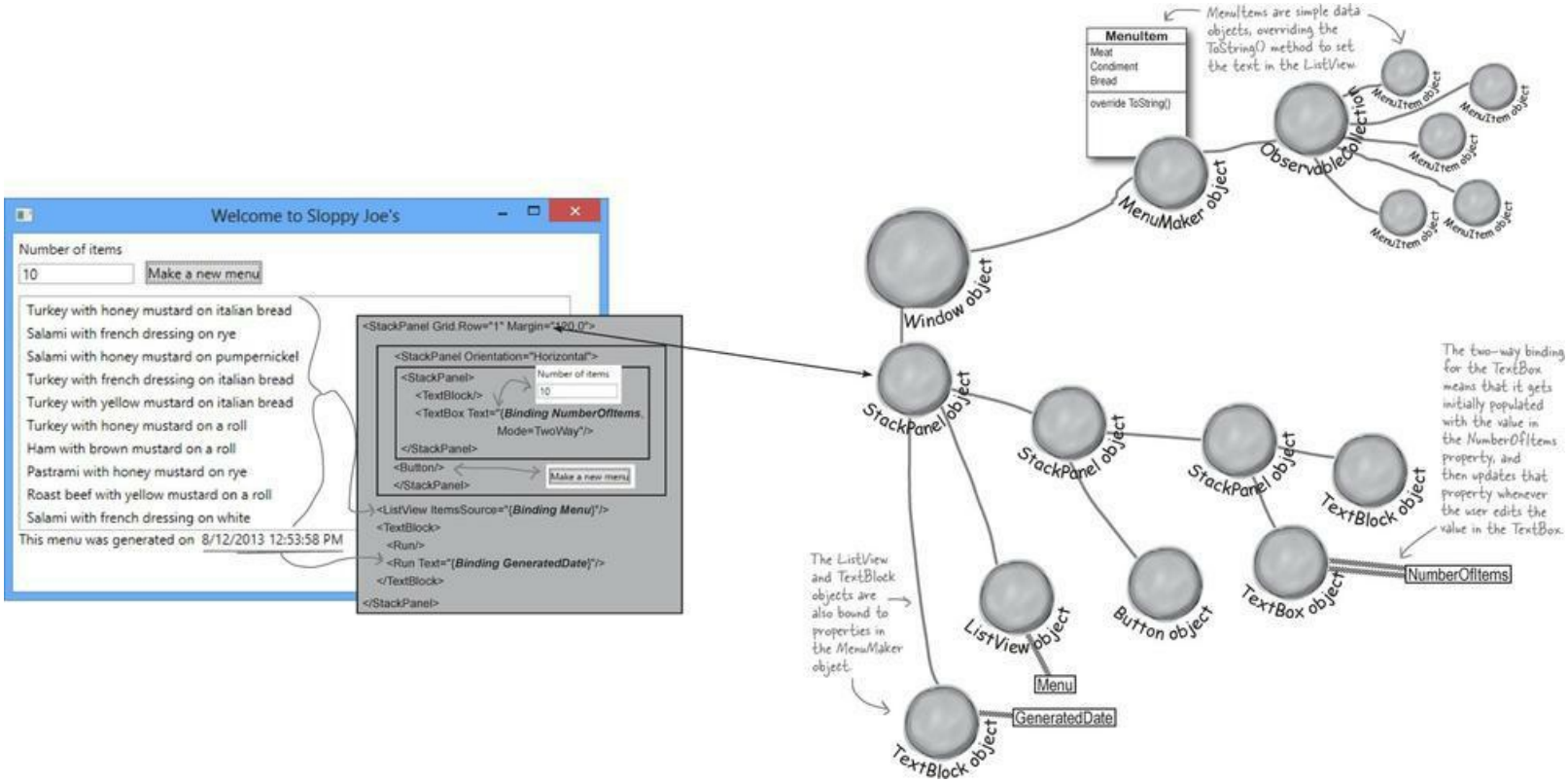
More dummy content...

Use data binding to build Sloppy Joe a better menu

Remember Sloppy Joe from Chapter 4? Well, he's heard that you're becoming an XAML pro, and he wants a WPF app for his sandwich menu. Let's build him one.

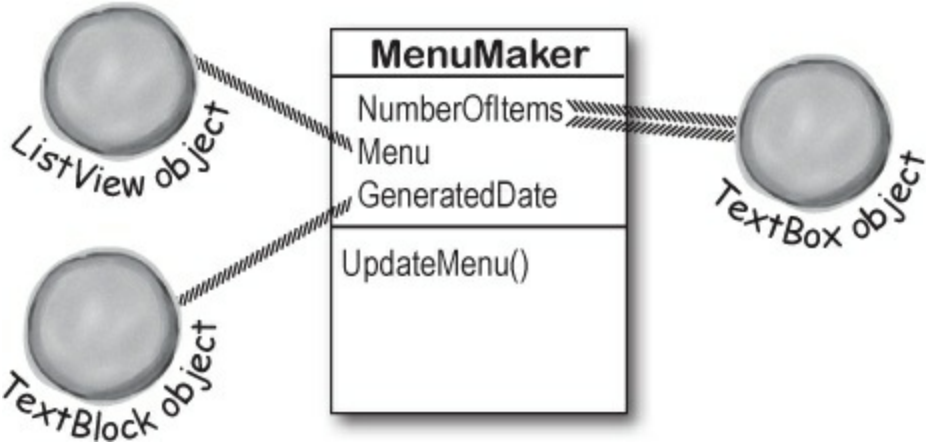
Here's the window we're going to build.

It uses one-way data binding to populate a ListView and a Run inside a TextBlock, and it uses two-way data binding for a TextBox, using one of its `<Run>` tags to do the actual binding.



We'll need an object with properties to bind to.

The `Window` object will have an instance of the `MenuMaker` class, which has three public properties: an int called `NumberOfItems`, an `ObservableCollection` of menu items called `Menu`, and a `DateTime` called `GeneratedDate`.



The `Window` object creates an instance of `MenuMaker` and uses it for the data context.

The constructor for the `Page` object will set the StackPanel's `DataContext` property to an instance of `MenuMaker`. The binding will all be done in XAML.

The TextBox uses two-way binding to set the number of menu items.

That means the TextBox doesn't need an `x:Name` property. Since it's bound to the `NumberOfItems` property in the `MenuMaker` object, we don't need to write any C# code that refers to it.

The button tells the `MenuMaker` to update.

The button calls the `MenuMaker`'s `UpdateMenu()` method, which updates its menu by clearing the `ObservableCollection` and then adding new `MenuItems` to it. The ListView will automatically update anytime the `ObservableCollection` changes.

Here's a coding challenge. Based on what you've read so far, how much of the new and improved Sloppy Joe app can you build before you flip the page and see the code for it?

DO THIS!

- **❶ Create the project.**
  Create a **new WPF Application project**. You'll keep the default window size. Set the window title to *Welcome to Sloppy Joe's*.
- **❷ Add the new and improved `MenuMaker` class.**
  You've come a long way since Chapter 4. Let's build a well-encapsulated class that lets you set the number of items with a property. You'll create an `ObservableCollection` of `MenuItem` in its constructor, which is updated every time the `UpdateMenu()` is called. That method will also update a `DateTime` property called `GeneratedDate` with a timestamp for the current menu. Add this `MenuMaker` class to your project:

Just right-click on the project name in the Solution Explorer and add a new class, just like you did with other projects.

```
using System.Collections.ObjectModel;

class MenuMaker {
    private Random random = new Random();
    private List<String> meats = new List<String>()
                { "Roast beef", "Salami", "Turkey", "Ham", "Pastrami" };
    private List<String> condiments = new List<String>() { "yellow mustard",
             "brown mustard", "honey mustard", "mayo", "relish", "french dressing" };
    private List<String> breads = new List<String>() { "rye", "white", "wheat",
            "pumpernickel", "italian bread", "a roll" };

    public ObservableCollection<MenuItem> Menu { get; private set; }
    public DateTime GeneratedDate { get; set; }
    public int NumberOfItems { get; set; }

    public MenuMaker() {
        Menu = new ObservableCollection<MenuItem>();
        NumberOfItems = 10;
        UpdateMenu();
    }
    private MenuItem CreateMenuItem() {
        string randomMeat = meats[random.Next(meats.Count)];
        string randomCondiment = condiments[random.Next(condiments.Count)];
        string randomBread = breads[random.Next(breads.Count)];
        return new MenuItem(randomMeat, randomCondiment, randomBread);
    }
    public void UpdateMenu() {
        Menu.Clear();
        for (int i = 0; i < NumberOfItems; i++) {
            Menu.Add(CreateMenuItem());
        }
        GeneratedDate = DateTime.Now;
    }
}
```

You'll need this using line because ObservableCollection<T> is in this namespace.

You'll use data binding to display data from these properties on your page. You'll also use two-way binding to update NumberOfItems.

The new CreateMenuItem() method returns MenuItem objects, not just strings. That will make it easier to change the way items are displayed if we want.

Take a closer look at how this works. It never actually creates a new MenuItem collection. It updates the current one by clearing it and adding new items.

What happens if the NumberOfItems is set to a negative number?

USE DATETIME TO WORK WITH DATES

You've already seen the DateTime type that lets you store a date. You can also use it to create and modify dates and times. It has a static property called Now that returns the current time. It also has methods like AddSeconds() for adding and converting seconds, milliseconds, days, etc., and properties like Hour and DayOfWeek to break down the date. How timely!

- **❸ Add the `MenuItem` class.**
  You've already seen how you can build more flexible programs if you use classes instead of strings to store data. Here's a simple class to hold a menu item — add it to your project, too:

```
class MenuItem {
    public string Meat { get; set; }
    public string Condiment { get; set; }
    public string Bread { get; set; }

    public MenuItem(string meat, string condiment, string bread) {
        Meat = meat;
        Condiment = condiment;
        Bread = bread;
    }

    public override string ToString() {
        return Meat + " with " + Condiment + " on " + Bread;
    }
}
```

The three strings that make up the item are passed into the constructor and held in read-only automatic properties.

Override the ToString() method so the MenuItem knows how to display itself.

- **❹ Build the XAML page.**
  Here's the screenshot. Can you build it using StackPanels? The TextBox has a width of 100. The bottom TextBlock has the style `BodyTextStyle`, and it has two `<Run>` tags (the second one just holds the date).

> **NOTE**
>
> Don't add dummy data this time. We'll let data binding do that for us.

Can you build this page on your own just from the screenshot before you see the XAML?

- **❺ Add object names and data binding to the XAML.**
  Here's the XAML that gets added to *MainWindow.xaml*. We used a StackPanel to lay it out, so you can replace the opening `<Grid>` and closing `</Grid>` tags with the XAML below. We named the button `newMenu`. Since we used data binding of the ListView, TextBlock, and TextBox, we didn't need to give them names. *(Here's a shortcut. We didn't even really need to name*

the button; we did it just to get the IDE to automatically add an event handler named newMenu_Click when we double-clicked it in the IDE. Try it out!)

```
<StackPanel Margin="5" x:Name="pageLayoutStackPanel">
    <StackPanel Orientation="Horizontal" Margin="0,0,0,10">
        <StackPanel Margin="0,0,10,0">
            <TextBlock Text="Number of items" Margin="0,0,0,5" />
            <TextBox Width="100" HorizontalAlignment="Left"
                Text="{Binding NumberOfItems, Mode=TwoWay}" />
        </StackPanel>
        <Button x:Name="newMenu" VerticalAlignment="Bottom"
                Click="newMenu_Click" Content="Make a new menu"/>
    </StackPanel>
    <ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0" />
    <TextBlock>
        <Run Text="This menu was generated on " />
        <Run Text="{Binding GeneratedDate}"/>
    </TextBlock>
</StackPanel>
```

We need two-way data binding to both get and set the number of items with the TextBox.

Here's that ListView control. Try swapping it out for ListBox to see how it changes your window.

This is where <Run> tags come in handy. You can have a single TextBlock but bind only part of its text.

- **❻ Add the code-behind for the page to MainWindow.xaml.cs.**
  The page constructor creates the menu collection and the `MenuMaker` instance and sets the data contexts for the controls that use data binding. It also needs a `MenuMaker` field called `menuMaker`.

```
MenuMaker menuMaker = new MenuMaker();

public MainWindow() {
    this.InitializeComponent();

    pageLayoutStackPanel.DataContext = menuMaker;
}
```

Your main window's class in MainWindow.xaml.cs gets a MenuMaker field, which is used as the data context for the StackPanel that contains all the bound controls.

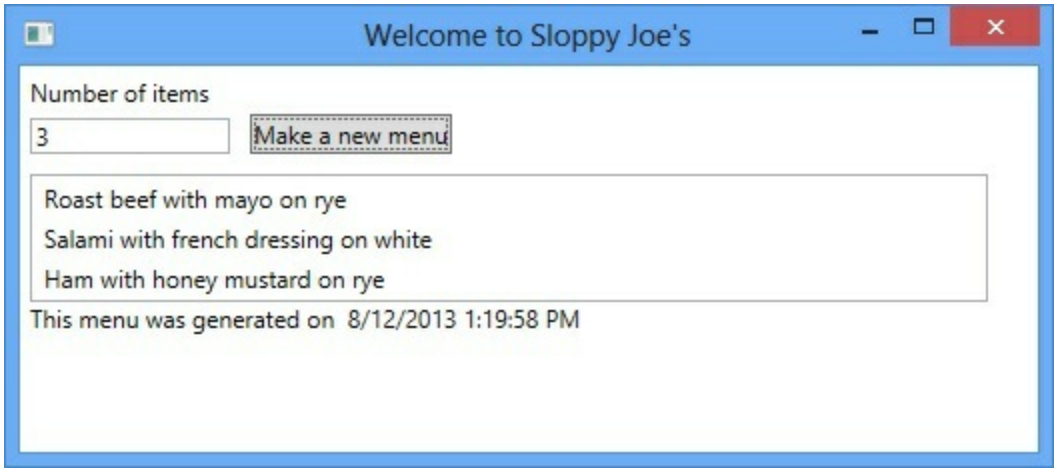
You just need to set the data context for the outer StackPanel. It will pass that data context on to all the controls contained inside it.

Finally, double-click on the button to generate a method stub for its `Click` event handler. Here's the code for it — it just updates the menu:

```
private void newMenu_Click(object sender, RoutedEventArgs e) {
    menuMaker.UpdateMenu();
}
```

There's an easy way to rename an event handler so that it updates XAML and C# code at the same time. Flip to leftover #8 in Appendix A to learn more about the refactoring tools in the IDE.

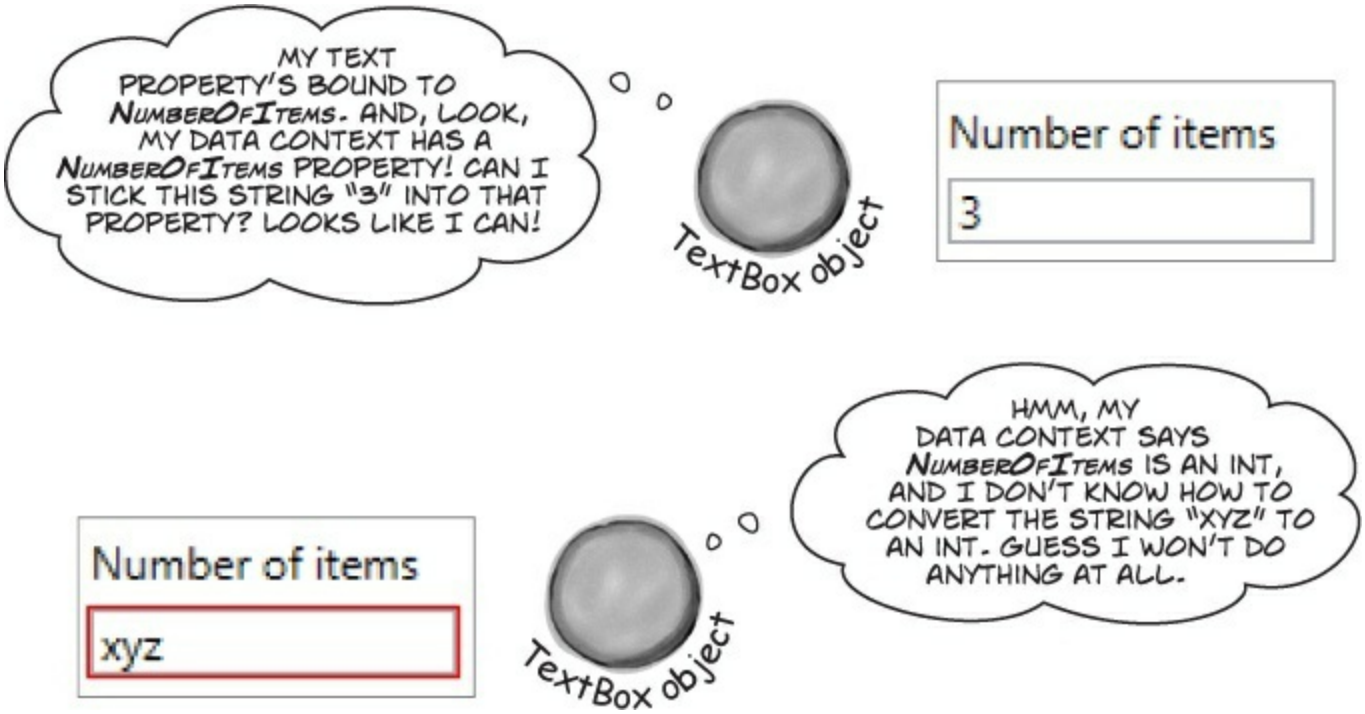
Now run your program! Try changing the TextBox to different values. Set it to 3, and it generates a menu with three items:



Now you can play with binding to see just how flexible it is. Try entering "xyz" or no data at all into the TextBox. Nothing happens! When you enter data into the TextBox, you're giving it a string. The TextBox is pretty smart about what it does with that string. It knows that its binding path is `NumberOfItems`, so it looks in its data context to see if there are any properties with that name, and then does its best to convert the string to whatever that property's type is.

NOTE

Keep your eye on the generated date. It's not updating, even though the menu updates. Hmm, maybe there's still something we need to do.



Use static resources to declare your objects in XAML

When you build a page with XAML, you're creating an object graph with objects like StackPanel, Grid, TextBlock, and Button. And you've seen that there's no magic or mystery to any of that — when you add a `<TextBox>` tag to your XAML, then your page object will have a TextBox field with a reference to an instance of TextBox. And if you give it a name using the `x:Name` property, your code-behind C# code can use that name to access the TextBox.

You can do exactly the same thing to create instances of *almost any* class and store them as fields in your page by adding a **static resource** to your XAML. And data binding works particularly well with static resources, especially when you combine it with the visual designer in the IDE. Let's go back to your program for Sloppy Joe and move the `MenuMaker` to a static resource.

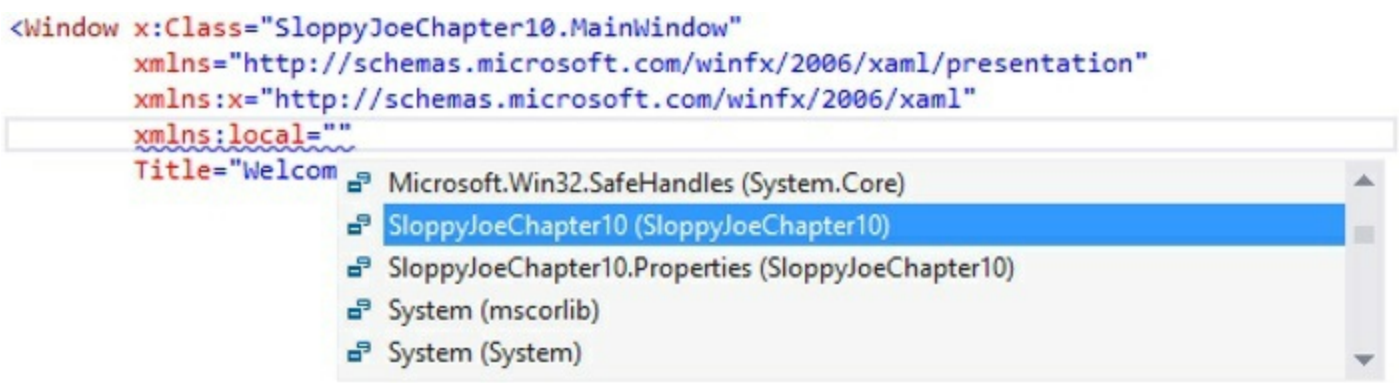
- **❶ DELETE THE MENUMAKER FIELD FROM THE CODE-BEHIND.**
  You're going to be setting up the `MenuMaker` class and the data context in the XAML, so delete these lines from your C# code:

```
~~MenuMaker menuMaker = new MenuMaker();~~

public MainWindow() {
    this.InitializeComponent();

   ~~pageLayoutStackPanel.DataContext = menuMaker;~~
}
```

- **❷ ADD YOUR PROJECT'S NAMESPACE TO THE XAML.**
  Look at the top of the XAML code for your window, and you'll see that the opening tag has a set of `xmlns` properties. Each of these properties defines a namespace:

```
<Window x:Class="SloppyJoeChapter10.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Welcome to Sloppy Joe's" Height="350" Width="525">
```

  Start adding a new xmlns property:



  Here's what you'll end up with:

When the namespace value starts with "using:" it refers to one of the namespaces in the project. It can also start with "http://" to refer to a standard XAML namespace.

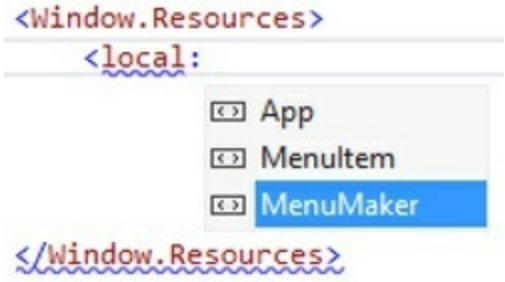
This is an XML namespace property. It consists of "xmlns:" followed by an identifier, in this case "local".

```
xmlns:local="using:SloppyJoeChapter10"
```

You'll use this identifier to create objects in your project's namespace.

Since we named our app SloppyJoeChapter10, the IDE created this namespace for us. Find the namespace that corresponds to your app, because that's where your `MenuMaker` lives.

- **❸ ADD THE STATIC RESOURCE TO YOUR XAML AND SET THE DATA CONTEXT.** Add a `<Window.Resources>` tag to the top of the XAML (just under the opening tag), and add a closing `</Window.Resources>` tag for it. Then type **`<local:`** between them to pop up an IntelliSense window:



You can add static resources only if their classes have parameterless constructors. This makes sense! If the constructor has a parameter, how would the XAML page know what arguments to pass to it?

The window shows all the classes in the namespace that you can use. Choose `MenuMaker`. Then give it the **resource key** `menuMaker` using the **`x:Key`** XAML property:

```
<local:MenuMaker x:Key="menuMaker"/>
```

Now your page has a static `MenuMaker` resource with the key `menuMaker`.

- **❹ SET THE DATA CONTEXT FOR YOUR STACKPANEL AND ALL OF ITS CHILDREN.** Then go to the outermost StackPanel and set its `DataContext` property:

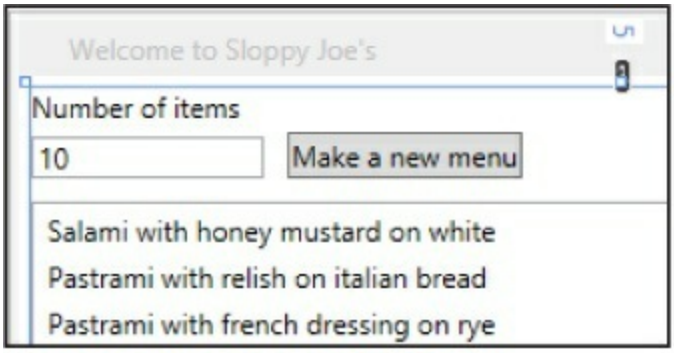
```
<StackPanel Margin="5"
      DataContext="{StaticResource ResourceKey=menuMaker}">
```

Finally, modify the button's Click event handler to find the static resource and method to update the menu:

When you use XAML to add a static resource to a Window, you can access it using its FindResource() method.

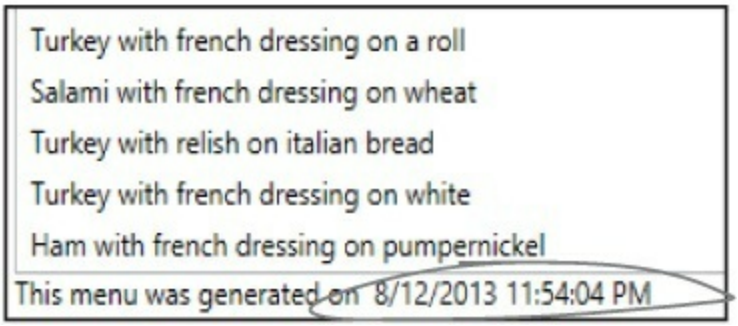
```
private void newMenu_Click(object sender, RoutedEventArgs e) {
    MenuMaker menuMaker = FindResource("menuMaker") as MenuMaker;
    menuMaker.UpdateMenu();
}
```

Your program will still work, just like before. But did you notice what happened in the IDE when you added the data context to the XAML? As soon as you added it, the IDE created an instance of MenuMaker and used its properties to populate all the controls that were bound to it. You got a menu generated immediately, right there in the designer — before you even ran your program. Neat!



The menu shows up in the designer immediately, even before you run your program.

Hmm, something's not quite right. It updates the menu items when the button is clicked, but the date doesn't change. What's going on?



Use a data template to display objects

When you show items in a list, you're showing contents of ListViewItem (which you use for ListViews), ListBoxItem, or ComboBoxItem controls, which get bound to objects in an `ObservableCollection`. Each ListViewItem in the Sloppy Joe menu generator is bound to a `MenuItem` object in its `Menu` collection. The ListViewItem objects call the `MenuMaker` objects' `ToString()` methods by default, but you can use a **data template** that uses data binding to display data from the bound object's properties.

Modify the `<ListView>` tag to add a basic data template. It uses the basic `{Binding}` to call the item's `ToString()`.

Leave the ListView tag intact, but replace /> with > and add a closing </ListView> tag at the bottom. Then add the ListView.ItemTemplate tag to contain the data template.

This is a really basic data template, and it looks just like the default one used to display the ListView Items.

```
<ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0">
    <ListView.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding}"/>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Adding a {Binding} without a path just calls the ToString() method of the bound object.

Change your data template to add some color to your menu.

Replace the <DataTemplate>, but leave the rest of the ListView intact.

You can bind individual Run tags. You can change each tag's color, font, and other properties, too.

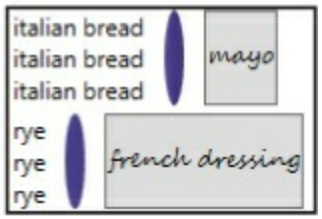
```
<DataTemplate>
    <TextBlock>
        <Run Text="{Binding Meat}" Foreground="Blue"/><Run Text=" on "/>
        <Run Text="{Binding Bread}" FontWeight="Light"/><Run Text=" with "/>
        <Run Text="{Binding Condiment}" Foreground="Red" FontWeight="ExtraBold"/>
    </TextBlock>
</DataTemplate>
```

Salami on pumpernickel with mayo
Pastrami on rye with relish
Salami on white with brown mustard

Go crazy! The data template can contain any controls you want.

```
<DataTemplate>
    <StackPanel Orientation="Horizontal">
        <StackPanel>
            <TextBlock Text="{Binding Bread}"/>
            <TextBlock Text="{Binding Bread}"/>
            <TextBlock Text="{Binding Bread}"/>
        </StackPanel>
        <Ellipse Fill="DarkSlateBlue" Height="Auto" Width="10" Margin="10,0"/>
        <Button Content="{Binding Condiment}" FontFamily="Segoe Script"/>
    </StackPanel>
</DataTemplate>
```

The DataTemplate object's Content property can hold only one object, so if you want multiple controls in your data template, you'll need a container like StackPanel.



THERE ARE NO DUMB QUESTIONS

Q: **Q: So I can use a StackPanel or a Grid to lay out my page. I can use XAML static resources, or I can use fields in code-behind. I can set properties on controls, or I can use data binding. Why are there so many ways to do the same things?**

A: **A:** Because C# and XAML are extremely flexible tools for building apps. That flexibility makes it possible to design very detailed pages that work on many different devices and displays. This gives you a very large toolbox that you can use to get your pages *just right*. So don't look at it as a confusing set of choices; look at it as many different options that you can choose from.

Q: **Q: I'm still not clear on how static resources work. What happens when I add a tag inside `<Window.Resources>`?**

A: **A:** When you add that tag, it updates the Window object and adds static resources. In this case, it **created an instance** of MenuMaker and added it to the Window object's resources. The Window object contains a dictionary called Resources, and if you use the debugger to explore the Window object after you add the tag you can find that it contains an instance of MenuMaker. When you declared the resource, you used `x:Key` to assign the resource a key. That allowed you to **use that key to look up your MenuMaker object** in the window's static resources with the FindResource() method.

Q: **Q: I used `x:Key` to set my MenuMaker resource's key. But earlier in the chapter, I used `x:Name` to give names to my controls. What's the difference? Why did I have to use FindResources() to look up the MenuMaker object — couldn't I give it a name instead?**

A: **A:** When you add a control to a WPF window, it actually adds a field to the Window object that's created by the XAML. When you use the `x:Name` property, you give it a name that you can use in your code. If you don't give it a name, the control object is still created as part of the Window object's graph. However, if you give it a name, then the XAML object is **given a field with that name** with a reference to that control. You can see this in your code by putting a breakpoint in the button's event handler and adding `newMenu` to the Watch window. You'll see that it refers to a System.Windows.Controls. Button object whose Content property is set to "Make a new menu."
Resources are treated differently: they're **added to a dictionary in the Window object**. The FindResource() method uses the key specified in the `x:Key` markup. Set the same breakpoint and try adding `this.Resources["menuMaker"]` to the Watch window. This time, you'll see a reference to your MenuMaker object, because you're looking it up in the Resources dictionary.

Q: **Q: Does my binding path have to be a string property?**

A: **A:** No, you can bind a property of any type. If it can be converted between the source and property types, then the binding will work. If not, the data will be ignored. And remember, not all properties on your controls are text, either. Let's say you've got a bool in your data context called `EnableMyObject`. You can bind it to any Boolean property, like `IsEnabled`. This will enable or disable the control based on the value of the `EnableMyObject` property:

```
IsEnabled="{Binding EnableMyObject}"
```

Of course, if you bind it to a text property it'll just print `True` or `False` (which, if you think about it, makes perfect sense).

Q: **Q: Why did the IDE display the data in my form when I added the static resource and set the data context in XAML, but not when I did it in C#?**

A: **A:** Because the IDE understands your XAML, which has all the information that it needs to create the objects to render your page. As soon as you added the `MenuMaker` resource to your XAML code, the IDE created an instance of `MenuMaker`. But it couldn't do that from the `new` statement in its constructor, because there could be many other statements in the constructor, and they would need to be run. The IDE runs the code-behind C# code only when the program is executed. But if you add a static resource to the page, the IDE will create it, just like it creates instances of TextBlock, StackPanel, and the other controls on your page. It sets the controls' properties to show them in the designer, so when you set up the data context and binding paths, those got set as well, and your menu items showed up in the IDE's designer.

The static resources in your page are instantiated when the page is first loaded and can be used at any time by the objects in the application.

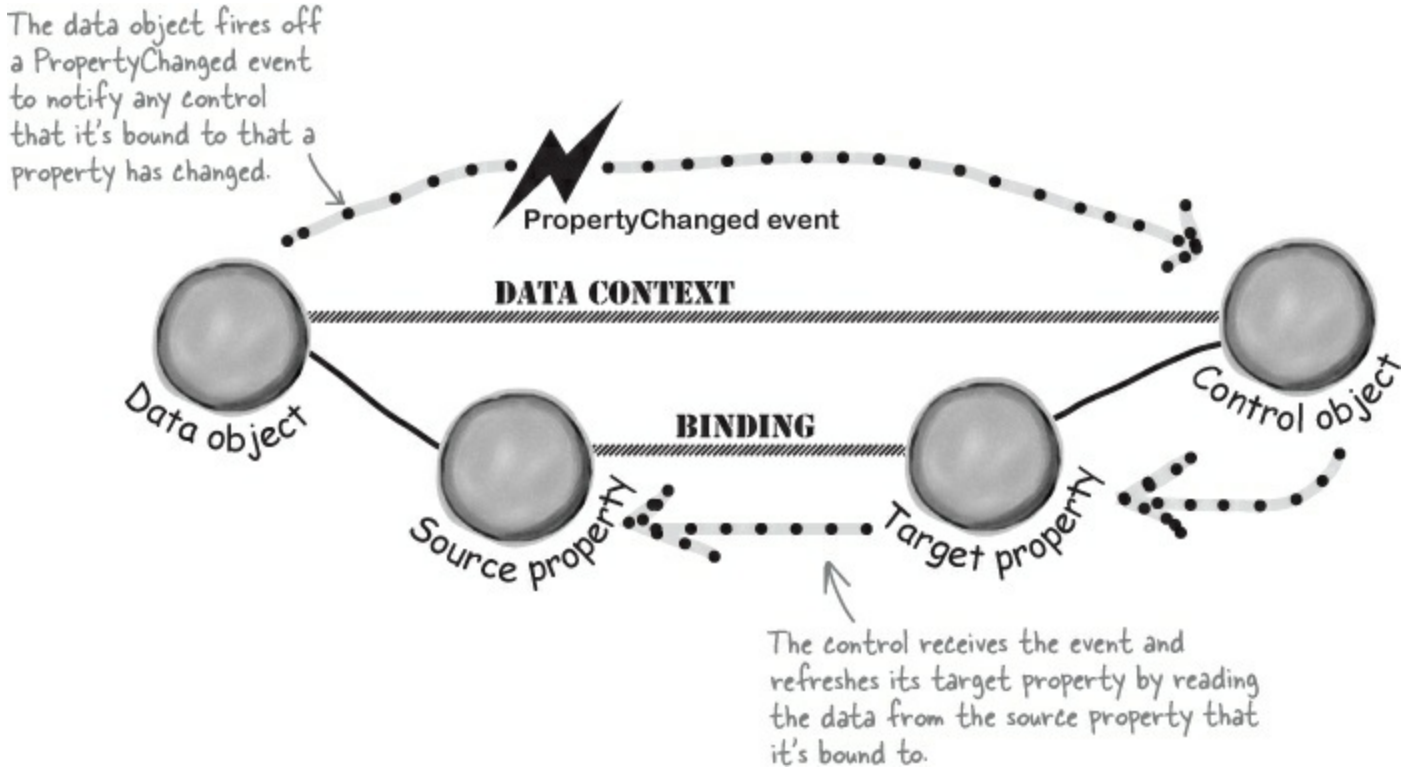
NOTE

The name "static resource" is a little misleading. Static resources are definitely created for each instance; they're not static fields!

INotifyPropertyChanged lets bound objects send updates

When the `MenuMaker` class updates its menu, the ListView that's bound to it gets updated. But the `MenuMaker` updates the `GeneratedDate` property at the same time. Why doesn't the TextBlock that's bound to it get updated, too? The reason is that every time an `ObservableCollection` changes, it **fires off an event** to tell any bound control that its data has changed. This is just like how a `Button` control raises a `Click` event when it's clicked, or a `Timer` raises a `Tick` event when its interval elapses. Whenever you add, remove, or delete items from an `ObservableCollection`, it raises an event.

You can make your data objects notify their target properties and bound controls that data has changed, too. All you need to do is **implement the `INotifyPropertyChanged` interface**, which contains a single event called `PropertyChanged`. Just fire off that event whenever a property changes, and watch your bound controls update themselves automatically.



WATCH IT!

Collections work *almost* the same way as data objects.

The ObservableCollection<T> object doesn't actually implement INotifyPropertyChanged. Instead, it implements a closely related interface called **INotifyCollectionChanged** *that fires off a CollectionChanged event instead of a PropertyChanged event. The control knows to look for this event because ObservableCollection implements the INotifyCollectionChanged interface. Setting a ListView's DataContext to an INotifyCollectionChanged object will cause it to respond to these events.*

Modify MenuMaker to notify you when the GeneratedDate property changes

RELAX

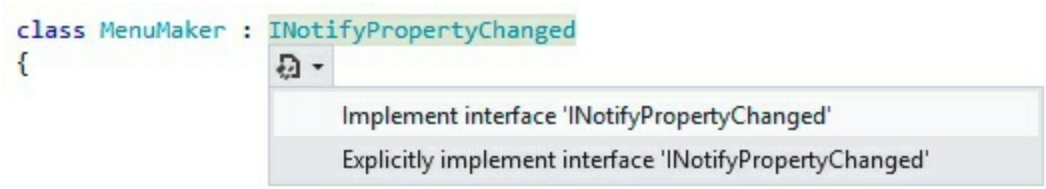
This is the first time you're raising events.

You've been writing event handler methods since Chapter 1, but this is the first time you're firing an event. You'll learn all about how this works and what's going on in Chapter 15. For now, all you need to know is that an interface can include an event, and that your `OnPropertyChanged()` method is following a standard C# pattern for raising events to other objects.

`INotifyPropertyChanged` is in the `System.ComponentModel` namespace, so start by adding this `using` statement to the top of the `MenuMaker` class file:

```
using System.ComponentModel;
```

Update the `MenuMaker` class to implement `INotifyPropertyChanged`, and then use the IDE to automatically implement the interface:



This will be a little different from what you saw in Chapter 7 and Chapter 8. It won't add any methods or properties. Instead, it will add an event:

```
public event PropertyChangedEventHandler PropertyChanged;
```

Next, add this `OnPropertyChanged()` method, which you'll use to raise the `PropertyChanged` event.

```
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

This is a standard .NET pattern for raising events.

Now all you need to do to notify a bound control that a property is changed is to call `OnPropertyChanged()` with the name of the property that's changing. We want the TextBlock that's bound to `GeneratedDate` to refresh its data every time the menu is updated, so all we need to do is add one line to the end of `UpdateMenu()`:

```
public void UpdateMenu() {

    Menu.Clear();

    for (int i = 0; i < NumberOfItems; i++) {

        Menu.Add(CreateMenuItem());

    }

    GeneratedDate = DateTime.Now;
```

```
        OnPropertyChanged("GeneratedDate");
    }
```

Now the date should change when you generate a menu.

WATCH IT!

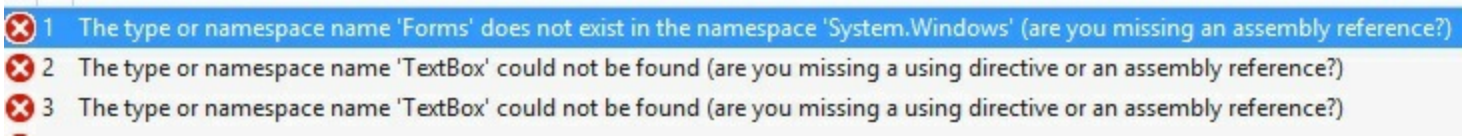
Don't forget to implement `INotifyPropertyChanged`.

Data binding works only when the controls implement that interface. If you leave `: INotifyPropertyChanged` out of the class declaration, your bound controls won't get updated — even if the data object fires PropertyChanged events.

EXERCISE

Finish porting the Go Fish! game to a WPF application. You'll need to modify the XAML from earlier in this chapter to add data binding, copy all the classes and enums from the Go Fish! game in Chapter 8 (or download them from our website), and update the `Player` and `Game` classes.

- **❶ Add the existing class files and change their namespace to match your app.**
  Add these files to your project from the Chapter 8 Go Fish! code: *Values.cs*, *Suits.cs*, *Card.cs*, *Deck.cs*, *CardComparer_bySuit.cs*, *CardComparer_byValue.cs*, *Game.cs*, and *Player.cs*. You can use the Add Existing Item option in the Solution Explorer, but you'll need to **change the namespace** in each of them to match your new projects (just like you did with multipart projects earlier in the book).
  Try building your project. You should get errors in *Game.cs* and *Player.cs* that look like this:



- **❷ Remove all references to WinForms classes and objects; add `using` lines to `Game`.**
  You're not in the WinForms world anymore, so delete `using System.Windows.Forms;` from the top of *Game.cs* and *Player.cs*. You'll also need to remove all mentions of TextBox. You'll need to modify the `Game` class to use `INotifyPropertyChanged` and `ObservableCollection<T>`, so add these `using` lines to the top of *Game.cs*:

```
using System.ComponentModel;
using System.Collections.ObjectModel;
```

- **❸ Add an instance of `Game` as a static resource and set up the data context.**
  Modify your XAML to add an instance of `Game` as a static resource and use it as the data context for the grid that contains the Go Fish! page you built earlier in the chapter. Here's the XAML for the static resource: **`<local:Game x:Key="game"/>`** — and you're going to need a new constructor because you can include only resources that have parameterless constructors:

```
public Game() {
    PlayerName = "Ed";
    Hand = new ObservableCollection<string>();
    ResetGame();
}
```

> **Make sure you add the `<Window.Resources>` section to the top of your XAML, and you'll also need to add the `xmlns:local` tag, exactly like you did in Use static resources to declare your objects in XAML and There are no Dumb Questions.**

- **❹ Add public properties to the `Game` class for data binding.**
  Here are the properties you'll be binding to properties of the controls in the page:

```
public bool GameInProgress { get; private set; }
public bool GameNotStarted { get { return !GameInProgress; } }
public string PlayerName { get; set; }
public ObservableCollection<string> Hand { get; private set; }
public string Books { get { return DescribeBooks(); } }
public string GameProgress { get; private set; }
```

- **❺ Use binding to enable or disable the TextBox, ListBox, and Buttons.**
  You want the "Your Name" TextBox and the "Start the game!" Button to be enabled only when the game is not started, and you want the "Your hand" ListBox and "Ask for a card" Button to be enabled only when the game is in progress. You'll add code to the `Game` class to set the `GameInProgress` property. Have a look at the `GameNotStarted` property. Figure out how it works, and then add the following property bindings to the TextBox, ListBox, and two Buttons:

You'll need two of each of these.

```
IsEnabled="{Binding GameInProgress}"   IsEnabled="{Binding GameNotStarted}"
IsEnabled="{Binding GameInProgress}"   IsEnabled="{Binding GameNotStarted}"
```

- **❻ Modify the `Player` class so it tells the `Game` to display the game's progress.**
  The WinForms version of the `Player` class takes a TextBox as a parameter for its constructor. Change that to take a reference to the `Game` class and store it in a private field. (Look at the `StartGame()` method below to see how this new constructor is used when adding players.) Find the lines that use the TextBox reference and replace them with calls to the `Game` object's `AddProgress()` method.
- **❼ Modify the `Game` class.**
  Change the `PlayOneRound()` method so that it's void instead of returning a Boolean, and have it use the `AddProgress()` method instead of the TextBox to display progress. If a player won, display that progress, reset the game, and return. Otherwise, refresh the `Hand` collection and describe the hands.
  You'll also need to add/update these four methods and figure out what they do and how they work.

```
public void StartGame() {
    ClearProgress();
    GameInProgress = true;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    Random random = new Random();
    players = new List<Player>();
    players.Add(new Player(PlayerName, random, this));
    players.Add(new Player("Bob", random, this));
    players.Add(new Player("Joe", random, this));
    Deal();
    players[0].SortHand();
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
    OnPropertyChanged("Books");
}

  public void ClearProgress() {
      GameProgress = String.Empty;
      OnPropertyChanged("GameProgress");
  }

  public void AddProgress(string progress)
  {
      GameProgress = progress +
             Environment.NewLine +
             GameProgress;
      OnPropertyChanged("GameProgress");
  }

  public void ResetGame() {
      GameInProgress = false;
      OnPropertyChanged("GameInProgress");
      OnPropertyChanged("GameNotStarted");
      books = new Dictionary<Values, Player>();
      stock = new Deck();
      Hand.Clear();
  }
```

You'll **also** need to implement the `INotifyPropertyChanged` interface and add the same `OnPropertyChanged()` method that you used in the `MenuMaker` class. The updated methods use it, and your modified `PullOutBooks()` method will also use it.

EXERCISE SOLUTION

Here's all the code-behind that you had to write.

```
Game game;

public MainWindow() {
    InitializeComponent();
    game = this.FindResource("game") as Game;
}
private void startButton_Click(object sender, RoutedEventArgs e) {
    game.StartGame();
}
private void askForACard_Click(object sender, RoutedEventArgs e) {
    if (cards.SelectedIndex >= 0)
        game.PlayOneRound(cards.SelectedIndex);
}
private void cards_MouseDoubleClick(object sender, MouseButtonEventArgs e) {
    if (cards.SelectedIndex >= 0)
        game.PlayOneRound(cards.SelectedIndex);
}
```

A These are the changes needed for the `Player` class:

```
class Player {
    private string name;
    public string Name { get { return name; } }
    private Random random;
    private Deck cards;
    private Game game;
    public Player(String name, Random random, Game game) {
        this.name = name;
        this.random = random;
        this.game = game;
        this.cards = new Deck(new Card[] { });
        game.AddProgress(name + " has just joined the game");
    }
    public Deck DoYouHaveAny(Values value)
    {
        Deck cardsIHave = cards.PullOutValues(value);
        game.AddProgress(Name + " has " + cardsIHave.Count + " " + Card.Plural(value));
        return cardsIHave;
    }
    public void AskForACard(List<Player> players, int myIndex, Deck stock, Values value) {
        game.AddProgress(Name + " asks if anyone has a " + value);
        int totalCardsGiven = 0;
        for (int i = 0; i < players.Count; i++) {
            if (i != myIndex) {
                Player player = players[i];
                Deck CardsGiven = player.DoYouHaveAny(value);
                totalCardsGiven += CardsGiven.Count;
                while (CardsGiven.Count > 0)
                    cards.Add(CardsGiven.Deal());
            }
        }
        if (totalCardsGiven == 0) {
            game.AddProgress(Name + " must draw from the stock.");
            cards.Add(stock.Deal());
        }
    }
    // ... the rest of the Player class is the same ...
```

These are the changes needed for the XAML:

```
<Grid Margin="10" DataContext="{StaticResource ResourceKey=game}">

    <TextBlock Text="Your Name" />
    <StackPanel Orientation="Horizontal" Grid.Row="1">
        <TextBox x:Name="playerName" FontSize="12" Width="150"
                 Text="{Binding PlayerName, Mode=TwoWay}"
                 IsEnabled="{Binding GameNotStarted}" />
        <Button x:Name="startButton" Margin="5,0" IsEnabled="{Binding GameNotStarted}"
                Content="Start the game!" Click="startButton_Click"/>
    </StackPanel>
    <TextBlock Text="Game progress" Grid.Row="2" Margin="0,10,0,0"/>
    <ScrollViewer Grid.Row="3" FontSize="12" Background="White" Foreground="Black"
                  Content="{Binding GameProgress}" />
    <TextBlock Text="Books" Margin="0,10,0,0" Grid.Row="4"/>
    <ScrollViewer FontSize="12" Background="White" Foreground="Black"
                  Grid.Row="5" Grid.RowSpan="2"
                  Content="{Binding Books}" />
    <TextBlock Text="Your hand" Grid.Row="0" Grid.Column="2" />
    <ListBox x:Name="cards" Background="White" FontSize="12"
             Height="Auto" Margin="0,0,0,10"
             Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"
             ItemsSource="{Binding Hand}" IsEnabled="{Binding GameInProgress}"
             MouseDoubleClick="cards_MouseDoubleClick" />
    <Button x:Name="askForACard" Content="Ask for a card"
        HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
        Grid.Row="6" Grid.Column="2"
        Click="askForACard_Click" IsEnabled="{Binding GameInProgress}" />
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="5*"/>
        <ColumnDefinition Width="40"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto" MinHeight="150" />
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
</Grid>
```

The data context for the grid is the Game class, since all of the binding is to properties on that class.

The TextBox has a two-way binding to PlayerName.

Here's the Click event handler for the Start button.

The Game Progress and Books ScrollViewers bind to the Progress and Books properties.

The IsEnabled property enables or disables the control. It's a Boolean property, so you can bind it to a Boolean property to turn the control on or off based on that property.

EXERCISE SOLUTION

Here's everything that changed in the `Game` class, including the code we gave you with the instructions.

```
using System.ComponentModel;
using System.Collections.ObjectModel;

class Game : INotifyPropertyChanged {
    private List<Player> players;
    private Dictionary<Values, Player> books;
    private Deck stock;

    public bool GameInProgress { get; private set; }
    public bool GameNotStarted { get { return !GameInProgress; } }
    public string PlayerName { get; set; }
    public ObservableCollection<string> Hand { get; private set; }
    public string Books { get { return DescribeBooks(); } }
    public string GameProgress { get; private set; }

    public Game() {
        PlayerName = "Ed";
        Hand = new ObservableCollection<string>();
        ResetGame();
    }

    public void AddProgress(string progress) {
        GameProgress = progress + Environment.NewLine + GameProgress;
        OnPropertyChanged("GameProgress");
    }

    public void ClearProgress() {
        GameProgress = String.Empty;
        OnPropertyChanged("GameProgress");
    }

    public void StartGame() {
        ClearProgress();
        GameInProgress = true;
        OnPropertyChanged("GameInProgress");
        OnPropertyChanged("GameNotStarted");
        Random random = new Random();
        players = new List<Player>();
        players.Add(new Player(PlayerName, random, this));
        players.Add(new Player("Bob", random, this));
        players.Add(new Player("Joe", random, this));
        Deal();
        players[0].SortHand();
        Hand.Clear();
        foreach (String cardName in GetPlayerCardNames())
            Hand.Add(cardName);
        if (!GameInProgress)
            AddProgress(DescribePlayerHands());
        OnPropertyChanged("Books");
    }
```

You need these lines for INotifyPropertyChanged and ObservableCollection.

These properties are used by the XAML data binding.

Here's the new Game constructor. We create only one collection and just clear it when the game is reset. If we created a new object, the form would lose its reference to it, and the updates would stop.

These methods make the game progress data binding work. New lines are added to the top so the old activity scrolls off the bottom of the ScrollViewer.

Here's the StartGame() method we gave you. It clears the progress, creates the players, deals the cards, and then updates the progress and books.

Every program you've written in the book so far can be adapted or rewritten as a WPF application using XAML. But there are so many ways to write them, and that's especially true when you're using XAML! That's why we gave you so much of the code for this exercise.

This used to return a Boolean value so the form could update its progress. Now it just needs to call AddProgress, and data binding will take care of the updating for us.

```
public void PlayOneRound(int selectedPlayerCard) {
    Values cardToAskFor = players[0].Peek(selectedPlayerCard).Value;
    for (int i = 0; i < players.Count; i++) {
        if (i == 0)
            players[0].AskForACard(players, 0, stock, cardToAskFor);
        else
            players[i].AskForACard(players, i, stock);
        if (PullOutBooks(players[i])) {
            AddProgress(players[i].Name + " drew a new hand");
            int card = 1;
            while (card <= 5 && stock.Count > 0) {
                players[i].TakeCard(stock.Deal());
                card++;
            }
        }
        OnPropertyChanged("Books");
        players[0].SortHand();
        if (stock.Count == 0) {
            AddProgress("The stock is out of cards. Game over!");
            AddProgress("The winner is... " + GetWinnerName());
            ResetGame();
            return;
        }
    }
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
}
```

The books changed, and the form needs to know about the change so it can refresh its ScrollViewer.

Here are the modifications to the PlayOneRound() method that update the progress when the game is over, or update the hand and the books if it's not.

This is the ResetGame() method from the instructions. It clears the books, stock, and hand.

```
public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}
```

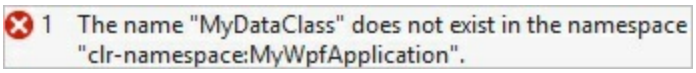
This is the standard PropertyChanged event pattern from earlier in the chapter.

```
public event PropertyChangedEventHandler PropertyChanged;
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}

// ... the rest of the Game class is the same ...
```

WATCH IT!

Are you getting a strange XAML error about a class not existing in the namespace? Make sure that ALL your C# code compiles and that every control's event handler method is declared in the code-behind.

Sometimes you'll get an error like this when you declare a static resource, even though you definitely have a class called MyDataClass in the namespace MyWpfApplication:



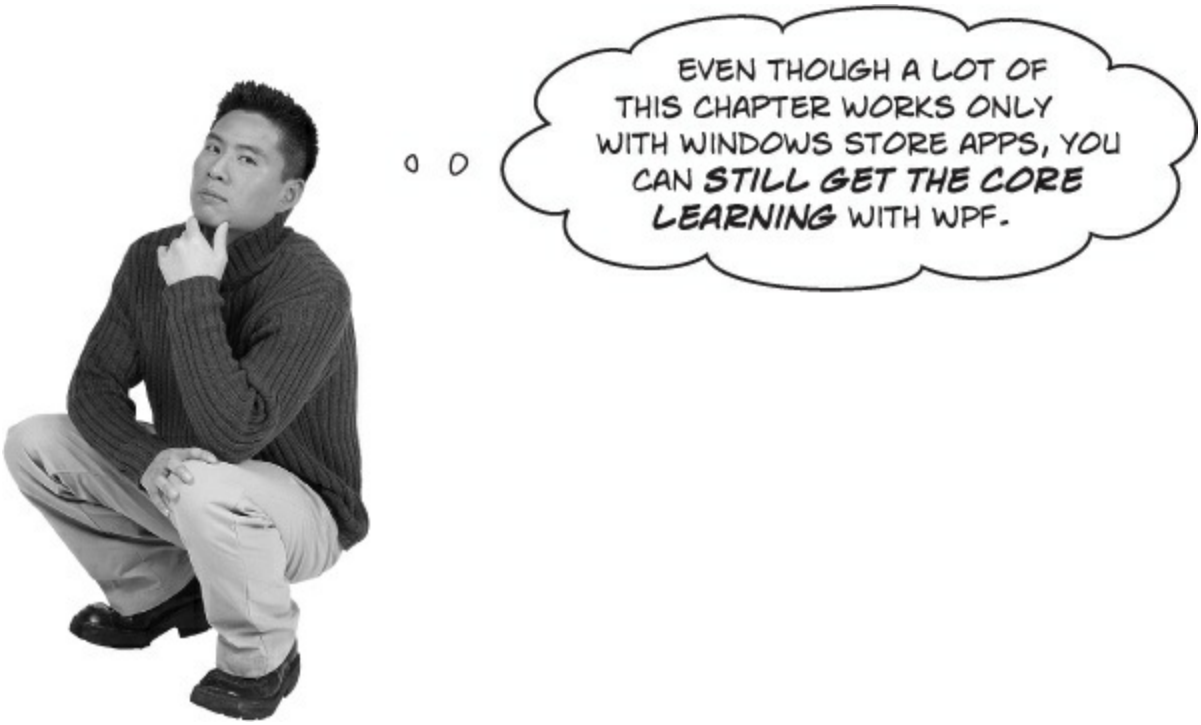
This is often caused by either an error in the code-behind or a missing event handler for a XAML control. This can be a little misleading, because the IDE is telling you that there's an error on the tag that declares the static resource, when the error is actually somewhere else in the code.

You can reproduce this yourself: create a new WPF project called MyWpfApplication, add a data class called MyDataClass, add it as a static resource to your page's <Window.Resources>, and add a button to your page. Then add `Click="Button_Click"` *to the XAML to add an event handler for the button, but* ***don't add the Button_Click() method****. When you try to rebuild your code, you should see the error above. You can make it go away by adding the Button_Click() method to the code-behind.*

NOTE

Sometimes the error message becomes a little clearer if you right-click on the project in the Solution Explorer, click "Unload Project" to unload it, and then right-click it again and choose "Reload Project" to load it again. This may cause the IDE to show you a different error message that might be more helpful.

Chapter 11



Windows Store was built for asynchronous programming, but WPF can still use it... but not all the tools are there.

Read through Brian runs into file trouble and Note in the main part of the book — see how Brian is shocked (shocked!) to find that his familiar file classes from Chapter 9 aren't there? Well, WPF apps don't have that problem. That's a good thing, because it means you can keep using the file classes and serialization that you're used to. But it also means that your WPF apps can't take advantage of the new asynchronous file and dialog classes that come with the .NET Framework for Windows Store.

In this appendix, we'll give you two replacement projects to show you how to use the `async` and `await` keywords and data contract serialization with WPF apps. Here's how we recommend that you work through Chapter 11:

- Windows Store apps use await to be more responsive and Do this! have replacements in this appendix. Use the replacements in place of the book pages.
- Use the FileIO class to read and write files–??? are specific to Windows Store apps. Skip them.
- Read Note and A data contract is an abstract definition of your object's data to learn about data contract serialization.
- Skip Use async methods to find and open files, Use async methods to find and open files, and KnownFolders helps you access high-profile folders; they apply only to Windows Store apps.
- Read The whole object graph is serialized to XML in the book. Then follow the "Do this!" project on the replacement Stream some Guy objects to XML files–Take your Guy Serializer for a test drive in this appendix.
- The rest of the chapter has you build a Windows Store replacement for Brian's excuse manager. The goal of this project is to learn about the file tools in the Windows.Storage namespace for Windows Store apps. We don't have a WPF alternative for this project, because those classes are specific to Windows Store apps.

C# programs can use await to be more responsive

What happens when you call `MessageBox.Show()` from a WinForms program? Everything stops, and your program freezes until the dialog disappears. That's literally the most unresponsive that a program can be! Windows Store apps should always be responsive, even when they're waiting for feedback from a user. But some things — like waiting for a dialog, or reading or writing all the bytes in a file — take a long time. When a method sits there and makes the rest of the program wait for it to complete, programmers call that **blocking**, and it's one of the biggest causes of program unresponsiveness.

Windows Store apps use **the `await` operator and the `async` modifier** to keep from becoming unresponsive during operations that block. You can see how it works by looking at an example of how a WPF could call a define task that blocks, but can be called asynchronously:

Declare the method using the async modifier to indicate that it can be called asynchronously.

```
private async Task LongTaskAsync()
{
    await Task.Delay(5000);
}
```

The Task class is in the System.Threading.Tasks namespace. Its Delay() method blocks for a specified number of milliseconds. That method is really similar to the Thread.Sleep() method that you used in Chapter 2, but it's defined with the async modifier so it can be called asynchronously with await.

The `await` operator causes the method that's running this code to stop and wait until the `ShowAsync()` method completes — and that method will block until the user chooses one of the commands. In the meantime, the rest of the program ***will keep responding to other events***. As soon as the `LongTaskAsync()` method returns, the method that called it will pick up where it left off (although it may wait until after any other events that started up in the meantime have finished).

If your method uses the `await` operator, then it **must be declared with the `async` modifier**:

```
private async void countButton_Click(object sender, RoutedEventArgs e) {
    // ... some code ...
    await LongTaskAsync();
    // ... some more code:
}
```

Notice how this is a Click event handler. Since it uses await, it also needs to be declared with the async modifier.

When a method is declared with `async`, you have some options with how you call it. If you call the method as usual, then as soon as it hits the `await` statement it returns, which keeps the blocking call from freezing your app.

You can see exactly how this works by **creating a new WPF application** with the following main window XAML:

Do this!

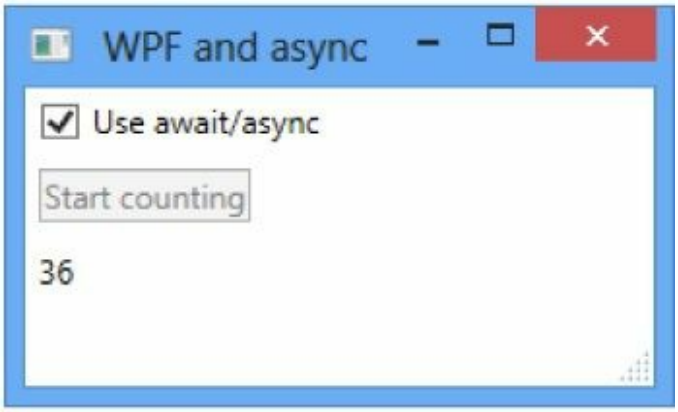
```
<Window x:Class="WpfAndAsync.MainWindow"
```

```
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WPF and async" Height="150" Width="200" ResizeMode="CanResizeWithGrip">
    <Grid>
        <StackPanel>
            <CheckBox x:Name="useAwaitAsync" IsChecked="True" Content="Use await/async" Margin="5"/>
            <Button x:Name="countButton" Content="Start counting"
                    HorizontalAlignment="Left" Click="countButton_Click" Margin="5"/>
            <TextBlock x:Name="progress" HorizontalAlignment="Left" Margin="5" />
        </StackPanel>
    </Grid>
</Window>
```

Here's the code-behind:

We named our project WpfAndAsync. If you named your project something else, you'll need to change this line to match its namespace: `x:Class="WpfAndAsync.MainWindow"`

```
using System.Threading;
using System.Windows.Threading;

public partial class MainWindow : Window {
    DispatcherTimer timer = new DispatcherTimer();

    public MainWindow() {
        InitializeComponent();

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

    int i = 0;
    void timer_Tick(object sender, EventArgs e) {
        progress.Text = (i++).ToString();
    }

    private async void countButton_Click(object sender, RoutedEventArgs e) {
        countButton.IsEnabled = false;
        timer.Start();
        if (useAwaitAsync.IsChecked == true)
            await LongTaskAsync();
        else
            LongTask();
        countButton.IsEnabled = true;
    }

    private void LongTask() {
        Thread.Sleep(5000);
        timer.Stop();
    }

    private async Task LongTaskAsync() {
        await Task.Delay(5000);
        timer.Stop();
    }
}
```



The button's event handler uses the CheckBox's IsChecked property. If the box is checked, the event handler calls `await LongTaskAsync()`, which is asynchronous. The method is called with `await`, so the event handler method pauses and lets the rest of the program continue to run. Try adding other buttons to the window that change properties or print output to the console. You'll be able to use them while the timer ticks.

If the CheckBox is not checked, IsChecked is false and the button's event handler calls `LongTask()`, which blocks. This causes the event handler method to block, which makes the entire program become unresponsive, and if you add other buttons they won't respond either.

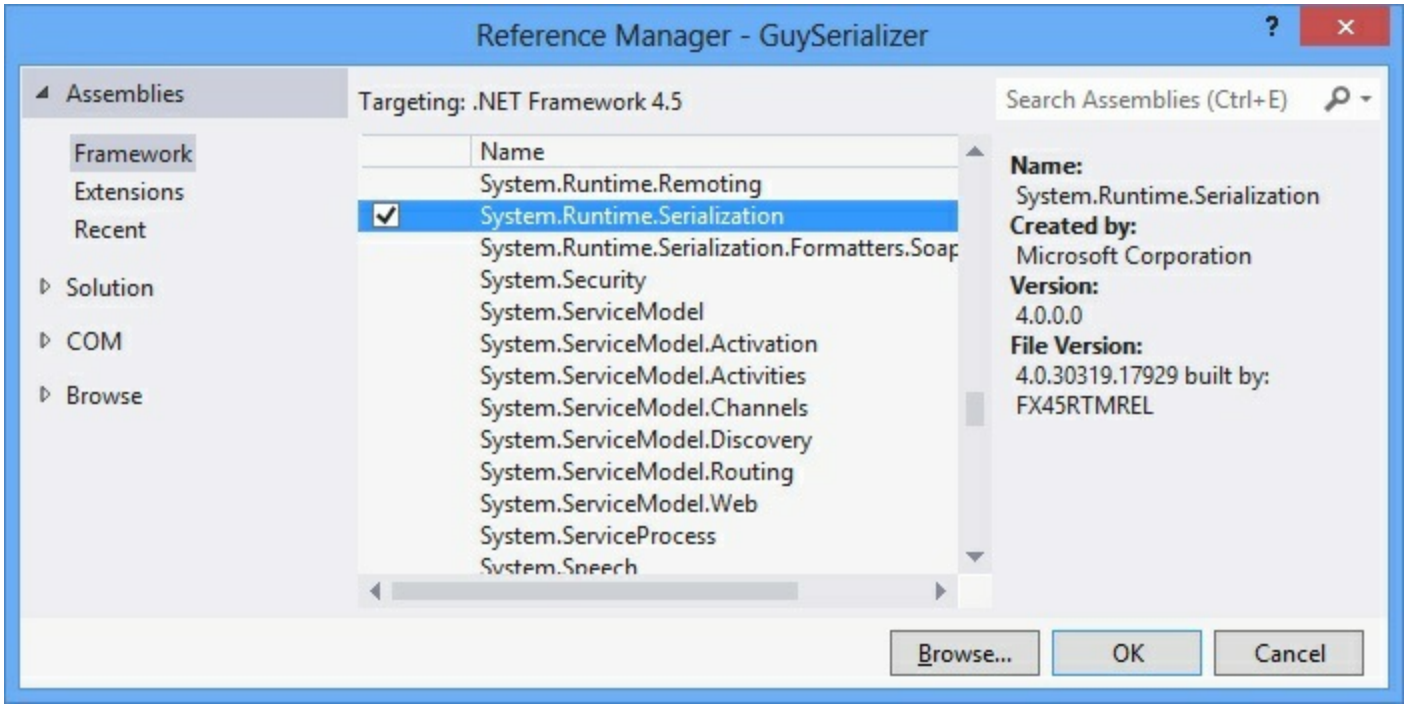
Make sure the box is checked, and then click the button. You'll see the numbers increase, and the form is responsive: the button disables itself, and you can move and resize the form. Then uncheck the box and click the button — now the form freezes.

Stream some Guy objects to a file

Here's a project to help you experiment with data contract serialization. **Create a new WPF application**. Then **add both classes** with the data contracts from The whole object graph is serialized to XML in the book (you'll need `using System.Runtime.Serialization` in each of them). And add the familiar `Suits` and `Values` enums, too (for the `Card` class). Here's the window you'll build next:

Do this!

- ❶ Before you start coding, you'll need to **right-click on References in the Solution Explorer and choose Add Reference** from the menu. Click on Framework, scroll down to System.Runtime.Serialization, check it, and click OK:



This will allow your WPF application to use the System.Runtime.Serialization namespace. You can also add an empty GuyManager class to get rid of the IDE error on the `<local:GuyManager>` tag when you add the XAML in step 2. You'll fill in the GuyManager in step 3 when you flip the page.

- ❷ Here's the XAML for the page.

We named this project GuySerializer. If your project has a different namespace, make sure you change these lines to match it.

```
<Window x:Class="GuySerializer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:GuySerializer"
        Title="Guy Serializer" Height="275" Width="525" ResizeMode="NoResize">

    <Window.Resources>
        <local:GuyManager x:Key="guyManager"/>
    </Window.Resources>

    <Grid DataContext="{StaticResource guyManager}" Margin="5">
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition Height="4*"/>
            <RowDefinition Height="3*"/>
        </Grid.RowDefinitions>

        <StackPanel>
            <Button x:Name="WriteJoe" Content="Write Joe"
                    HorizontalAlignment="Left" Click="WriteJoe_Click"/>
            <TextBlock Text="{Binding Joe}" Margin="0,0,10,20" TextWrapping="Wrap"/>
        </StackPanel>

        <StackPanel Grid.Column="1">
            <Button x:Name="WriteBob" Content="Write Bob"
                    HorizontalAlignment="Left" Click="WriteBob_Click"/>
            <TextBlock Text="{Binding Bob}" Margin="0,0,0,20"  TextWrapping="Wrap"/>
        </StackPanel>

        <StackPanel Grid.Column="2" Margin="10,0,0,0">
            <Button x:Name="WriteEd" Content="Write Ed"
                    HorizontalAlignment="Left" Click="WriteEd_Click"/>
            <TextBlock Text="{Binding Ed}" Margin="0,0,0,20"  TextWrapping="Wrap"/>
        </StackPanel>

        <StackPanel Grid.Row="1" Grid.ColumnSpan="2" Margin="0,0,20,0">
            <TextBlock>Last filename written</TextBlock>
            <TextBox Text="{Binding GuyFile, Mode=TwoWay}"
                    TextWrapping="Wrap" Height="60" Margin="0,0,0,20"/>
        </StackPanel>

        <StackPanel Grid.Row="1" Grid.Column="2"  Margin="10,0,0,0">
            <Button x:Name="ReadNewGuy" Content="Read a new Guy"
                    HorizontalAlignment="Left" Click="ReadNewGuy_Click" />
            <StackPanel>
                <TextBlock Text="New guy:"/>
                <TextBlock TextWrapping="Wrap" Text="{Binding NewGuy}"/>
            </StackPanel>
        </StackPanel>
    </Grid>
</Window>
```

The grid's data context is the GuyManager static resource.

The page has three columns and two rows.

Each column in the top row has a StackPanel with a TextBlock and a Button.

This TextBlock is bound to the Ed property in GuyManager.

The first cell in the bottom row spans two columns. It has several controls bound to properties. Why do you think we used a TextBox for the path?

We're not done yet — flip the page!

- ❸ Add the `GuyManager` class.

```
using System.ComponentModel;
using System.IO;
using System.Runtime.Serialization;

class GuyManager : INotifyPropertyChanged
{
    private Guy joe = new Guy("Joe", 37, 176.22M);
    public Guy Joe
    {
        get { return joe; }
    }

    private Guy bob = new Guy("Bob", 45, 4.68M);
    public Guy Bob
    {
        get { return bob; }
    }

    private Guy ed = new Guy("Ed", 43, 37.51M);
    public Guy Ed
    {
        get { return ed; }
    }

    public Guy NewGuy { get; set; }

    public string GuyFile { get; set; }


    public void ReadGuy()
    {
        if (String.IsNullOrEmpty(GuyFile))
            return;

        using (Stream inputStream = File.OpenRead(GuyFile))
        {
            DataContractSerializer serializer = new DataContractSerializer(typeof(Guy));
            NewGuy = serializer.ReadObject(inputStream) as Guy;
        }
        OnPropertyChanged("NewGuy");
    }
```

There are three read-only Guy properties with private backing fields. The XAML has a TextBlock bound to each of them.

A fourth TextBlock is bound to this Guy property, which is set by the ReadGuy() method.

The ReadGuy() method uses familiar System.IO methods to open a stream and read from it. But instead of using a BinaryFormatter, it uses a DataContractSerializer to serialize data from an XML file.

This program uses TextBoxes that are bound to read-only properties that have only get accessors. If you try to bind to a property that has a public get accessor with a private set accessor, you'll get an error. Luckily, a backing field will work just fine.

```
        public void WriteGuy(Guy guyToWrite)
        {
            GuyFile = Path.GetFullPath(guyToWrite.Name + ".xml");

            if (File.Exists(GuyFile))
                File.Delete(GuyFile);
            using (Stream outputStream = File.OpenWrite(GuyFile))
            {
                DataContractSerializer serializer = new DataContractSerializer(typeof(Guy));
                serializer.WriteObject(outputStream, guyToWrite);
            }

            OnPropertyChanged("GuyFile");
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void OnPropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
            if (propertyChangedEvent != null)
            {
                propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
```

This uses the GetFullPath() method in the Path class (in System.IO) to get the full path of the filename to write.

If the file exists, it's deleted, then recreated using a file stream. It's serialized using the data contract serializer.

Here's the same code you used earlier to implement INotifyPropertyChanged and fire off PropertyChanged events.

- ❹ Here's the code-behind for *MainWindow.xaml.cs*:

```
public partial class MainWindow : Window
{
    GuyManager guyManager;

    public MainWindow() {
        InitializeComponent();

        guyManager = FindResource("guyManager") as GuyManager;
    }

    private void WriteJoe_Click(object sender, RoutedEventArgs e) {
        guyManager.WriteGuy(guyManager.Joe);
    }
    private void WriteBob_Click(object sender, RoutedEventArgs e) {
        guyManager.WriteGuy(guyManager.Bob);
    }
    private void WriteEd_Click(object sender, RoutedEventArgs e) {
        guyManager.WriteGuy(guyManager.Ed);
    }
    private void ReadNewGuy_Click(object sender, RoutedEventArgs e) {
        guyManager.ReadGuy();
    }
}
```

Take your Guy Serializer for a test drive

Use the Guy Serializer to experiment with data contract serialization:

- Write each `Guy` object to the files — they'll be written to the bin\Debug folder in your projects folder. Click the ReadGuy button to read the guy that was just written. It uses the path in the TextBox to read the file, so try updating that path to read a different guy. Try reading a file that doesn't exist. What happens?
- Open up the Simple Text Editor you built earlier in the chapter. You added XML files as options for the open and save file pickers, so you can use it to edit Guy files. Open one of the Guy files, change it, save it, and read it back into your Guy Serializer. What happens if you add invalid XML? What if you change the card suit or value so it doesn't match a valid enum value?
- Try adding or removing the `DataMember` names (`[DataMember(Name="...")]`). What does that do to the XML? What happens when you update the contract and then try to load a previously saved XML file? Can you fix the XML file to make it work?
- Try changing the namespace of the `Card` data contract. What happens to the XML?

THERE ARE NO DUMB QUESTIONS

Q: **Q: Sometimes I make a change in my XAML or my code, and the IDE's designer gives me a message that I need to rebuild. What's going on?**

A: **A:** The XAML designer in the IDE is really clever. It's able to show you an updated page in real time as you make changes to your XAML code. You already know that when the XAML uses static resources, that adds object references to the `Page` class. Well, those objects need to get instantiated in order for them to be displayed in the designer. If you make a change to the class that's being used for a static resource, the designer doesn't get updated until you rebuild that class. That makes sense — the IDE rebuilds your project only when you ask it to, and until you do that it doesn't actually have the compiled code in memory that it needs to instantiate the static resources.

You can use the IDE to see exactly how this works. Open your Guy Serializer and edit the `Guy.ToString()` method to add some extra words to the return value. Then go back to the main page designer. It's still showing the old output. Now choose Rebuild from the Build menu. The designer will update itself as soon as the code finishes rebuilding. Try making another change, but don't rebuild yet. Instead, add another TextBlock that's bound to a `Guy` object. The IDE will use the old version of the object until you rebuild.

Q: **Q: I'm confused about namespaces. How is the namespace in the program different from the one in an XML file?**

A: **A:** Let's take a step back and understand why namespaces are necessary. C#, XML files, the Windows filesystem, and web pages all use different (but often related) naming systems to give each class, XML document, file, or web page its own unique name. So why is this important? Well, let's say back in Chapter 9, you created a class called `KnownFolders` to help Brian keep track of excuse folders. Uh-oh! Now you find out that the .NET Framework for Windows Store already has a `KnownFolders` class. No worries. The .NET `KnownFolders` class is in the Windows.Storage namespace, so it can exist happily alongside your class with the same name, and that's called **disambiguation**.

Data contracts also need to disambiguate. You've seen several different versions of a `Guy` class throughout this book. What if you wanted to have two different contracts to serialize different versions of `Guy`? You can put them in different namespaces to disambiguate them. And it makes sense that these namespaces would be separate from the ones for your classes, because you can't really confuse classes and contracts.

One more thing. Your WPF applications can use the same OpenFileDialog and SaveFileDialog classes that you used in your WinForms projects. Here's an MSDN page that has more information and code samples:

http://msdn.microsoft.com/en-us/library/aa969773.aspx

Chapter 12

Exception handling works the same in WPF as it does in WinForms and Windows Store.

If you flip through the replacement pages for Chapter 12, you'll notice that there's no XAML. That's because the material on exception handling that we cover in *Head First C#* is basically the same whether you're working on a WPF application, a WinForms program, a Windows Store app, or even a console application.

Here's how you should use this appendix for Chapter 12:

- Read through There are no Dumb Questions in the book, including the "Sharpen your Pencil" exercise.
- Use the appendix replacement pages for Brian's code did something unexpected and 577.
- Read All exception objects inherit from Exception and The debugger helps you track down and prevent exceptions in your code in the book.
- Follow Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager–If you have code that ALWAYS should run, use a finally block in this appendix, and skip 591 in the main part of the book.
- Finish the rest of the chapter in the book.
- Then do all of Chapter 13 in the book, too!

Once you're done with this chapter, you can go straight through Chapter 13 in the book. It doesn't depend on Windows 8 or Windows Store apps at all.

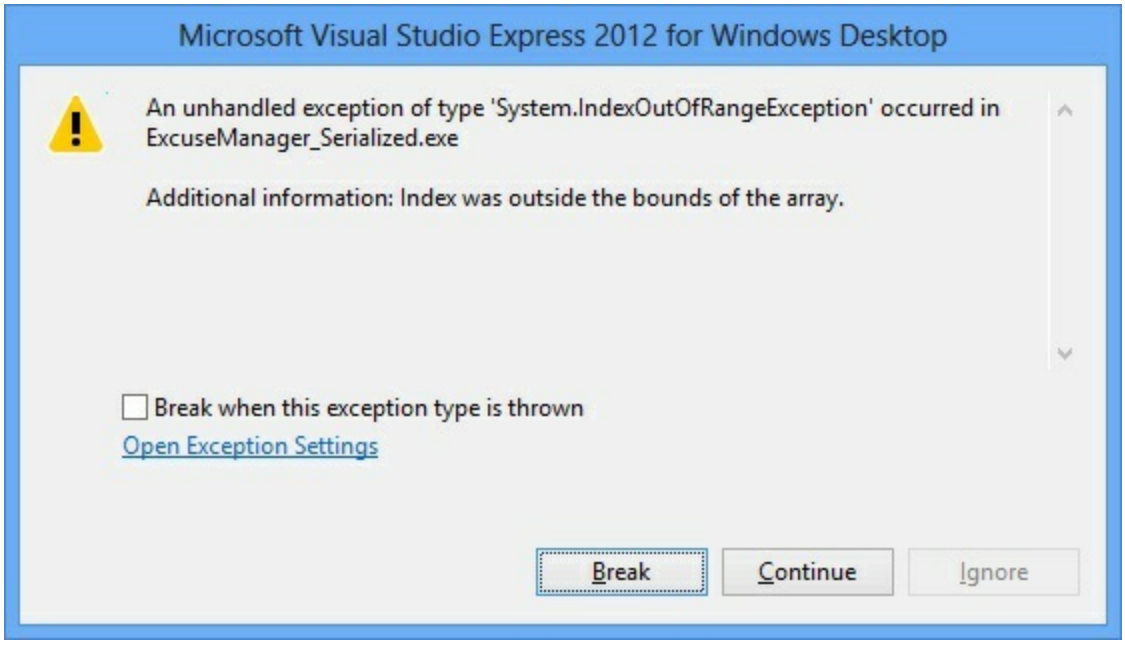
Brian's code did something unexpected

This appendix depends on the Excuse Manager WinForms app that you built in Chapter 9. If your code doesn't match the code in the appendix, you can download it from http://headfirstlabs.com/hfcsharp.

When Brian wrote his Excuse Manager, he never expected the user to try to pull a random excuse out of an empty directory.

DO THIS!

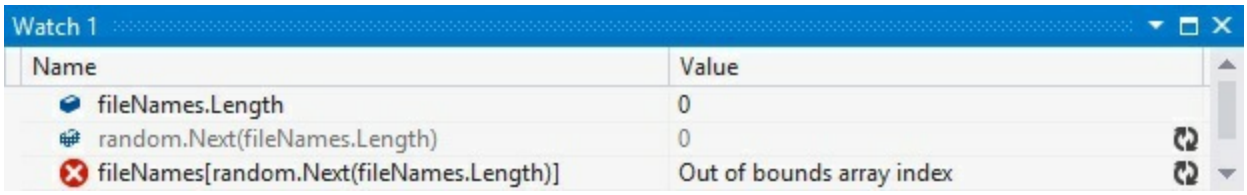
- ❶ The problem happened when Brian pointed his Excuse Manager program at an empty folder on his laptop and clicked the Random Excuse button. Let's take a look at it and see if we can figure out what went wrong. Here's the unhandled exception window that popped up when he ran the program in the IDE:



- ❷ OK, that's a good starting point. It's telling us that there's some value that doesn't fall inside some range. Clicking the Break button drops the IDE back into the debugger, with the execution halted on a specific line of code:

```
public Excuse(Random random, string folder)
{
    string[] fileNames = Directory.GetFiles(folder, "*.excuse");
    OpenFile(fileNames[random.Next(fileNames.Length)]);
}
```

- ❸ Let's use the Watch window to track down the problem. Add a watch for `fileNames.Length`. Looks like that returns 0. Try adding a watch for `random.Next(fileNames.Length)`. That returns 0, too. So add a watch for `fileNames[random.Next(fileNames.Length)]`. This time the Value column in the Watch window has the same error message that you saw in step 1: "Out of bounds array index."



NOTE

You can call methods and use indexers in the Watch window. When one of those things throws an exception, you'll see that exception in the Watch window, too.

- ❹ So what happened? It turns out that `Directory.GetFiles()` returns an empty array when you point it at an empty folder. So

`fileNames.Length` is zero, and passing 0 to `Random.Next()` will always return 0 as well. Try to get the 0th element of an empty array and your program will throw a `System.IndexOutOfRangeException`, with the message "Index was outside the bounds of the array."

Now that we know what the problem is, we can fix it. All we need to do is check to see if the selected folder has excuses in it before we try to load a random excuse from it:

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    if (Directory.GetFiles(selectedFolder).Length == 0)
        MessageBox.Show("There are no excuse files in the selected folder.");
    else if (CheckChanged())
    {
        currentExcuse = new Excuse(random, selectedFolder);
        UpdateForm(false);
    }
}
```

By checking for excuse files in the folder **before** we create the Excuse object, we can prevent the exception from being thrown—and display a helpful dialog, too.

What do you think about that solution? Does it make the most sense to put it in the form, or would it be better to find a way to encapsulate it inside the Excuse class?

That's right. Exceptions are a really useful tool that you can use to find places where your code acts in ways you don't expect.

A lot of programmers get frustrated the first time they see an exception. But exceptions are really useful, and you can use them to your advantage. When you see an exception, it's giving you a lot of clues to help you figure out when your code is reacting to a situation that you didn't anticipate. And that's good for you: it lets you know about a new scenario that your program has to handle, and it gives you an opportunity to **do something about it**.

Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager

Let's use the debugger to take a closer look at the problem that we ran into in the Excuse Manager. You've probably been using the debugger a lot over the last few chapters, but we'll go through it step by step anyway to make sure we don't leave out any details.

DEBUG THIS

- **❶ ADD A BREAKPOINT TO THE RANDOM BUTTON'S EVENT HANDLER.**
  You've got a starting point — the exception happens when the Random Excuse button is clicked after an empty folder is selected. So open up the button's event handler and use Debug→Toggle Breakpoint (F9) to add a breakpoint to the first line of the method. Start debugging, **choose an empty folder**, and then click the Random button to make your program break at the breakpoint:

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    if (Directory.GetFiles(selectedFolder).Length == 0)
        MessageBox.Show("There are no excuse files in the selected folder.");
    else if (CheckChanged())
    {
        currentExcuse = new Excuse(random, selectedFolder);
        UpdateForm(false);
    }
}
```

- **❷ STEP INTO THE EXCUSE CONSTRUCTOR.**
  We want to reproduce the problem, but we already added code to get past it. No problem. Right-click on the line `currentExcuse = new Excuse(random, selectedFolder);` and choose **Set Next Statement** (Ctrl+Shift+F10). Then use **Step Into** (F11) to step into the constructor:

```
public Excuse(Random random, string folder)
{
    string[] fileNames = Directory.GetFiles(folder, "*.excuse");
    OpenFile(fileNames[random.Next(fileNames.Length)]);
}
```

You used the debugger to skip past the workaround that you added to avoid the exception, so now the Excuse constructor is about to throw the exception again.

- **❸ STEP THROUGH THE PROGRAM UNTIL IT THROWS THE EXCEPTION.**
  You've already seen how handy the Watch window is. Now we'll use it to reproduce the exception. Choose Step Over (F10) twice to get your program to throw the exception. Then use the IDE to select `fileNames.Length`, right-click on it, and choose Add Watch to add a watch. Then do it again for `random.Next(fileNames.Length)` and `fileNames[random.Next(fileNames.Length)]`:

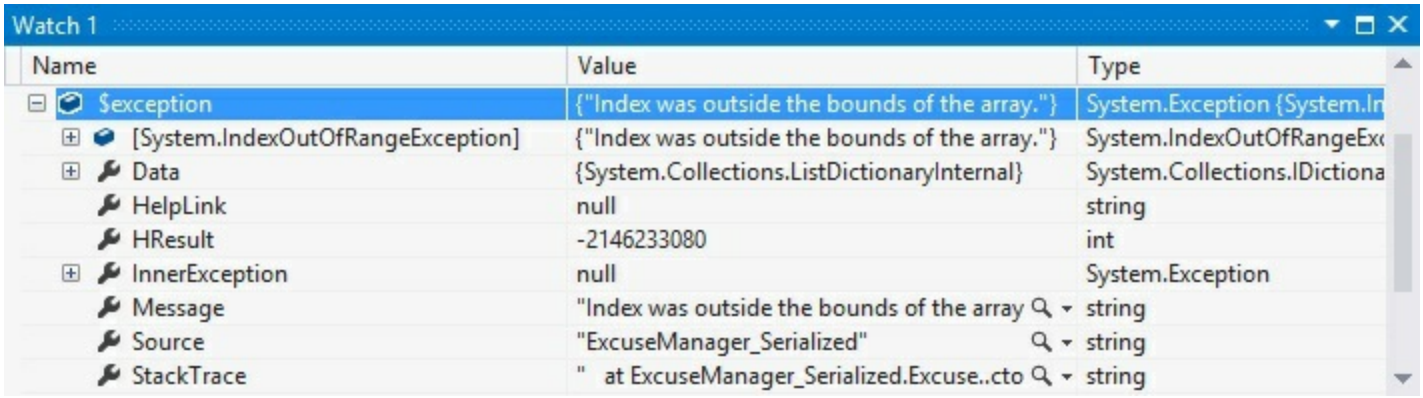
Watch 1

Name	Value
fileNames.Length	0
random.Next(fileNames.Length)	0
fileNames[random.Next(fileNames.Length)]	Out of bounds array index

The Watch window has another very useful feature. It lets you **change the value** of variables and fields that it's displaying, and it even lets you **execute methods and create new objects**. When you do, it displays its reevaluate icon 🔄 that you can click to tell it to execute that method again.

- **❹ ADD A WATCH FOR THE EXCEPTION OBJECT.**

Debugging is a little like *performing a forensic crime scene investigation on your program.* You don't necessarily know what you're looking for until you find it, so you need to use your debugger "CSI kit" to follow clues and track down the culprit. One important tool is adding `$exception` to the Watch window, because it shows you the contents of the Exception object that's been thrown:



When you get an exception, you can go back and reproduce it in the debugger and use the `Exception` object to help you fix your code.

THERE ARE NO DUMB QUESTIONS

Q: **Q: How do I know where to put a breakpoint?**

A: **A:** That's a really good question, and there's no one right answer. When your code throws an exception, it's always a good idea to start with the statement that threw it. But usually, the problem actually happened earlier in the program, and the exception is just fallout from it. For example, the statement that throws a divide-by-zero error could be dividing values that were generated 10 statements earlier but just haven't been used yet. So there's no one good answer to where you should put a breakpoint, because every situation is different. But as long as you've got a good idea of how your code works, you should be able to figure out a good starting point.

Q: **Q: Can I run any method in the Watch window?**

A: **A:** Yes. Any statement that's valid in your program will work inside the Watch window, even things that make absolutely no sense to run inside a Watch window. Here's an example. Bring up a program, start it running, break it, and then **add this to the Watch window**: `System.Threading.Thread.Sleep(2000)`. That method causes your program to delay for two seconds.There's no reason you'd ever do that in real life, but it's interesting to see what happens: the IDE will block and you'll get a wait cursor for two seconds while the method evaluates. Then, since `Sleep()` has no return value, the Watch window will display the value `Expression has been evaluated and has no value` to let you know that it didn't return anything. But it did evaluate it. Not only that, but it displays IntelliSense pop-ups to help you type code into the window. That's useful because it shows the available properties and methods for objects currently in memory.

Q: **Q: Wait, so isn't it possible for me to run something in the Watch window that'll change the way my program runs?**

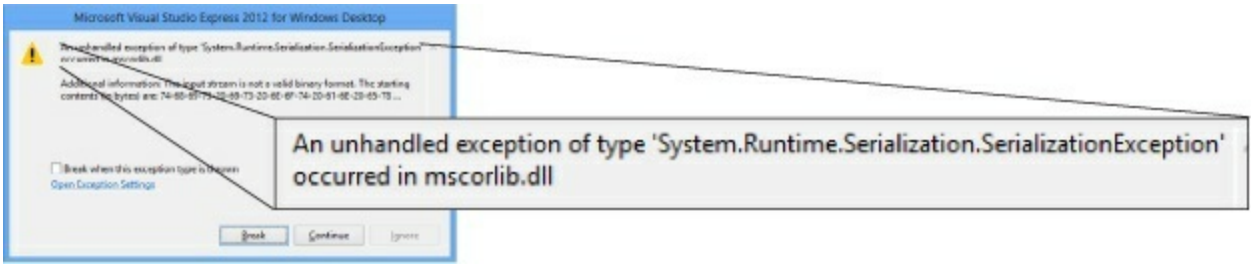
A: **A:** Yes! Not permanently, but it can definitely affect your program's output. But even better, just ***hovering*** over fields inside the debugger can cause your program to change its behavior, because hovering over a property ***executes its*** `get` ***accessor***. If you have a property that has a `get` accessor that executes a method, then hovering over that property will cause that method to execute. And if that method sets a value in your program, then that value will stay set if you run the program again. And that can cause some pretty unpredictable results inside the debugger. Programmers have a name for results that seem to be unpredictable and random: they're called **heisenbugs** (which is a joke that makes sense to physicists and cats trapped in boxes).

When you run your program inside the IDE, an unhandled exception will cause it to break as if it had run into a breakpoint.

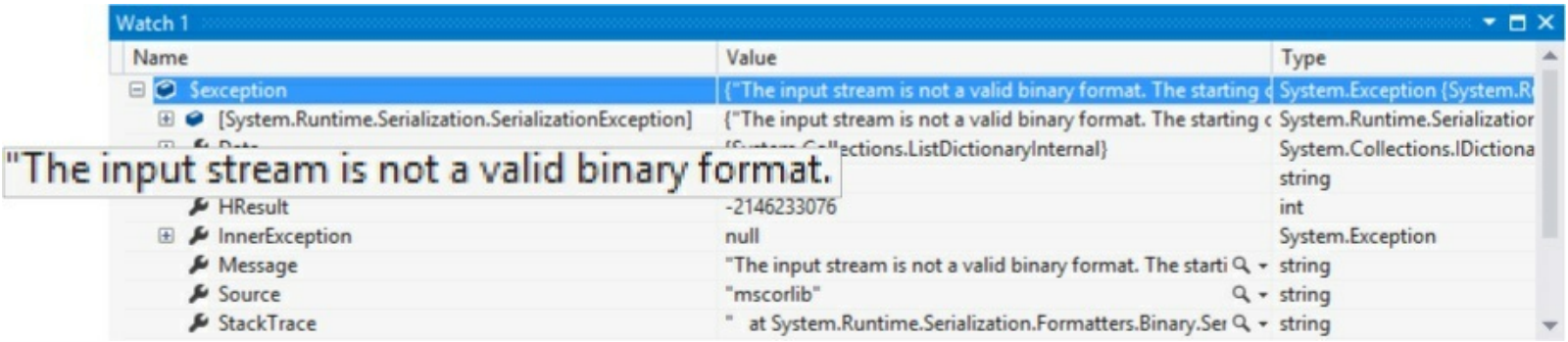
Uh-oh — the code's still got problems...

Brian was happily using his Excuse Manager when he accidentally chose a folder full of files that weren't created by the Excuse Manager. Let's see what happens when he tries to load one of them....

- ❶ You can re-create Brian's problem. Take a random file that isn't a serialized excuse and give it the .excuse file extension.
- ❷ Pop open the Excuse Manager in the IDE and open up the file you created. It throws an exception! Look at the message, then click the Break button to start investigating.



- ❸ Open up the Locals window and expand `$exception` (you can also enter it into the Watch window). Take a close look at its members to see if you can figure out what went wrong.



DO YOU SEE WHY THE PROGRAM THREW THE EXCEPTION?
DOES IT MAKE SENSE FOR THE PROGRAM TO CRASH IF IT ENCOUNTERS AN INVALID EXCUSE XML FILE?
CAN YOU THINK OF ANYTHING YOU CAN DO ABOUT THIS?

Actually, there is something you can do about it.

Yes, it's true that users screw up all the time. That's a fact of life. But that doesn't mean you can't do anything about it. There's a name for programs that deal with bad data, malformed input, and other unexpected situations gracefully: they're called **robust** programs. And C# gives you some really powerful exception handling tools to help you make your programs more robust. Because while you *can't* control what your users do, you *can* make sure that your program doesn't crash when they do it.

NOTE

ro-bust, adj.

sturdy in construction; able to withstand or overcome adverse conditions. *After the Tacoma Narrows Bridge disaster, the civil engineering team looked for a more **robust** design for the bridge that would replace it.*

WATCH IT!

Serializers will throw an exception if there's anything at all wrong with a serialized file.

It's easy to get the Excuse Manager to throw a `SerializationException` — just feed it any file that's not a serialized Excuse object. When you try to deserialize an object from a file, `DataContractSerializer` expects the file to contain a serialized object that matches the contract of the class that it's trying to read. If the file contains anything else, almost anything at all, then the `ReadObject()` method will throw a `SerializationException`.

NOTE

The BinaryFormatter class will also throw a SeralizationException if you give it a file that doesn't contain exactly the right serialized object. It's even more finicky than DataContractSerializer!

Handle exceptions with try and catch

In C#, you can basically say, "**Try** this code, and if an exception occurs, **catch** it with this *other* bit of code." The part of the code you're trying is the **`try` block**, and the part where you deal with exceptions is called the **`catch` block**. In the `catch` block, you can do things like print a friendly error message instead of letting your program come to a screeching halt:

> **NOTE**
>
> You'll also need to add these lines to the top of Excuse.cs:
>
> ```
> using System.Runtime.Serialization;
> using System.Windows.Forms;
> ```

Put the code that might throw an exception inside the try block. If no exception happens, it'll get run exactly as usual, and the statements in the catch block will be ignored. But if a statement in the try block throws an exception, the rest of the try block **won't get executed**.

```
private void OpenFile(string excusePath) {
    try
    {
        this.ExcusePath = excusePath;
        BinaryFormatter formatter = new BinaryFormatter();
        Excuse tempExcuse;
        using (Stream input = File.OpenRead(excusePath))
        {
            tempExcuse = (Excuse)formatter.Deserialize(input);
        }
        Description = tempExcuse.Description;
        Results = tempExcuse.Results;
        LastUsed = tempExcuse.LastUsed;
    }
    catch (SerializationException)
    {
        MessageBox.Show("Unable to read " + excusePath);
        LastUsed = DateTime.Now;
    }
}
```

This is the **try block**. You start exception handling with try. In this case, we'll put the existing code in it.

You'll recognize the code here because we surrounded the entire method with this try block.

The catch keyword means that the block immediately following it contains an **exception handler**.

When an exception is thrown, the program immediately jumps to the **catch** statement and starts executing the **catch block**.

What happens if you leave out this last line of code? Can you figure out why we included it in the catch block?

This is the simplest kind of exception handling: stop the program, write out the exception message, and keep running.

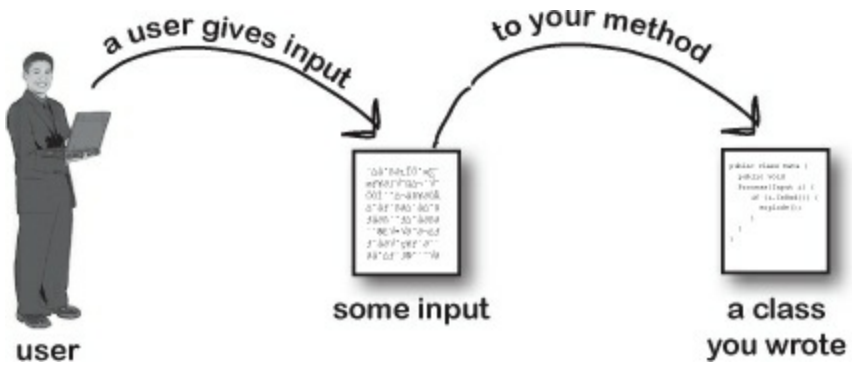
BRAIN POWER

If throwing an exception makes your code automatically jump to the `catch` block, what happens to the objects and data you were working with before the exception happened?

What happens when a method you want to call is risky?

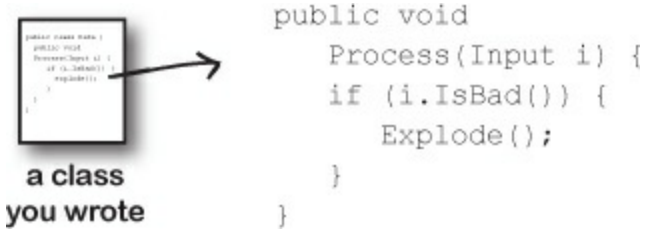
Users are unpredictable. They feed all sorts of weird data into your program and click on things in ways you never expected. And that's just fine, because you can handle unexpected input with good exception handling.

- **❶ Let's say your user is using your code and gives it some input that it didn't expect.**



- **❷ That method does something risky, something that might not work at runtime.**

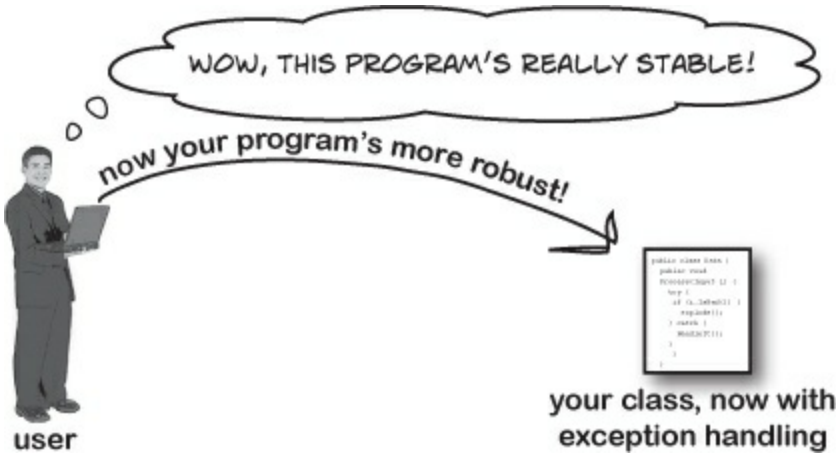
> **NOTE**
>
> "Runtime" just means "while your program is running." Some people refer to exceptions as "runtime errors."



- **❸ You need to *know* that the method you're calling is risky.**

- **❹ You then write code that can handle the failure if it does happen. You need to be prepared, just in case.**

> **NOTE**
>
> If you can come up with a way to do a less risky thing that avoids throwing the exception, that's the best possible outcome! But some risks just can't be avoided, and that's when you want to do this.



THERE ARE NO DUMB QUESTIONS

Q: **Q: So when do I use `try` and `catch`?**

A: **A:** Anytime you're writing risky code, or code that could throw an exception. The trick is figuring out which code is risky, and which code is safer. You've already seen that code that uses input provided by a user can be risky. Users give you incorrect files, words instead of numbers, and names instead of dates, and they pretty much click everywhere you could possibly imagine. A good program will take all that input and work in a calm, predictable way. It might not give the users a result they can use, but it will let them know that it found the problem and hopefully suggest a solution.

Q: **Q: How can a program suggest a solution to a problem it doesn't even know about in advance?**

A: **A:** That's what the `catch` block is for. A `catch` block is executed only when code in the `try` block throws an exception. It's your chance to make sure the user knows that something went wrong, and to let the user know that it's a situation that might be corrected.
If the Excuse Manager simply crashes when there's bad input, that's not particularly useful. But if it tries to read the input and displays garbage in the form, that's also not useful — in fact, some people might say that it's worse. But if you have the program display an error message telling the user that it couldn't read the file, then the user has an idea of what went wrong, and information that he can use to fix the problem.

Q: **Q: So the debugger should really only be used to troubleshoot exceptions then?**

A: **A:** No. As you've already seen many times throughout the book, the debugger's a really useful tool that you can use to examine any code you've written. Sometimes it's useful to step through your code and check the values of certain fields and variables — like when you've got a really complex method and you want to make sure it's working properly.
But as you may have guessed from the name "debugger," its most common use is to track down and remove bugs. Sometimes those bugs are exceptions that get thrown. But a lot of the time, you'll be using the debugger to try to find other kinds of problems, like code that gives a result that you don't expect.

Q: **Q: I'm not sure I totally got what you did with the Watch window.**

A: **A:** When you're debugging a program, you usually want to pay attention to how a few variables and fields change. That's where the Watch window comes in. If you add watches for a few variables, the Watch window updates their values every time you step into, out of, or over code. That lets you monitor exactly what happens to them after every statement, which can be really useful when you're trying to track down a problem.
The Watch window also lets you type in any statement you want, and even call methods, and the IDE will evaluate it and display the results. If the statement updates any of the fields and variables in your program, then it does that, too. That lets you change values while your program is running, which can be another really useful tool for reproducing exceptions and other bugs.

NOTE

Any changes you make in the Watch window just affect the data in memory, and last only as long as the program is running. Restart your program, and values that you changed will be undone.

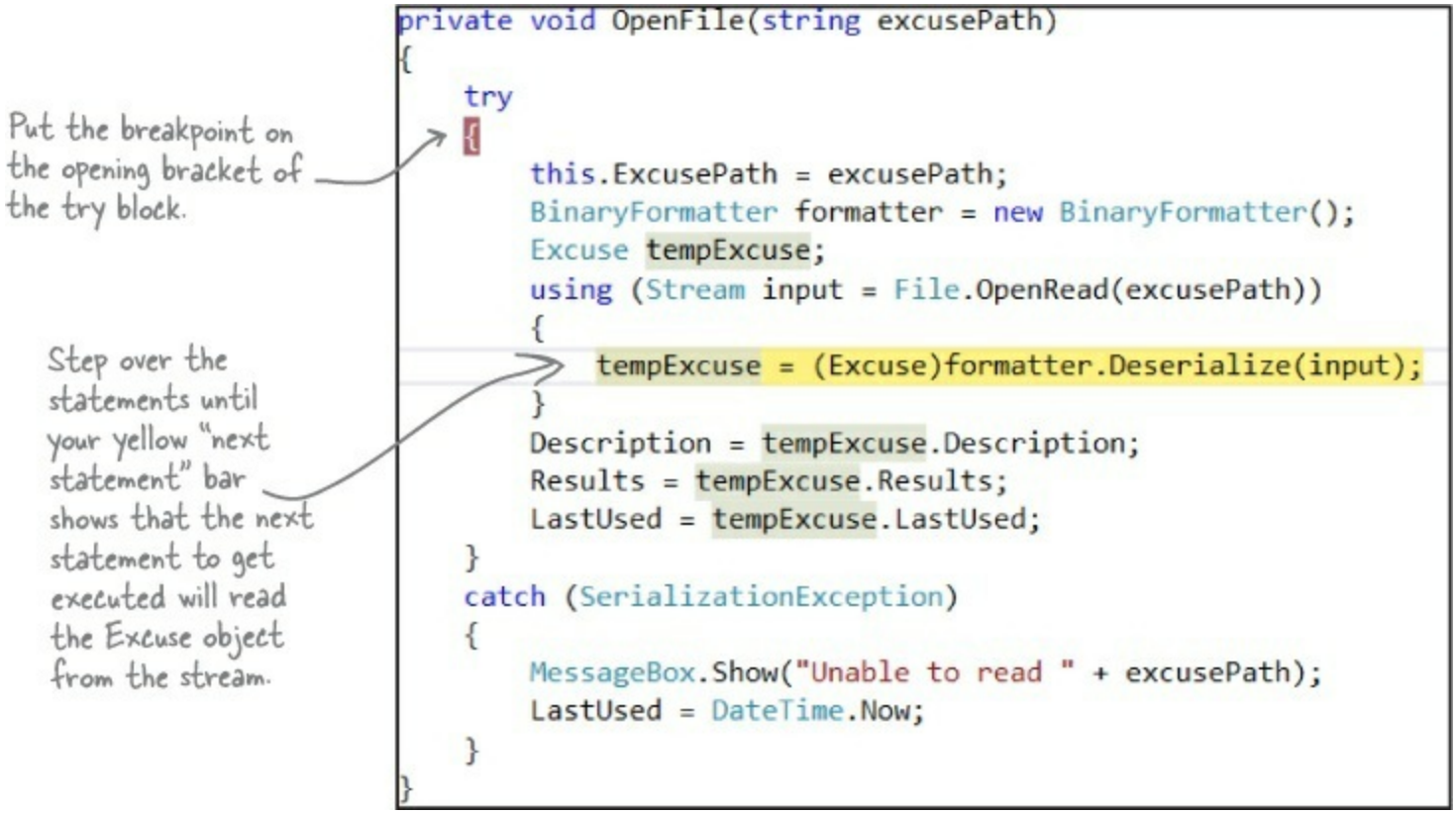
The catch block is executed only when code in the try block throws an exception. It gives you a chance to make sure your user has the information to fix the problem.

Use the debugger to follow the try/catch flow

An important part of exception handling is that when a statement in your `try` block throws an exception, the rest of the code in the block gets **short-circuited**. The program's execution immediately jumps to the first line in the `catch` block. ***But don't take our word for it...***

DEBUG THIS

- ❶ Add the `try/catch` from a few pages ago to your Excuse Manager app's `ReadExcuseAsync()` method. Then place a breakpoint on the opening bracket `{` in the `try` block.
- ❷ Start debugging your app and open up a file that's **not a valid excuse file** (but still has the *.excuse* extension). When the debugger breaks on your breakpoint, click the Step Over button (or F10) five times to get to the statement that calls `ReadObject()` to deserialize the `Excuse` object. Here's what your debugger screen should look like:



- ❸ Step over the next statement. As soon as the debugger executes the `Deserialize()` statement, the exception is thrown and the program **short-circuits** right past the rest of the method and **jumps straight to the `catch` block**.

The debugger will highlight the catch statement with its yellow "next statement" block, but it shows the rest of the block in gray to show you that it's about to execute the whole thing.

```
private void OpenFile(string excusePath)
{
    try
    {
        this.ExcusePath = excusePath;
        BinaryFormatter formatter = new BinaryFormatter();
        Excuse tempExcuse;
        using (Stream input = File.OpenRead(excusePath))
        {
            tempExcuse = (Excuse)formatter.Deserialize(input);
        }
        Description = tempExcuse.Description;
        Results = tempExcuse.Results;
        LastUsed = tempExcuse.LastUsed;
    }
    catch (SerializationException)
    {
        MessageBox.Show("Unable to read " + excusePath);
        LastUsed = DateTime.Now;
    }
}
```

- ❹ Start the program again by pressing the Continue button (or F5). It'll begin running the program again, starting with whatever's highlighted by the yellow "next statement" block — in this case, the `catch` block. It will just display the dialog and then act as if nothing happened. The ugly crash has now been handled.

WATCH IT!

Keep risky code out of the constructor!

You've noticed by now that a constructor doesn't have a return value, not even `void`. That's because a constructor doesn't actually return anything. Its only purpose is to initialize an object — which is a problem for exception handling inside the constructor. When an exception is thrown inside the constructor, then the statement that tried to instantiate the class ***won't end up with an instance of the object****.*

Here's a career tip: a lot of C# programming job interviews include a question about how you deal with exceptions in a constructor.

If you have code that should ALWAYS run, use a finally block

When your program throws an exception, a couple of things can happen. If the exception ***isn't*** handled, your program will stop processing and crash. If the exception ***is*** handled, your code jumps to the `catch` block. But what about the rest of the code in your `try` block? What if you were closing a stream, or cleaning up important resources? That code needs to run, even if an exception occurs, or you're going to make a mess of your program's state. That's where the **`finally`** block comes in really handy. It comes after the `try` and `catch` blocks. The **`finally` block always runs**, whether or not an exception was thrown.

```
private void OpenFile(string excusePath) {
    try {
        this.ExcusePath = excusePath;
        BinaryFormatter formatter = new BinaryFormatter();
        Excuse tempExcuse;
        using (Stream input = File.OpenRead(excusePath))
        {
            tempExcuse = (Excuse)formatter.Deserialize(input);
        }
        Description = tempExcuse.Description;
        Results = tempExcuse.Results;
        LastUsed = tempExcuse.LastUsed;
    }
    catch (SerializationException) {
        MessageBox.Show("Unable to read " + excusePath);
        LastUsed = DateTime.Now;
    }
    finally
    {
        // Any code here will get executed no matter what
    }
}
```

If there is no exception thrown during the try block, the code in the finally block will execute after the try block completes. If there's an exception handled by a catch block, then it will short-circuit as usual, and then run the finally block after the catch block.

Always catch specific exceptions like `SerializationException`. You typically follow a `catch` statement with a specific kind of exception telling it what to catch. It's valid C# code to just have **`catch (Exception)`** and you can even leave the exception type out and just use **`catch`**. When you do that, it **catches all exceptions**, no matter what type of exception is thrown. But it's a ***really bad practice to have a catch-all exception handler*** like that. Your code should always catch as specific an exception as possible.

Reminder: Once you finish Chapter 12, you can go straight through Chapter 13 in the book. It doesn't depend on Windows 8 or Windows Store apps at all.

Chapter 14

LINQ works with *any* kind of C# program.

When you read Chapter 14 in the main part of the book, you'll see that it's structured differently from other chapters. It has a series of increasingly complex LINQ queries, and small console apps to demonstrate each of them. Throughout the chapter, you'll also see exercises to build a Windows Store app that combines all the queries into a single user interface. Over the next few pages of this appendix, we'll show you how to build a WPF application that executes those same queries. Here's how we recommend you use this appendix with Chapter 14:

- Read through Jimmy could use some help in the book.
- Even though pages in the chapter through 665 are about building a Windows Store app, read them — especially the parts about **anonymous types**. It will help to get a sense of how the Comic, ComicQuery, and ComicQueryManager classes work.
- LINQ is versatile and Note describe more LINQ queries. You can skim Add the new queries to Jimmy's app and 669, because those are more Windows Store-related pages.
- Read Bullet Points–There are no Dumb Questions, but don't do the exercise in Bullet Points.
- You can skip the rest of the chapter, because it's related to Windows Store apps. Instead, follow the replacement There are no Dumb Questions–683.

Build a WPF comic query application

When you read through Chapter 14 in the book, you saw that we built a Windows Store app to execute the LINQ queries throughout the chapter. Since we followed the principle of separation of concerns, the classes for managing data and issuing queries were separated from the code that created the user interface. That let us **reuse the same data and query management classes** to build another app using the Visual Studio Split App template. Now we'll be able to take advantage of the same separation of concerns and build a WPF application using the same data and query classes.

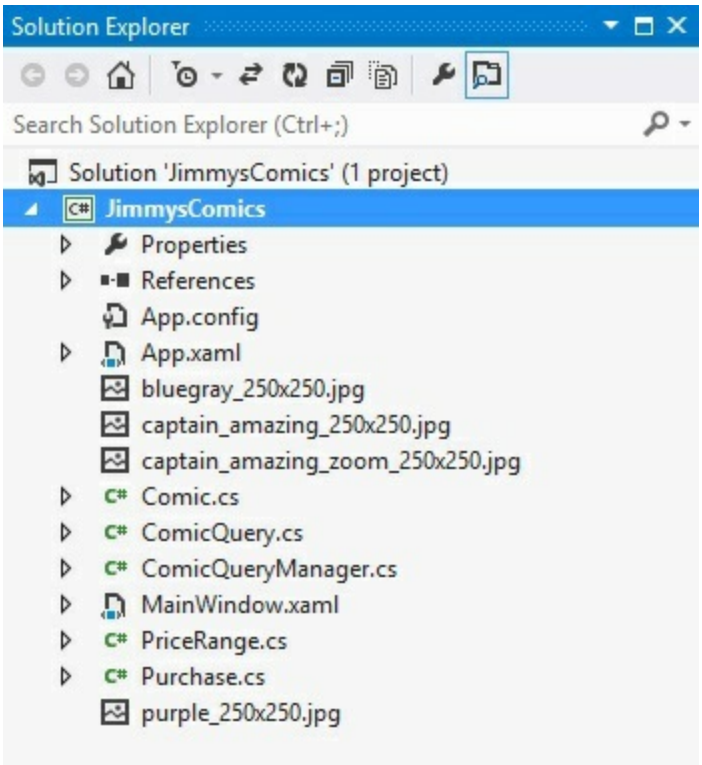
DO THIS!

- **❶ CREATE A NEW WPF APPLICATION AND ADD EXISTING CLASSES AND IMAGES FROM THE COMIC APP.**
  Before you start this project, you'll need to download source code to the JimmysComics app from Chapter 14. See the Head First Labs website (http://headfirstlabs.com/hfcsharp) for a link to the source code.
  Once you've got the source code, you'll build a new WPF application called JimmysComics. Then right-click on the project name in the Solution Explorer and choose "Add Existing Item" to add the following items from the Windows Store app we built in the book (you can download the source from the book's website):

  Purchase.cs
  Comic.cs
  ComicQuery.cs
  ComicQueryManager.cs
  PriceRange.cs.
  The following files are in the Assets folder: bluegray_250x250.jpg, bluegray_250x250.jpg, captain_ amazing_250x250.jpg, captain_amazing_zoom_250x250.jpg — add them to the root level of your WPF application so they're alongside your XAML and C# files.

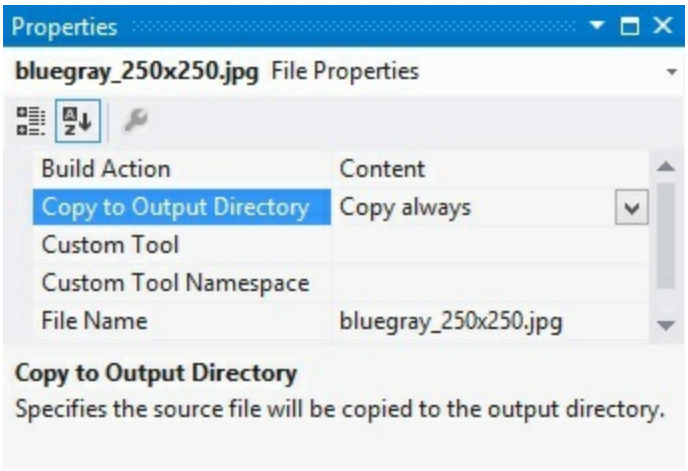
  > If you give your project a different name, make sure you change the namespace for the C# files you added to match your project's namespace.

  Your Solution Explorer should look like this:



  You'll also need to select each image file in the Solution Explorer and **use the Properties window** to set "Build Action" to

`Content` and "Copy to Output Directory" to `Copy always`. Here's what it looks like — make sure you do this for each of the .jpg files that you added:



- **❷ MAKE TWO MODIFICATIONS TO COMICQUERYMANAGER.CS.**
  There are two small changes you'll need to make to ComicQueryManager.cs. WPF applications cannot use the Windows.UI namespace because it's only part of the .NET Framework for Windows Store. You'll need to change the using statements at the top to replace "Windows.UI" with "System.Windows":

```
using System.Collections.ObjectModel;
using System.Windows.Media.Imaging;
```

And WPF applications load images slightly differently from Windows Store apps, so you'll need to change the CreateImageFromAssets() method in ComicQueryManager. Here's the new method:

```
private static BitmapImage CreateImageFromAssets(string imageFilename)
{
    try
    {
        Uri uri = new Uri(imageFilename, UriKind.RelativeOrAbsolute);
        return new BitmapImage(uri);
    }
    catch (System.IO.IOException)
    {
        return new BitmapImage();
    }
}
```

You copied the .jpg files into your project's top-level folder. This new CreateImageFromAssets() method will load those files.

- **❸ ADD CODE-BEHIND FOR THE MAIN WINDOW.**
  Here's all the code-behind you'll need for MainWindow.xaml.cs.

```
public partial class MainWindow : Window
{
    ComicQueryManager comicQueryManager;

    public MainWindow()
    {
        InitializeComponent();

        comicQueryManager = FindResource("comicQueryManager") as ComicQueryManager;
        comicQueryManager.UpdateQueryResults(comicQueryManager.AvailableQueries[0]);
    }

    private void ListView_SelectionChanged(object sender, SelectionChangedEventArgs e)
    {
        if (e.AddedItems.Count >= 1 && e.AddedItems[0] is ComicQuery)
        {
            comicQueryManager.CurrentQueryResults.Clear();
            comicQueryManager.UpdateQueryResults(e.AddedItems[0] as ComicQuery);
        }
    }
}
```

The ListView control fires its SelectionChanged event whenever the user selects or deselects items. The items that were selected can be found in the e.AddedItems collection.

- **❹ ADD THE XAML FOR THE MAIN WINDOW.**
  Here's the XAML for the main window. Remember, if you used a different project name, make sure you change `JimmysComics` to match your project's namespace.

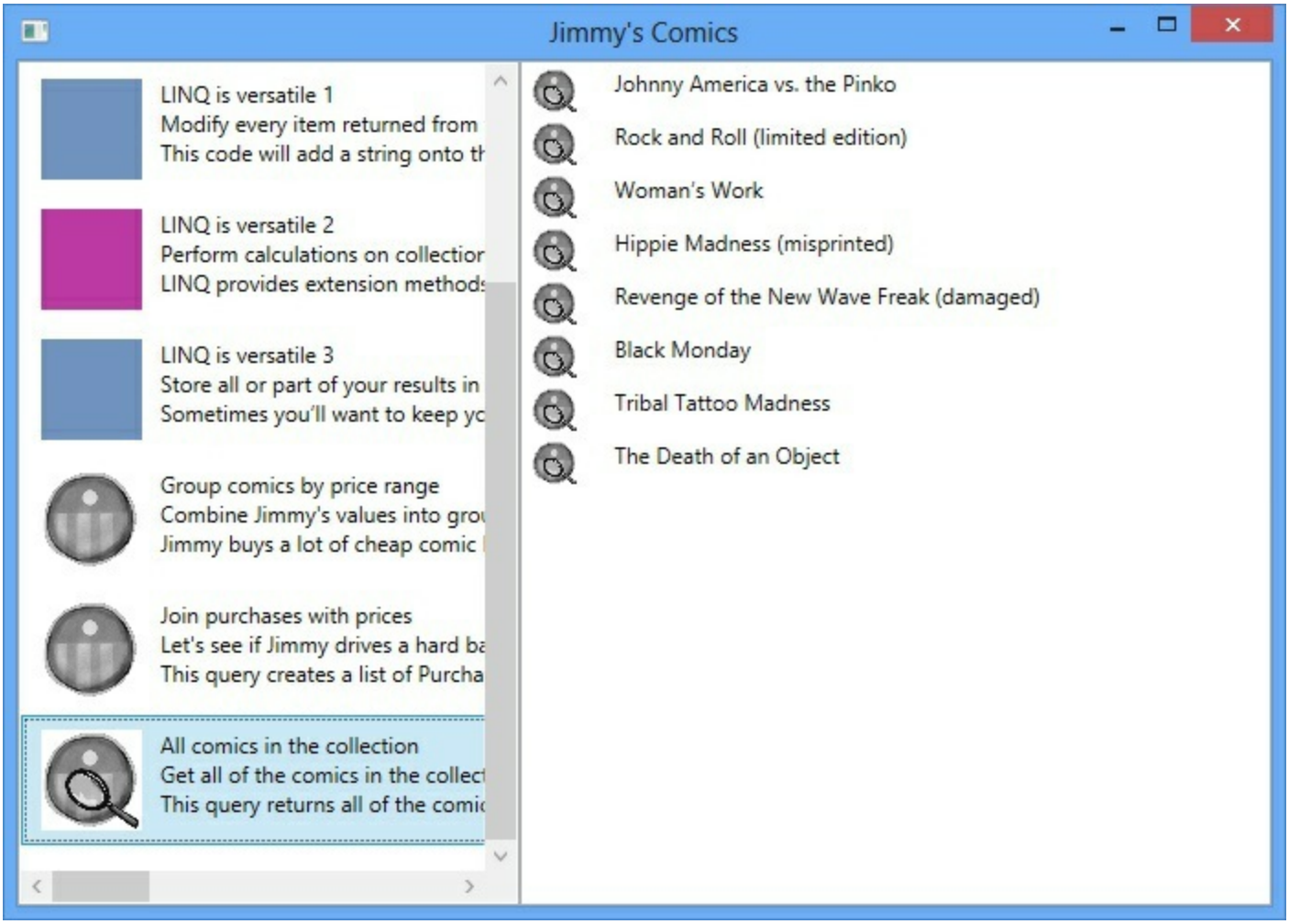
```
<Window x:Class="JimmysComics.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:JimmysComics"
        Title="Jimmy's Comics" Height="350" Width="525">

    <Window.Resources>
        <local:ComicQueryManager x:Key="comicQueryManager"/>
    </Window.Resources>

    <Grid DataContext="{StaticResource ResourceKey=comicQueryManager}">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="2*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <ListView SelectionMode="Single" ItemsSource="{Binding AvailableQueries}"
                  SelectionChanged="ListView_SelectionChanged">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <Grid Height="55" Margin="6">
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="Auto"/>
                            <ColumnDefinition Width="*"/>
                        </Grid.ColumnDefinitions>
                        <Border Width="55" Height="55">
                            <Image Source="{Binding Image}" Stretch="UniformToFill"/>
                        </Border>
                        <StackPanel Grid.Column="1" VerticalAlignment="Top" Margin="10,0,0,0">
                            <TextBlock Text="{Binding Title}" TextWrapping="NoWrap"/>
                            <TextBlock Text="{Binding Subtitle}" TextWrapping="NoWrap"/>
                            <TextBlock Text="{Binding Description}" TextWrapping="NoWrap"/>
                        </StackPanel>
                    </Grid>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>

        <ListView Grid.Column="1" SelectionMode="Single"
                  ItemsSource="{Binding CurrentQueryResults}">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                        <Image Source="{Binding Image}" Margin="0,0,20,0"
                               Stretch="UniformToFill" Width="25" Height="25"
                               VerticalAlignment="Top" HorizontalAlignment="Right"/>
                        <StackPanel>
                            <TextBlock Text="{Binding Title}" />
                        </StackPanel>
                    </StackPanel>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>

    </Grid>
</Window>
```

This ListView's SelectionMode is set to Single so only one query can be selected at a time.

The ListView on the right has an item template that displays information about each query.

The ListView on the right has an item template that shows individual items in the query results.

When you run the app, the queries appear on the left, and the results of the selected query appear on the right.



QUERIES THAT RETURN COMIC BOOKS HAVE ADDITIONAL INFORMATION: PRICE, SYNOPSIS, EVEN A COVER IMAGE. CAN YOU FIGURE OUT HOW TO GET THE COMIC QUERIES TO DISPLAY ALL THE INFORMATION ABOUT EACH COMIC? YOU'LL NEED TO ADD THE COMIC BOOK COVER IMAGES TO THE PROJECT. YOU'LL FIND SOME HELPFUL XAML CODE IN THE CHAPTER ON PAGES 689 AND 690.

NOTE

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Chapter 15

Events are useful for any app, but especially important for understanding XAML.

Events can be simple and straightforward, because you've been using them throughout the book. But there's a lot more depth to them than you might expect. This chapter helps you understand events in more detail.

Here's what we recommend for this chapter:

- Read the chapter in the book through page 711.
- Use the replacement pages in this appendix for the exercise in Exercise–713 and its solution in Exercise Solution–Note.
- Read Generic EventHandlers let you define your own event types–719 in the book.
- Windows Store apps use events for process lifetime management–723 are specific to Windows Store apps, but **we recommend that you read them anyway**. They give you some insight not just into Windows Store apps, but also into some basic features of Windows 8.
- We provide replacement pages for XAML controls use routed events-729 in this appendix.
- Read the rest of the chapter in the book. The only pages you should skip are the top of MessageDialog uses the callback pattern, and Use delegates to use the Windows settings charm–743.

EXERCISE

It's time to put what you've learned so far into practice. Your job is to complete the `Ball` and `Pitcher` classes, add a `Fan` class, and make sure they all work together with a very basic version of your baseball simulator.

- **❶ COMPLETE THE PITCHER CLASS.**
  Below is what we've got for `Pitcher`. Add the `CatchBall()` and `CoverFirstBase()` methods. Both should create a string saying that the catcher has either caught the ball or run to first base and add that string to a public `ObservableCollection<string>` called `PitcherSays`.

```
class Pitcher {
    public Pitcher(Ball ball) {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e) {
        if (e is BallEventArgs){
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
                CatchBall();
            else
                CoverFirstBase();
        }
    }
}
```

You'll need to implement these two methods to add a string to the PitcherSays ObservableCollection.

- **❷ WRITE A FAN CLASS.**
  Create another class called `Fan`. `Fan` should also subscribe to the `BallInPlay` event in its constructor. The fan's event handler should see if the distance is greater than 400 feet and the trajectory is greater than 30 (a home run), and grab for a glove to try to catch the ball if it is. If not, the fan should scream and yell. Everything that the fan screams and yells should be added to an `ObservableCollection<string>` called `FanSays`.

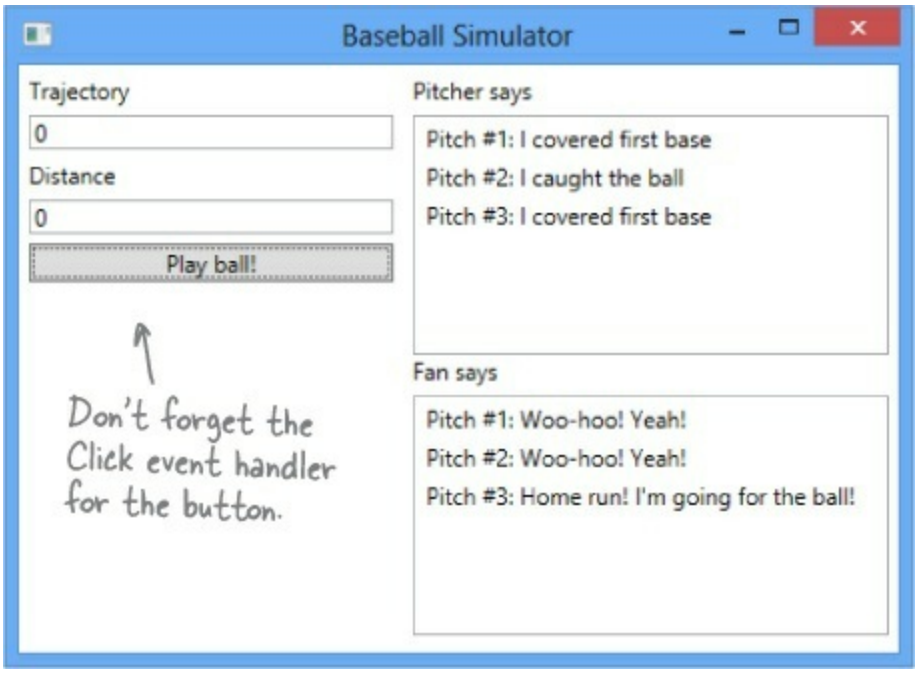
> **NOTE**
>
> Look at the output on the facing page to see exactly what it should print.

- **❸ BUILD A VERY SIMPLE SIMULATOR.**
  If you didn't do it already, create a new WPF Application and add the following `BaseballSimulator` class. Then add it as a static resource to the page.

```
using System.Collections.ObjectModel;

class BaseballSimulator {
    private Ball ball = new Ball();
    private Pitcher pitcher;
    private Fan fan;
    public ObservableCollection<string> FanSays { get { return fan.FanSays; } }
    public ObservableCollection<string> PitcherSays { get { return pitcher.PitcherSays; } }
```

```
        public int Trajectory { get; set; }
        public int Distance { get; set; }
        public BaseballSimulator() {
            pitcher = new Pitcher(ball);
            fan = new Fan(ball);
        }
        public void PlayBall() {
            BallEventArgs ballEventArgs = new BallEventArgs(Trajectory, Distance);
            ball.OnBallInPlay(ballEventArgs);
        }
    }
```

- **❹ BUILD THE MAIN WINDOW.**
  Can you come up with the XAML just from looking at the screenshot to the right? The two TextBox controls are bound to the `Trajectory` and `Distance` properties of the `BaseballSimulator` static resource, and the pitcher and fan chatter are ListView controls bound to the two `ObservableCollections`.
  See if you can make your simulator generate the above fan and pitcher chatter with three successive balls put into play. Write down the values you used to get the result below:



Ball 1:	**Ball 2:**	**Ball 3:**
Trajectory: _____	Trajectory: _____	Trajectory: _____
Distance: _____	Distance: _____	Distance: _____

EXERCISE SOLUTION

Here are the `Ball` and `BallEventArgs` from earlier, and the new `Fan` class that needed to be added:

```
class Ball {
    public event EventHandler BallInPlay;
    public void OnBallInPlay(BallEventArgs e) {
        EventHandler ballInPlay = BallInPlay;
        if (ballInPlay != null)
            ballInPlay(this, e);
    }
}
```

The OnBallInPlay() method just raises the BallInPlay event—but it has to check to make sure it's not null; otherwise, it'll throw an exception.

Read-only automatic properties work really well in event arguments because the event handlers read only the data passed to them.

```
class BallEventArgs : EventArgs {
    public int Trajectory { get; private set; }
    public int Distance { get; private set; }
    public BallEventArgs(int trajectory, int distance)
    {
        this.Trajectory = trajectory;
        this.Distance = distance;
    }
}

using System.Collections.ObjectModel;
class Fan {
    public ObservableCollection<string> FanSays = new ObservableCollection<string>();
    private int pitchNumber = 0;

    public Fan(Ball ball) {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e)  {
        pitchNumber++;
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if (ballEventArgs.Distance > 400 && ballEventArgs.Trajectory > 30)
                FanSays.Add("Pitch #" + pitchNumber
                            + ": Home run! I'm going for the ball!");
            else
                FanSays.Add("Pitch #" + pitchNumber + ": Woo-hoo! Yeah!");
        }
    }
}
```

The Fan object's constructor chains its event handler onto the BallInPlay event.

The fan's BallInPlay event handler looks for any ball that's high and long.

Here's the code-behind for the page:

```
public partial class MainWindow : Window {
    BaseballSimulator baseballSimulator;

    public MainWindow() {
        InitializeComponent();

        baseballSimulator = FindResource("baseballSimulator") as BaseballSimulator;
    }

    private void Button_Click(object sender, RoutedEventArgs e) {
        baseballSimulator.PlayBall();
    }
}
```

Here's the XAML for the page. It also needs: `<local:BaseballSimulator x:Key="baseballSimulator"/>`

```
<Window.Resources>
        <local:BaseballSimulator x:Key="baseballSimulator"/>
</Window.Resources>
```

Make sure you also add the xmlns:local property to the <Window> tag.

```
<Grid Margin="5" DataContext="{StaticResource ResourceKey=baseballSimulator}">
  <Grid.ColumnDefinitions>
      <ColumnDefinition Width="200" />
      <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <StackPanel Margin="0,0,10,0">
    <TextBlock Text="Trajectory" Margin="0,0,0,5"/>
    <TextBox Text="{Binding Trajectory, Mode=TwoWay}"  Margin="0,0,0,5"/>
    <TextBlock Text="Distance" Margin="0,0,0,5"/>
    <TextBox Text="{Binding Distance, Mode=TwoWay}" Margin="0,0,0,5"/>
    <Button Content="Play ball!" Click="Button_Click"/>
  </StackPanel>
  <StackPanel Grid.Column="1">
    <TextBlock Text="Pitcher says" Margin="0,0,0,5"/>
    <ListView ItemsSource="{Binding PitcherSays}" Height="125"/>
    <TextBlock Text="Fan says" Margin="0,0,0,5"/>
    <ListView ItemsSource="{Binding FanSays}" Height="125"/>
  </StackPanel>
</Grid>
```

And here's the `Pitcher` class (it needs `using System.Collections.ObjectModel;` at the top):

```
class Pitcher {
    public ObservableCollection<string> PitcherSays = new ObservableCollection<string>();
    private int pitchNumber = 0;

    public Pitcher(Ball ball)     {
        ball.BallInPlay += ball_BallInPlay;
    }

    void ball_BallInPlay(object sender, EventArgs e) {
        pitchNumber++;
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
                CatchBall();
            else
                CoverFirstBase();
        }
    }
    private void CatchBall() {
        PitcherSays.Add("Pitch #" + pitchNumber + ": I caught the ball");
    }
    private void CoverFirstBase() {
        PitcherSays.Add("Pitch #" + pitchNumber + ": I covered first base");
    }
}
```

We gave you the pitcher's BallInPlay event handler. It looks for any low balls.

Here are the values we used to get the output. Yours might be a little different.

Ball 1:	**Ball 2:**	**Ball 3:**
Trajectory: 75	Trajectory: 48	Trajectory: 40
Distance: 105	Distance: 80	Distance: 435

XAML controls use routed events

Flip to Note in the main part of the book and have a closer look at the IntelliSense window that pops up when you type `override` into the IDE. Yes, it's for a Windows Store app, but the *same exact principle* applies to WPF. Two of the names of the event argument types are a little different from the others. The `DoubleTapped` event's second argument has the type `DoubleTappedRoutedEventArgs`, and the `GotFocus` event's is a `RoutedEventArgs`. The reason is that the `DoubleTapped` and `GotFocus` events are **routed events**. These are like normal events, except for one difference: when a control object responds to a routed event, first it fires off the event handler method as usual. Then it does something else: if the event hasn't been handled, it **sends the routed event up to its container**. The container fires the event, and then if it isn't handled, it sends the routed event up to its container. The event keeps **bubbling up** until it's either handled or it hits the **root**, or the container at the very top. Here's a typical routed event handler method signature.

```
private void EventHandler(object sender, RoutedEventArgs e)
```

The `RoutedEventArgs` object has a property called **`Handled`** that the event handler can use to indicate that it's handled the event. Setting this property to `true` **stops the event from bubbling up**.

In both routed and standard events, the `sender` parameter always contains a reference to the object that called the event handler. So if an event is bubbled up from a control to a container like a Grid, then when the Grid calls its event handler, `sender` will be a reference to the Grid control. But what if you want to find out which control fired the original event? No problem. The `RoutedEventArgs` object has a property called **`OriginalSource`** that contains a reference to the control that initially fired the event. If `OriginalSource` and `sender` point to the same object, then the control that called the event handler is the same control that originated the event and started it bubbling up.

IsHitTestVisible determines if an element is “visible” to the pointer or mouse

Typically, any element on the page can be “hit” by the pointer or mouse — as long as it meets certain criteria. It needs to be visible (which you can change with the `Visibility` property), it has to have a `Background` or `Fill` property that’s not null (but can be `Transparent`), it must be enabled (with the `IsEnabled` property), and it has to have a `height` and `width` greater than zero. If all of these things are true, then the **`IsHitTestVisible` property will return `True`**, and that will cause it to respond to pointer or mouse events.

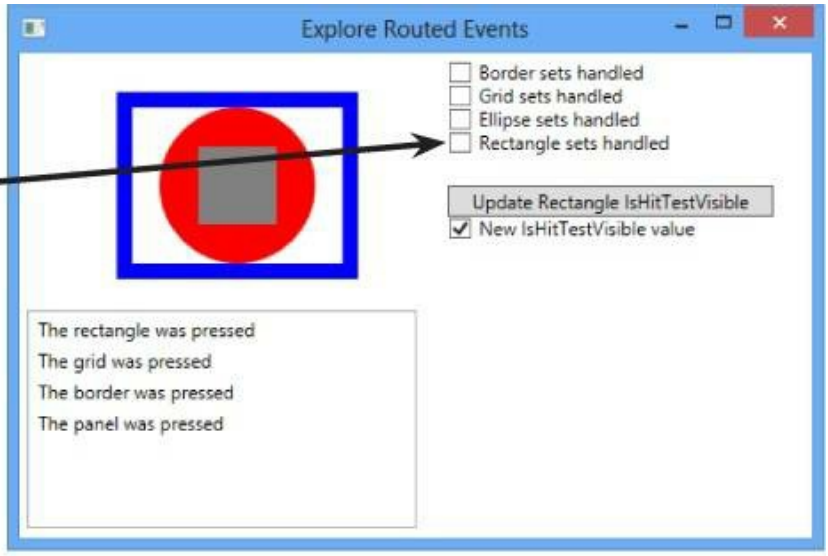
This property is especially useful if you want to make your events “invisible” to the mouse. If you set `IsHitTestVisible` to `False`, then any pointer taps or mouse clicks will **pass right through the control**. If there’s another control below it, that control will get the event instead.

> **The structure of controls that contain other controls that in turn contain yet more controls is called an object tree, and routed events bubble up the tree from child to parent until they hit the root element at the top.**

You can see a list of input events that are routed events here: http://msdn.microsoft.com/en-us/library/windows/apps/Hh758286.aspx

Create an app to explore routed events

Here's a WPF application that you can use to experiment with routed events. It's got a StackPanel that contains a Border, which contains a Grid, and inside that grid are an Ellipse and a Rectangle. Have a look at the screenshot. See how the Rectangle is on top of the Ellipse? If you put two controls into the same cell, they'll stack on top of each other. But both of those controls have the same parent: the Grid, whose parent is the Border, and the Border's parent is the StackPanel. Routed events from the Rectangle or Ellipse bubble up through the parents to the root of the **object tree**.



You've already seen the **CheckBox** control, which you can use to toggle a value on and off. The `Content` property sets the label for the control. The `IsChecked` property is a **Nullable<bool>** because in addition to on and off, it can also have a third indeterminate state

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <StackPanel x:Name="panel" MouseDown="StackPanel_MouseDown">
        <Border BorderThickness="10" BorderBrush="Blue" Width="155" x:Name="border"
            Margin="20" MouseDown="Border_MouseDown">
            <Grid x:Name="grid" MouseDown="Grid_MouseDown">
                <Ellipse Fill="Red"  Width="100" Height="100"
                    MouseDown="Ellipse_MouseDown"/>
                <Rectangle Fill="Gray" Width="50" Height="50"
                    MouseDown="Rectangle_MouseDown" x:Name="grayRectangle"/>
            </Grid>
        </Border>
        <ListBox BorderThickness="1" Width="250" Height="140" x:Name="output" Margin="0,0,20,0"/>
    </StackPanel>
    <StackPanel Grid.Column="1">
        <CheckBox Content="Border sets handled" x:Name="borderSetsHandled"/>
        <CheckBox Content="Grid sets handled" x:Name="gridSetsHandled" />
        <CheckBox Content="Ellipse sets handled" x:Name="ellipseSetsHandled"/>
        <CheckBox Content="Rectangle sets handled" x:Name="rectangleSetsHandled"/>
        <Button Content="Update Rectangle IsHitTestVisible"
            Click="UpdateHitTestButton" Margin="0,20,20,0"/>
        <CheckBox IsChecked="True" Content="New IsHitTestVisible value"
            x:Name="newHitTestVisibleValue" />
    </StackPanel>
</Grid>
```

Routed events bubble up the object tree.

IsChecked defaults to False. This CheckBox has it set to True because controls always have IsHitTestVisible set to true by default.

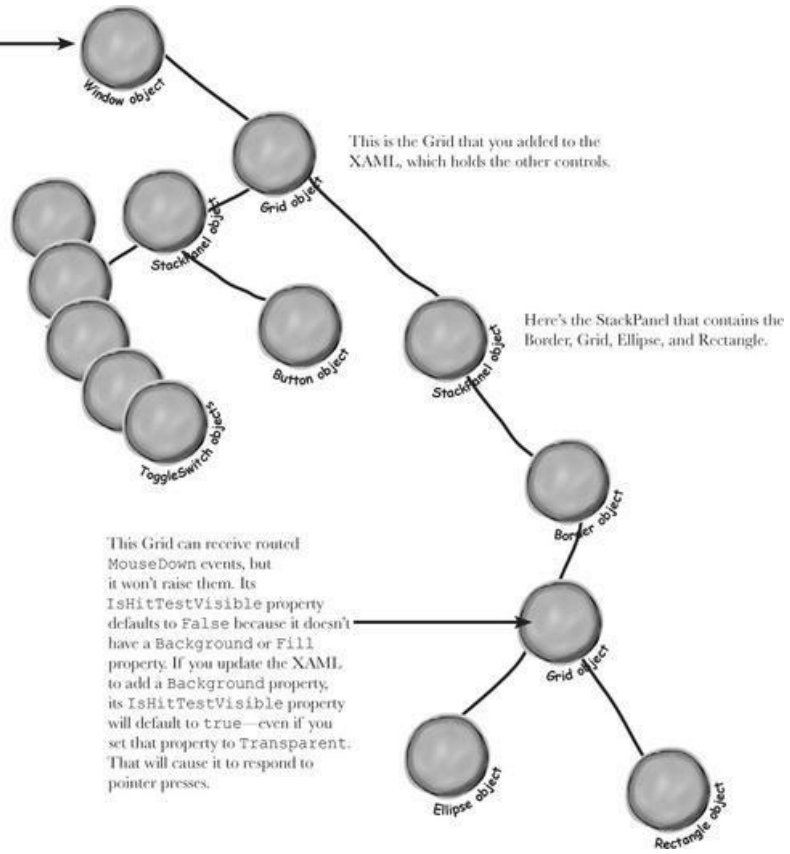
Flip the page to finish the app →

YOU'LL NEED THIS OBSERVABLECOLLECTION TO DISPLAY OUTPUT IN THE LISTBOX.

Make a field called `outputItems` and set the `ListBox.ItemsSource` property in the page

constructor. And don't forget to add the `using System.Collections.ObjectModel;` statement for `ObservableCollection<T>`.

```
public partial class MainWindow : Window {
    ObservableCollection<string> outputItems = new ObservableCollection<string>();

    public MainWindow() {
        this.InitializeComponent();

        output.ItemsSource = outputItems;
    }
```



Here's the code-behind. Each control's `MouseDown` event handler clears the output if it's the original source, and then it adds a string to the output. If its "handled" toggle switch is on, it uses `e.Handled` to handle the event.

```
private void Ellipse_MouseDown(object sender, MouseButtonEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The ellipse was pressed");
    if (ellipseSetsHandled.IsChecked == true) e.Handled = true;
}

private void Rectangle_MouseDown(object sender, MouseButtonEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The rectangle was pressed");
    if (rectangleSetsHandled.IsChecked == true) e.Handled = true;
}

private void Grid_MouseDown(object sender, MouseButtonEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The grid was pressed");
    if (gridSetsHandled.IsChecked == true) e.Handled = true;
}

private void Border_MouseDown(object sender, MouseButtonEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The border was pressed");
    if (borderSetsHandled.IsChecked == true) e.Handled = true;
}

private void StackPanel_MouseDown(object sender, MouseButtonEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The panel was pressed");
}

private void UpdateHitTestButton(object sender, RoutedEventArgs e) {
    grayRectangle.IsHitTestVisible = (bool)newHitTestVisibleValue.IsChecked;
}
```

The Click event handler for the button uses the IsOn property of the toggle switch to turn IsHitTestVisible on or off for the Rectangle control.

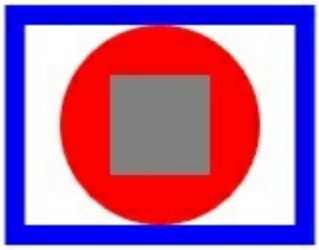
HERE'S THE OBJECT GRAPH FOR YOUR MAIN WINDOW.

The `Mainwindow` class is at the root of the object tree. When you create the new WPF application, the *MainWindow.xaml* and *MainWindow.xaml.cs* files create an object that **extends the `Window` class**.

Flip the page to use your new app to explore routed events

RUN THE APP AND CLICK OR TAP THE GRAY RECTANGLE.

You should see the output in the screenshot to the right.



The rectangle was pressed
The grid was pressed
The border was pressed
The panel was pressed

You can see exactly what's going on by putting a breakpoint on the first line of `Rectangle_MouseDown()`, the Rectangle control's `MouseDown` event handler:

```
private void Rectangle_MouseDown(object sender, MouseButtonEventArgs e)
{
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The rectangle was pressed");
    if (rectangleSetsHandled.IsChecked == true) e.Handled = true;
}
```

Click the gray rectangle again — this time the breakpoint should fire. Use Step Over (F10) to **step through the code line by line**. First you'll see the `if` block execute to clear the `outputItems` `ObservableCollection` that's bound to the ListBox. This happens because `sender` and `e.OriginalSource` reference the same Rectangle control, which is true only inside the event handler method for the control that originated the event (in this case, the control that you clicked or tapped), so `sender == e.OriginalSource` is true.

When you get to the end of the method, **keep stepping through the program**. The event will bubble up through the object tree, first running the Rectangle's event handler, then the Grid's event handler, then the Border's, then the Panel's, and finally it runs an event handler method that's part of `LayoutAwarePage` — this is outside of your code and not part of the routed event, so it will always run. Since none of those controls are the original source for the event, none of their senders will be the same as `e.OriginalSource`, so none of them clear the output.

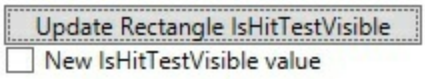
TURN ISHITTESTVISIBLE OFF, PRESS THE "UPDATE" BUTTON, AND THEN CLICK OR TAP THE RECTANGLE.

You should see this output.

The ellipse was pressed
The grid was pressed
The border was pressed
The panel was pressed

Wait a minute! You pressed the Rectangle, but the Ellipse control's `MouseDown` event handler fired.

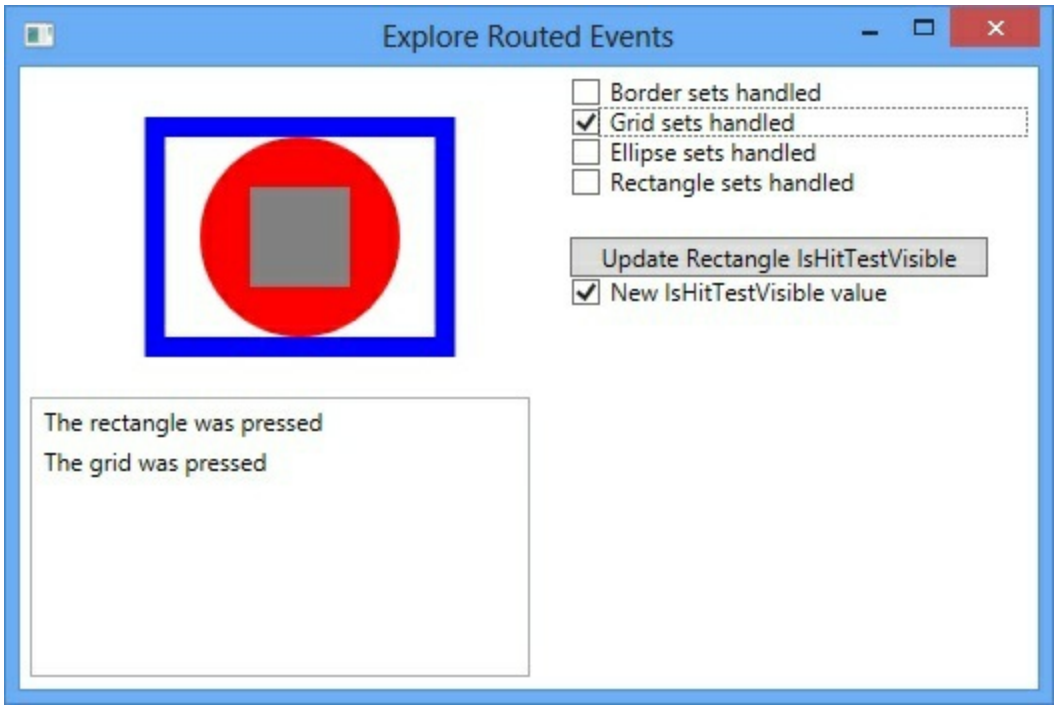
What's going on?



When you pressed the button, its `Click` event handler updated the Rectangle control's `IsHitTestVisible` property to `false`, which made it "invisible" to pointer presses, clicks, and other pointer events. So when you tapped the Rectangle, your tap passed right through it to the topmost control underneath it on the page that has `IsHitTestVisible` set to `true` and has a `Background` property that's set to a color or `Transparent`. In this case, it finds the Ellipse control and fires its `MouseDown` event.

CHECK THE "GRID SETS HANDLED" BOX AND CLICK OR TAP THE GRAY RECTANGLE.

You should see this output.



So why did only two lines get added to the output ListBox? **Step through the code again** to see what's going on. This time, `gridSetsHandled.IsOn` was true because you toggled the `gridSetsHandled` to `On`, so the last line in **the Grid's event handler set `e.IsHandled` to `true`**. As soon as a routed event handler method does that, **the event stops bubbling up**. As soon as the Grid's event handler completes, the app sees that the event has been handled, so it doesn't call the Border or Panel's event handler method, and instead skips to the event handler method in `LayoutAwarePage` that's outside of the code you added.

USE THE APP TO EXPERIMENT WITH ROUTED EVENTS.

Here are a few things to try:

- Click on the gray Rectangle and the red Ellipse and watch the output to see how the events bubble up.
- Turn on each of the toggle switches, starting at the top, to cause the event handlers to set `e.Handled` to `true`. Watch the events stop bubbling when they're handled.
- Set breakpoints and debug through all of the event handler methods.

- Try setting a breakpoint in the Ellipse's event handler method, and then turn the gray Rectangle's `IsHitTestVisible` property on and off by toggling the bottom switch and pressing the button. Step through the code for the Rectangle when `IsHitTestVisible` is set to `false`.
- Stop the program and add a `Background` property to the Grid to make it visible to pointer hits.

A routed event first fires the event handler for the control that originated the event, and then bubbles up through the control hierarchy until it hits the top — or an event handler sets e.Handled to true.

NOTE

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Chapter 16

Great developers follow design patterns.

In this chapter, you'll learn about Model-View-ViewModel (MVVM), a design pattern for building effective WPF apps. Along the way, you'll learn what a design pattern is, and you'll learn how to use XAML controls to create great animations.

Here's how we recommend that you work through Chapter 16:

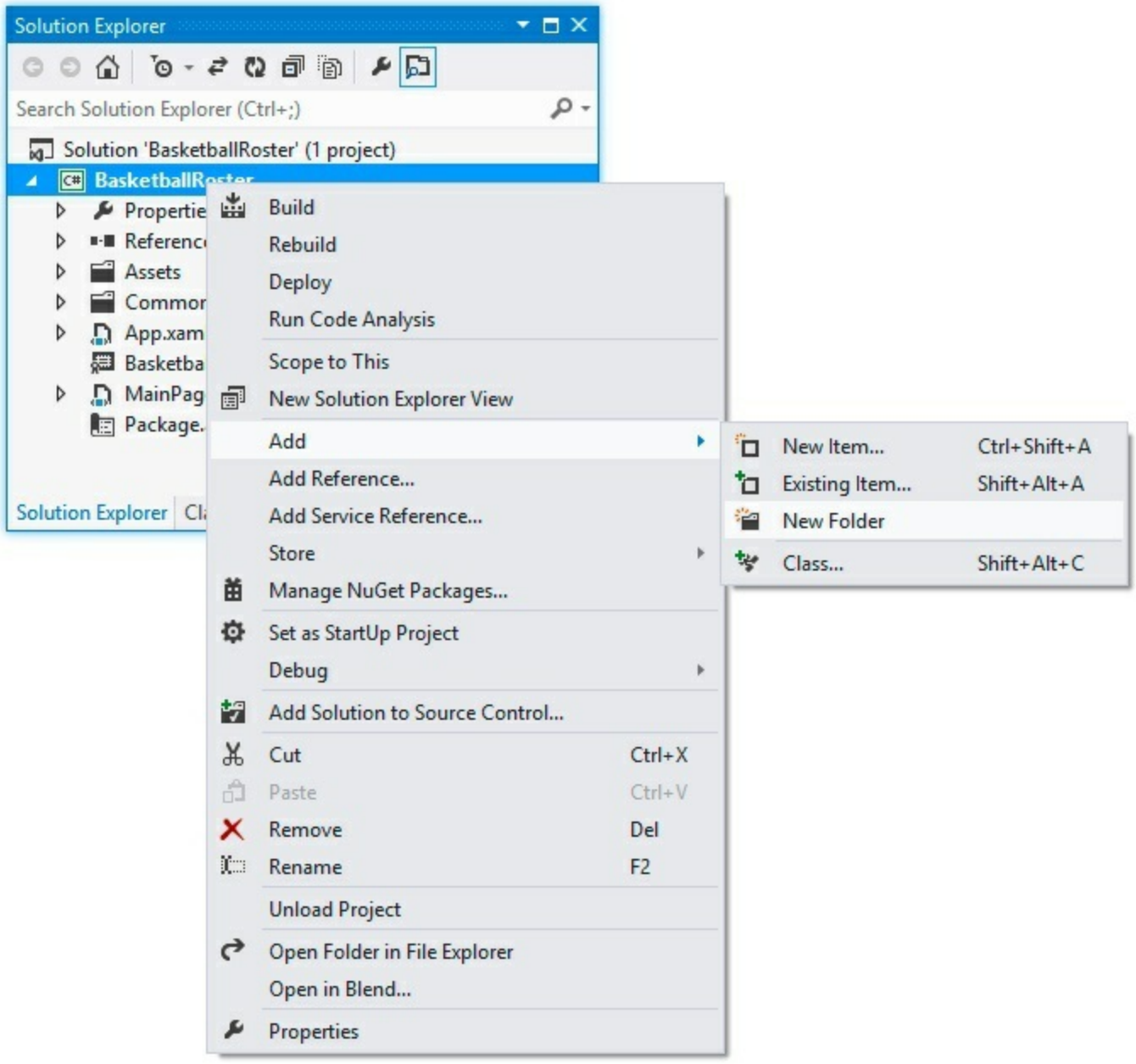
- Read through MVVM lets you design for binding and data.
- Follow our replacement pages for 750–757.
- Read Note–Events alert the rest of the app to state changes.
- Start the Stopwatch project in MVVM means thinking about the state of the app in the book, and continue it using a combination of book pages and appendix replacement Build the view for a simple stopwatch, Finish the stopwatch app, Converters automatically convert values for binding–773, and Build an analog stopwatch using the same ViewModel–787.
- Read C# can build "real" animations, too in the book.
- The rest of Chapter 16 is replaced with Create a user control to animate a picture–Part III in this appendix.
- There's information in Congratulations! (But you're not done yet...) about how to do Lab #3.

Use the MVVM pattern to start building the basketball roster app

Create a new WPF application and **make sure it's called BasketballRoster** (because we'll be using the namespace `BasketballRoster` in the code, and this will make sure your code matches what's on the next few pages).

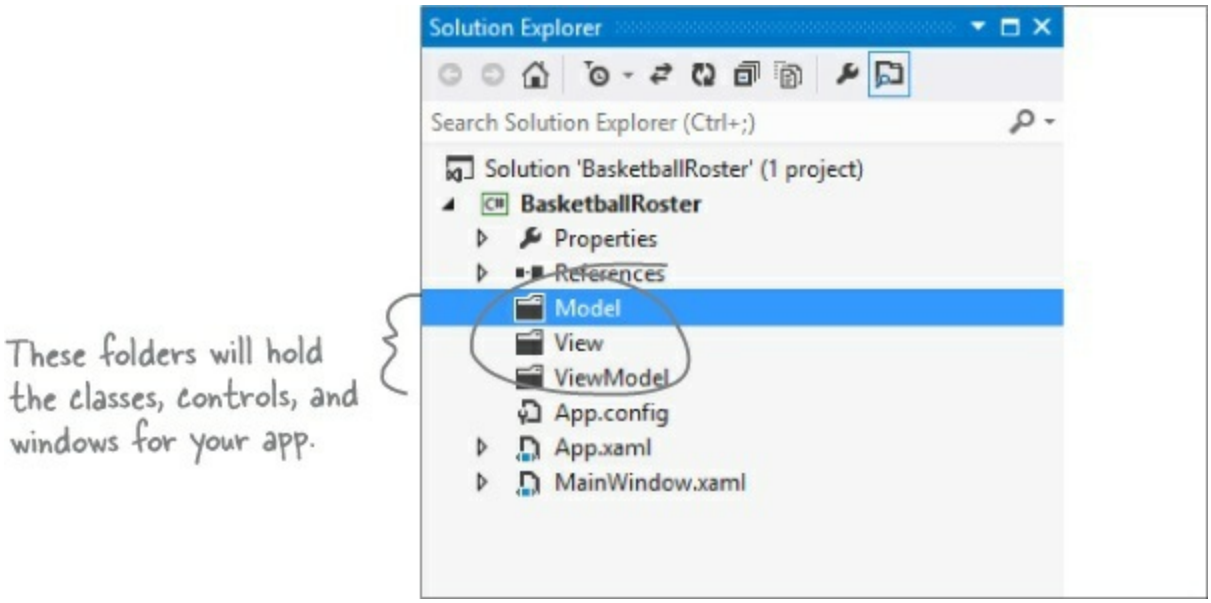
DO THIS

- **❶ CREATE THE MODEL, VIEW, AND VIEWMODEL FOLDERS IN THE PROJECT.**
  Right-click on the project in the Solution Explorer and choose New Folder from the Add menu:



When you use the Solution Explorer to add a new folder to your project, the IDE creates a new namespace based on the folder name. This causes the Add→Class... menu option to create classes with that namespace. So if you add a class to the Model folder, the IDE will add `BasketballRoster.Model` to the `namespace` line at the top of the class file.

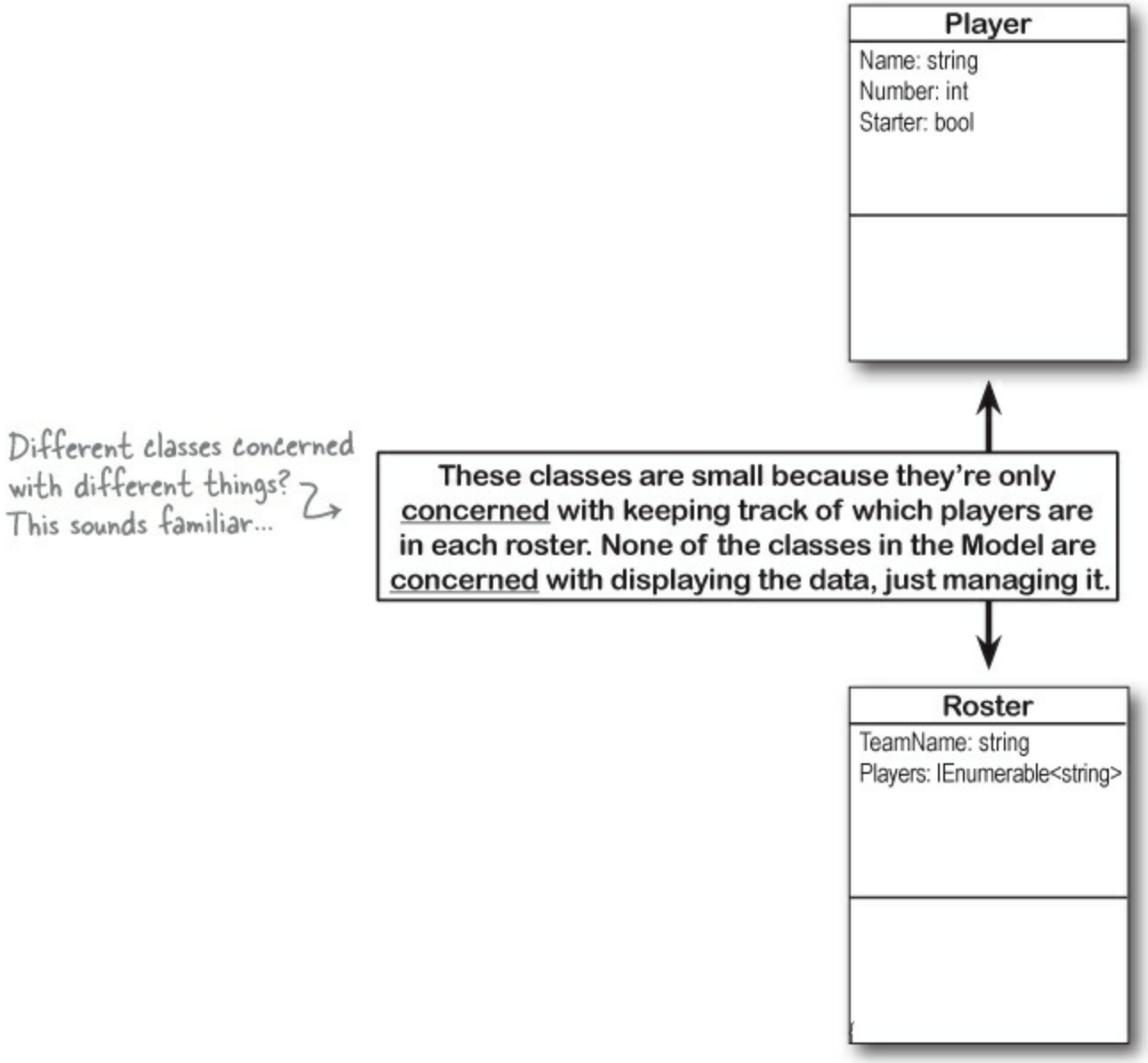
Add a *Model* folder. Then do it two more times to add the *View* and *ViewModel* folders, so your project looks like this:



- **❷ START BUILDING THE MODEL BY ADDING THE PLAYER CLASS.**
  Right-click on the *Model* folder and **add a class called `Player`**. When you add a class into a folder, the IDE updates the namespace to add the folder name to the end. Here's the `Player` class:

When you add a class file into a folder, the IDE adds the folder name to the namespace.

```
namespace BasketballRoster.Model {
    class Player {
        public string Name { get; private set; }
        public int Number { get;  private set; }
        public bool Starter { get; private set; }

        public Player(string name, int number, bool starter) {
            Name = name;
            Number = number;
            Starter = starter;
        }
    }
}
```



- **❸ FINISH THE MODEL BY ADDING THE ROSTER CLASS**
  Next, **add the `Roster` class to the *Model* folder**. Here's the code for it.

```
namespace BasketballRoster.Model {
    class Roster {
        public string TeamName { get; private set; }

        private readonly List<Player> _players = new List<Player>();
        public IEnumerable<Player> Players {
            get { return new List<Player>(_players); }
        }

        public Roster(string teamName, IEnumerable<Player> players)
            TeamName = teamName;
            _players.AddRange(players);
        }
    }
}
```

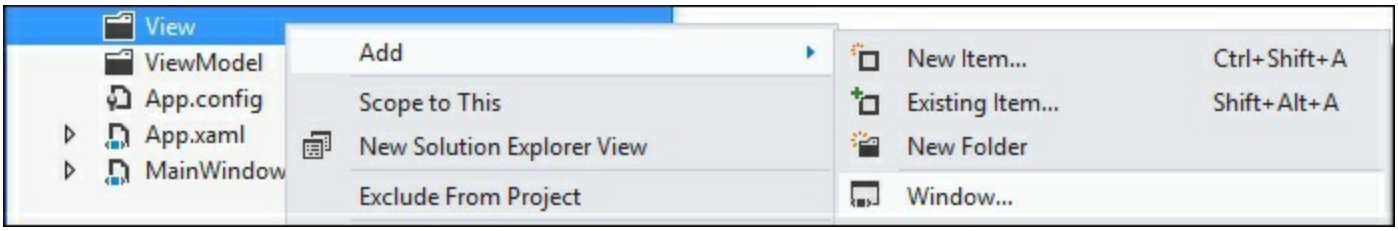
The _ tells you that this field is private.

We added an underscore to the beginning of the name of the `_players` field. Adding an underscore to the beginning of private fields is a very common naming convention. We're going to use it throughout this chapter so you can get used to seeing it.

Your *Model* folder should now look like this:

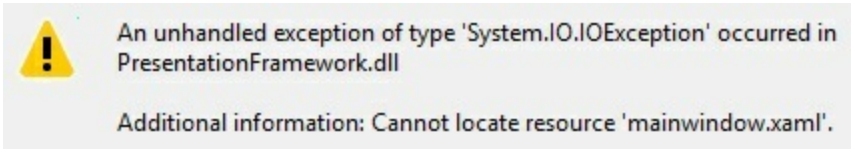
We'll add the view on the next page →

- **❹ ADD A NEW MAIN WINDOW TO THE VIEW FOLDER.**
  Right-click on the *View* folder and **add a new Window called** *LeagueWindow.xaml*.



Your project's View folder should now have a XAML window in it called LeagueWindow.xaml. This is just like the MainWindow.xaml window that you've been working with throughout the book. It's still a Window object with a graph that's defined with XAML. The only difference is that it's called LeagueWindow instead of MainWindow.

- **❺ DELETE THE MAIN WINDOW AND REPLACE IT WITH YOUR NEW WINDOW.**
  Delete the *MainWindow.xaml* file from the project by **right-clicking on it and choosing Delete**. Now try building and running your project — you'll get an exception when the program starts:



Well, that makes sense, since you deleted MainWindow.xaml. When a WPF application starts up, it shows the **window specified in the StartupUri property in the <Application> tag App.xaml**:

```
<Application x:Class="BasketballRoster.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Open App.xaml and edit StartupUri so your program pops up the window you just added:

```
<Application x:Class="BasketballRoster.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="View/LeagueWindow.xaml">
```

Once you make that change, rebuild and rerun your program. Now it should start and show your newly added window.

User controls let you create your own controls

Take a look at the basketball roster program that you're building. Each team gets an identical set of controls: a TextBlock, another TextBlock, a ListView, another TextBlock, and another ListView, all wrapped up by a StackPanel inside a Border. Do we really need to add two identical sets of controls to the page? What if we want to add a third and fourth team — that's going to mean a whole lot of duplication. And that's where **user controls** come in. A user control is a class that you can use to create your own controls. You use XAML and code-behind to build a user control, just like you do when you build a page. Let's get started and add a user control to your BasketballRoster project.

- **❶ Add a new user control to your View folder.**
  Right-click on the *View* folder and add a new item. Choose

from the dialog and call it *RosterControl.xaml*.
- **❷ Look at the code-behind for the new user control.**
  Open up *RosterControl.xaml.cs*. Your new control extends the `UserControl` base class. Any code-behind that defines the user control's behavior goes here.

```
namespace BasketballRoster.View
{
    /// <summary>
    /// Interaction logic for RosterControl.xaml
    /// </summary>
    public partial class RosterControl : UserControl
    {
        public RosterControl()
        {
            InitializeComponent();
        }
    }
}
```

- **❸ Look at the XAML for the new user control.**
  The IDE added a user control with an empty `<Grid>`. Your XAML will go here.

UserControl is a base class that gives you a way to encapsulate controls that are related to each other, and lets you build logic that defines the behavior of the control.

Before you flip the page, see if you can figure out what XAML should go into the new `RosterControl` by looking at the Windows Store app screenshot in The Head First Basketball Conference needs an app.

- It will have a `<StackPanel>` to stack up the controls that live inside a blue `<Border>`. Can you figure out which property gives a Border control rounded corners?
- It has two ListView controls that display data for players, so it also needs a `<UserControl.Resources>` section that contains a `DataTemplate`. We called it `PlayerItemTemplate`.
- Bind the ListView items to properties called `Starters` and `Bench`, and the top TextBlock to a property called `TeamName`.
- The Border control lives inside a `<Grid>` with a single row that has `Height="Auto"` to keep it from expanding past the bottom of the ListView controls to fill up the entire page.

"Teach a man to fish..."

We're nearing the end of the book, so we want to challenge you with problems that are similar to ones you'll face in the real world. A good programmer takes a lot of educated guesses, so we're giving you barely enough information about how a `UserControl` works. You don't even have binding set up, so you won't see data in the designer! How much of the XAML can you build before you flip the page to see the code for `RosterControl`?

- **❹ Finish the RosterControl XAML.**
  Here's the code for the `RosterControl` user control that you added to the *View* folder. Did you notice how we gave you properties for binding, but no data context? That should make sense. The two controls on the page show different data, so the page will set different data contexts for each of them.

```
<UserControl x:Class="BasketballRoster.View.RosterControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="300">

    <UserControl.Resources>
        <DataTemplate x:Key="PlayerItemTemplate">
            <TextBlock>
                <Run Text="{Binding Name, Mode=OneWay}"/>
                <Run Text=" #"/>
                <Run Text="{Binding Number, Mode=OneWay}"/>
            </TextBlock>
        </DataTemplate>
    </UserControl.Resources>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <Border BorderThickness="2" BorderBrush="Blue" CornerRadius="6" Background="Black">
            <StackPanel Margin="20">
                <TextBlock Foreground="White" FontFamily="Segoe" FontSize="20px"
                           FontWeight="Bold" Text="{Binding TeamName}" />
                <TextBlock Foreground="White" FontFamily="Segoe" FontSize="16px"
                           Text="Starting Players" Margin="0,5,0,0"/>
                <ListView Background="Black" Foreground="White"  Margin="0,5,0,0"
                                      ItemTemplate="{StaticResource PlayerItemTemplate}"
                                      ItemsSource="{Binding Starters}" />
                <TextBlock Foreground="White"  FontFamily="Segoe" FontSize="16px"
                           Text="Bench Players" Margin="0,5,0,0"/>
                <ListView Background="Black" Foreground="White" ItemsSource="{Binding Bench}"
                          ItemTemplate="{StaticResource PlayerItemTemplate}" Margin="0,5,0,0"/>
            </StackPanel>
        </Border>
    </Grid>
</UserControl>
```

You already know that controls change size based on their Height and Width properties. You can change these numbers to alter how the control is displayed in the IDE's Designer window when you're modifying it.

We put the data template for the ListView items in its own static resource. Then, instead of having a <ListView.ItemTemplate> section we used the static resource using the ItemTemplate property in the ListView tag:

```
ItemTemplate="{StaticResource PlayerItemTemplate}"
```

You can use the CornerRadius property to give a Border rounded corners.

Both ListView controls use the same template defined as a static resource.

EXERCISE

Build the ViewModel for the BasketballRoster app by looking at the data in the Model and the bindings in the View, and figuring out what "plumbing" the app needs to connect them together.

- **❶ ADD THE ROSTER CONTROLS TO LEAGUEWINDOW.XAML.**

First add these `xmlns` properties to the page so it recognizes the new namespaces:

```
xmlns:view="clr-namespace:BasketballRoster.View"
xmlns:viewmodel="clr-namespace:BasketballRoster.ViewModel"
```

Then add an instance of `LeagueViewModel` as a static resource:

```
<Window.Resources>
    <viewmodel:LeagueViewModel x:Key="LeagueViewModel"/>
</Window.Resources>
```

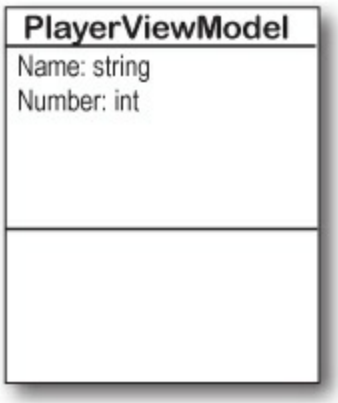
Now you can add a StackPanel with two `RosterControls` to the page:

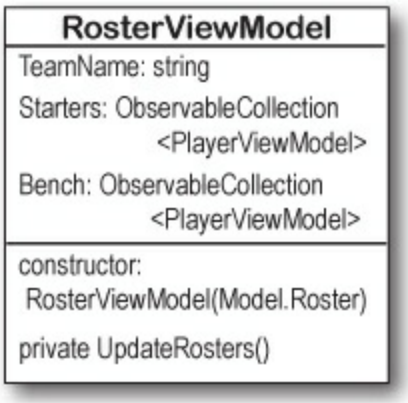
```
<StackPanel Orientation="Horizontal" Margin="5"
            VerticalAlignment="Center" HorizontalAlignment="Center"
            DataContext="{StaticResource ResourceKey=LeagueViewModel}" >
    <view:RosterControl Width="200" DataContext="{Binding JimmysTeam}" Margin="0,0,20,0" />
    <view:RosterControl Width="200" DataContext="{Binding BriansTeam}" />
</StackPanel>
```

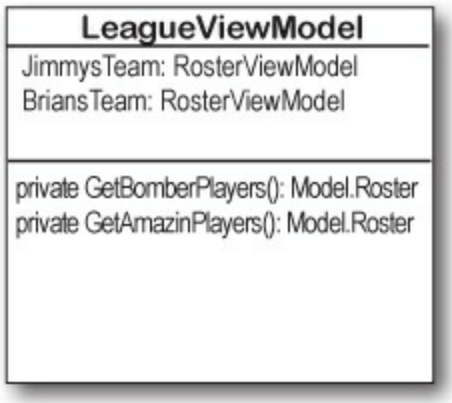
- **❷ CREATE THE VIEWMODEL CLASSES.**

> **NOTE**
>
> Make sure you created the classes and pages in the right folders; otherwise, the namespaces won't match the code in the solution.

Create these three classes in the *ViewModel* folder.







- **❸ MAKE THE VIEWMODEL CLASSES WORK.**

The `PlayerViewModel` class is a simple data object with two properties.
The `LeagueViewModel` class has two private methods to create dummy data for the page. It creates `Model.Roster` objects for each team that get passed to the RosterViewModel constructor.
The `RosterViewModel` class has a constructor that takes a `Model.Roster` object. It sets the `TeamName` property, and then it calls its private `UpdateRosters()` method, which uses LINQ queries to extract the starting and bench players and update the `Starters` and `Bench` properties. Add **`using Model;`** to the top of the classes so you can use objects in the `Model` namespace.

NOTE

— See Do you design for binding or for working with data? for a hint about the LINQ query...

If the IDE gives you an error message in the XAML designer that `LeagueViewModel` does not exist in the `ViewModel` namespace, but you're 100% certain you added it correctly, try right-clicking on the BasketballRoster project and choosing Unload Project, and then right-click again and choose Reload Project to reload it. But make sure you don't have any errors in any of the C# code files.

EXERCISE SOLUTION

The ViewModel for the BasketballRoster app has three classes: `LeagueViewModel`, `PlayerViewModel`, and `RosterViewModel`. They all live in the *ViewModel* folder.

If you left out the `using Model;` line then you'd have to use `Model.Roster` instead of `Roster` everywhere.

```
namespace BasketballRoster.ViewModel {
    using Model;
    using System.Collections.ObjectModel;

    class LeagueViewModel {
        public RosterViewModel BriansTeam { get; set; }
        public RosterViewModel JimmysTeam { get; set; }

        public LeagueViewModel() {
            Roster briansRoster = new Roster("The Bombers", GetBomberPlayers());
            BriansTeam = new RosterViewModel(briansRoster);

            Roster jimmysRoster = new Roster("The Amazins", GetAmazinPlayers());
            JimmysTeam = new RosterViewModel(jimmysRoster);
        }

        private IEnumerable<Player> GetBomberPlayers() {
            List<Player> bomberPlayers = new List<Player>() {
                new Player("Brian", 31, true),
                new Player("Lloyd", 23, true),
                new Player("Kathleen",6, true),
                new Player("Mike", 0, true),
                new Player("Joe", 42, true),
                new Player("Herb",32, false),
                new Player("Fingers",8, false),
            };
            return bomberPlayers;
        }

        private IEnumerable<Player> GetAmazinPlayers() {
            List<Player> amazinPlayers = new List<Player>() {
                new Player("Jimmy",42, true),
                new Player("Henry",11, true),
                new Player("Bob",4, true),
                new Player("Lucinda", 18, true),
                new Player("Kim", 16, true),
                new Player("Bertha", 23, false),
                new Player("Ed",21,  false),
            };
            return amazinPlayers;
        }
    }
}
```

LeagueViewModel exposes RosterViewModel objects that a RosterControl can use as its data context. It creates the Roster model object for the RosterViewModel to use.

This private method generates dummy data for the Bombers by creating a new List of Player objects.

Dummy data typically goes in the ViewModel because the state of an MVVM application is managed using instances of the Model classes that are encapsulated inside the ViewModel objects.

You use classes from the View to store your data, which is why this method returns Player objects and not PlayerViewModel objects.

Here's the PlayerViewModel. It's just a simple data object with properties for the data template to bind to.

```
namespace BasketballRoster.ViewModel {
    class PlayerViewModel {
        public string Name { get; set; }
        public int Number { get; set; }

        public PlayerViewModel(string name, int number) {
            Name = name;
            Number = number;
        }
    }
}
```

In a typical MVVM app, only classes in the ViewModel implement `INotifyPropertyChanged` because those are the only objects that XAML controls are bound to.

```
namespace BasketballRoster.ViewModel {
    using Model;
    using System.Collections.ObjectModel;
    using System.ComponentModel;

    class RosterViewModel {
        public ObservableCollection<PlayerViewModel> Starters { get; set; }
        public ObservableCollection<PlayerViewModel> Bench { get; set; }

        private Roster _roster;

        private string _teamName;
        public string TeamName {
            get { return _teamName; }
            set {
                _teamName = value;
            }
        }

        public RosterViewModel(Roster roster) {
            _roster = roster;

            Starters = new ObservableCollection<PlayerViewModel>();
            Bench = new ObservableCollection<PlayerViewModel>();

            TeamName = _roster.TeamName;

            UpdateRosters();
        }

        private void UpdateRosters() {
            var startingPlayers =
                from player in _roster.Players
                where player.Starter
                select player;

            foreach (Player player in startingPlayers)
                Starters.Add(new PlayerViewModel(player.Name, player.Number));

            var benchPlayers =
                from player in _roster.Players
                where player.Starter == false
                select player;

            foreach (Player player in benchPlayers)
                Bench.Add(new PlayerViewModel(player.Name, player.Number));
        }
    }
}
```

This is where the app stores its state—in Roster objects encapsulated inside the ViewModel. The rest of the class translates the Model data into properties that the View can bind to.

Whenever the TeamName property changes, the RosterViewModel fires off a PropertyChanged event so any object bound to it will get updated.

This LINQ query finds all the starting players and adds them to the Starters ObservableCollection property.

Here's a similar LINQ query to find the bench players.

In a typical MVVM app, only classes in the ViewModel implement `INotifyPropertyChanged`. That's because the ViewModel contains the only objects that XAML controls are bound to. In this project, however, we didn't need to implement INotifyPropertyChanged because the bound properties are updated in the constructor. If you wanted to modify the project to let Brian and Jimmy change their team names, you'd need to fire a PropertyChanged event in the TeamName set accessor.

There is one change you'll need to make to get the ViewModel code in Add the stopwatch ViewModel and 767 in the book to work. In Add the stopwatch ViewModel you're given three using statements, including this one:

```
using Windows.UI.Xaml;
```

You'll need to replace it with this using statement:

```
using System.Windows.Threading;
```

The Windows.UI.Xaml namespace is part of the .NET Framework for Windows Store, so you don't use it for WPF applications. But you need System.Windows.Threading because your ViewModel has a DispatcherTimer.

Other than that change, the code is identical. This is a good example of decoupled layers in the Model-View-ViewModel pattern: since you used identical C# code (except for that one using statement) for the ViewModel and Model, you could reuse those classes to port the stopwatch app to WPF.

Build the view for a simple stopwatch

Here's the XAML for a simple stopwatch control. **Add a WPF user control to the *View* folder called *BasicStopwatch.xaml*** and add this code. The control has TextBlock controls to display the elapsed and lap times, and buttons to start, stop, reset, and take the lap time.

```
<UserControl x:Class="Stopwatch.View.BasicStopwatch"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300"
             xmlns:viewmodel="clr-namespace:Stopwatch.ViewModel">

    <UserControl.Resources>
        <viewmodel:StopwatchViewModel x:Key="viewModel"/>
    </UserControl.Resources>

    <Grid DataContext="{StaticResource ResourceKey=viewModel}">
        <StackPanel>
            <TextBlock>
                <Run>Elapsed time: </Run>
                <Run Text="{Binding Hours, Mode=OneWay}"/>
                <Run>:</Run>
                <Run Text="{Binding Minutes, Mode=OneWay}"/>
                <Run>:</Run>
                <Run Text="{Binding Seconds, Mode=OneWay}"/>
            </TextBlock>
            <TextBlock>
                <Run>Lap time: </Run>
                <Run Text="{Binding LapHours, Mode=OneWay}"/>
                <Run>:</Run>
                <Run Text="{Binding LapMinutes, Mode=OneWay}"/>
                <Run>:</Run>
                <Run Text="{Binding LapSeconds, Mode=OneWay}"/>
            </TextBlock>
            <StackPanel Orientation="Horizontal">
                <Button Click="StartButton_Click" Margin="0,0,5,0">Start</Button>
                <Button Click="StopButton_Click" Margin="0,0,5,0">Stop</Button>
                <Button Click="ResetButton_Click" Margin="0,0,5,0">Reset</Button>
                <Button Click="LapButton_Click">Lap</Button>
            </StackPanel>
        </StackPanel>
    </Grid>
</UserControl>
```

You'll need this xmlns property to add the namespace. We called our project Stopwatch, so the ViewModel namespace is Stopwatch.ViewModel.

This user control stores an instance of the ViewModel as a static resource and uses it as its data context. It doesn't need its container to set a data context. It keeps track of its own state.

The ViewModel has read-only properties for Hours, Minutes, Seconds, etc. WPF requires one-way binding for read-only properties.

This TextBlock is bound to properties in the ViewModel that return the elapsed time.

This TextBlock is bound to properties that expose the lap time.

The ViewModel must be firing off PropertyChanged events to keep these values up to date.

You'll need to add `Click` event handlers to the control and a `StopwatchViewModel` class to the `ViewModel` namespace for this to compile.

NOTE

Here's a hint: use a DispatcherTimer to constantly check the Model and update the properties.

The code for the ViewModel is in Add the stopwatch ViewModel and 767 in the book. How much of the ViewModel code can you build just from the View and Model code before you flip

the page? Add a BasicStopwatch control to the main window and see how far you can get.

But be really careful and don't assume the IDE is necessarily wrong. Sometimes an error in the XAML for one page (like a broken `xmlns` property) can cause all the designers to break.

Finish the stopwatch app

There are just a few more loose ends to tie together. Your BasicStopwatch user control doesn't have event handlers, so you need to add them. And then you just need to add the control to your main window.

- ❶ First, **go back to *BasicStopwatch.xaml.cs*** and add these event handlers to the code-behind:

```
ViewModel.StopwatchViewModel viewModel;

public BasicStopwatch() {
    InitializeComponent();

    viewModel = FindResource("viewModel") as ViewModel.StopwatchViewModel;
}

private void StartButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Start();
}

private void StopButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Stop();
}

private void ResetButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Reset();
}

private void LapButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Lap();
}
```

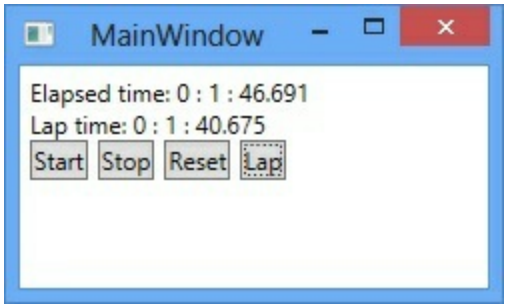
The buttons in the view just call methods in the ViewModel. This is a pretty typical pattern for the View.

- ❷ Here's **all the XAML** for *MainWindow.xaml*:

```
<Window x:Class="Stopwatch.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="150" Width="250"
        xmlns:view="clr-namespace:Stopwatch.View">
    <Grid>
        <view:BasicStopwatch Margin="5"/>
    </Grid>
</Window>
```

All the behavior is in the user control, so there's no code-behind for the main window.

Your app should now run. Click the Start, Stop, Reset, and Lap buttons to see your stopwatch work.

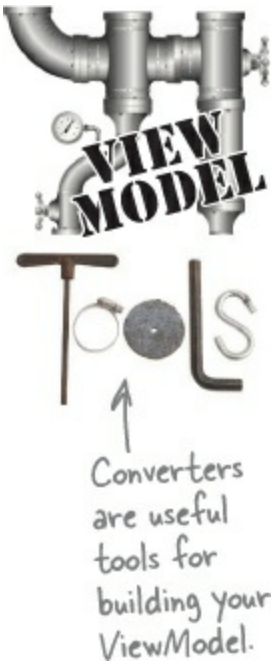


NOTE

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Converters automatically convert values for binding

Anyone with a digital clock knows that it typically shows the minutes with a leading zero. Our stopwatch should also show the minutes with two digits. And it should show the seconds with two digits, and round to the nearest hundredth of a second. We *could* modify the ViewModel to expose string values that are formatted properly, but that would mean that we'd need to keep adding more and more properties each time we wanted to reformat the same data. That's where **value converters** come in very handy. A value converter is an object that the XAML binding uses to modify data before it's passed to the control. You can build a value converter by implementing the `IValueConverter` interface (which is in the `System.Windows.Data` namespace). Add a value converter to your stopwatch now.



- **❶ Add the TimeNumberFormatConverter class to the ViewModel folder.**
  Add `using System.Windows.Data;` to the top of the class, and then have it implement the `IValueConverter` interface. Use the IDE to automatically implement the interface. This will add two method stubs for the `Convert()` and `ConvertBack()` methods.
- **❷ Implement the Convert() method in the value converter.**
  The `Convert()` method takes several parameters — we'll use two of them. The `value` parameter is the raw value that's passed into the binding, and `parameter` lets you specify a parameter in XAML.

```
using System.Windows.Data;

class TimeNumberFormatConverter : IValueConverter {
    public object Convert(object value, Type targetType,
        object parameter, System.Globalization.CultureInfo culture) {
        if (value is decimal)
            return ((decimal)value).ToString("00.00");
        else if (value is int) {
            if (parameter == null)
                return ((int)value).ToString("d1");
            else
                return ((int)value).ToString(parameter.ToString());
        }
        return value;
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, System.Globalization.CultureInfo culture) {
        throw new NotImplementedException();
    }
}
```

This converter knows how to convert decimal and int values. For int values, you can optionally pass in a parameter.

The ConvertBack() method is used for two-way binding. We're not using that in this project, so you can leave the method stub as is.

Is it a good idea to leave this `NotImplementedException` in your code? For this project, this is code that is never supposed to be run. If it does get run, is it better to fail silently, so the user never sees it? Or is it better to throw an exception so that you can track down the problem? Which of those gives you a more robust app? There's not necessarily one right answer.

- **❸ Add the converter to your stopwatch control as a static resource.**
  It should go right below the `ViewModel` object:

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Key="viewModel"/>
    <viewmodel:TimeNumberFormatConverter x:Key="timeNumberFormatConverter"/>
</UserControl.Resources>
```

The designer may make you rebuild the solution after you add this line. In rare cases, you might even need to unload and reload the project.

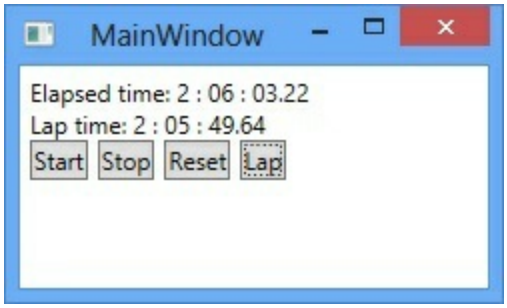
- **❹ Update the XAML code to use the value converter.**
  Modify the `{Binding}` markup by adding the `Converter=` to it in each of the `<Run>` tags.

```
<TextBlock>
    <Run>Elapsed time: </Run>
    <Run Text="{Binding Hours, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}}"/>
    <Run>:</Run>
    <Run Text="{Binding Minutes, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}, ConverterParameter=d2}"/>
    <Run>:</Run>
    <Run Text="{Binding Seconds, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}}"/>
</TextBlock>
<TextBlock>
    <Run>Lap time: </Run>
    <Run Text="{Binding LapHours, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}}"/>
    <Run>:</Run>
    <Run Text="{Binding LapMinutes, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}, ConverterParameter=d2}"/>
    <Run>:</Run>
    <Run Text="{Binding LapSeconds, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}}"/>
</TextBlock>
```

If there's no parameter specified, don't forget the extra closing bracket }}.

Use the ConverterParameter syntax to pass a parameter into the converter.

Now the stopwatch runs the values through the converter before passing them into the TextBlock controls, and the numbers are formatted correctly on the page.



Converters can work with many different types

TextBlock and TextBox controls work with text, so binding strings or numbers to the `Text` property makes sense. But there are many other properties, and you can bind to those as well. If your ViewModel has a Boolean property, it can be bound to any `true`/`false` property. You can even bind properties that use enums — the `IsVisible` property uses the `Visibility` enum, which means you can also write value converters for it. Let's add Boolean and `Visibility` binding and conversion to the stopwatch.

Here are two converters that will come in handy.

Sometimes you want to bind Boolean properties like `IsEnabled` so that a control is enabled if the bound property is `false`. We'll add a new converter called `BooleanNotConverter`, which uses the `!` operator to invert a Boolean target property.

```
IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
```

You'll often want to have controls show or hide themselves based on a Boolean property in the data context. You can only bind the Visibility property of a control to a target property that's of the type `Visibility` (meaning it returns values like `Visibility.Collapsed`). We'll add a converter called `BooleanVisibilityConverter` that will let us bind a control's `Visibility` property to a Boolean target property to make it visible or invisible.

```
Visibility="{Binding Running, Converter={StaticResource visibilityConverter}}"
```

- **❶ MODIFY THE VIEWMODEL'S TICK EVENT HANDLER.**
  Modify the `DispatcherTimer`'s `Tick` event handler to raise a `PropertyChanged` event if the value of the `Running` property has changed:

```
int _lastHours;
int _lastMinutes;
decimal _lastSeconds;
bool _lastRunning;
void TimerTick(object sender, object e) {
    if (_lastRunning != Running) {
        _lastRunning = Running;
        OnPropertyChanged("Running");
    }
    if (_lastHours != Hours) {
        _lastHours = Hours;
        OnPropertyChanged("Hours");
    }
    if (_lastMinutes != Minutes) {
        _lastMinutes = Minutes;
        OnPropertyChanged("Minutes");
    }
    if (_lastSeconds != Seconds) {
        _lastSeconds = Seconds;
        OnPropertyChanged("Seconds");
    }
}
```

We added the Running check to the timer. Would it make more sense to have the Model fire an event instead?

- **❷ ADD A CONVERTER THAT INVERTS BOOLEAN VALUES.**

Here's a value converter that converts `true` to `false` and vice versa. You can use it with Boolean properties on your controls like `IsEnabled`.

```
using System.Windows.Data;

class BooleanNotConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter,
                          System.Globalization.CultureInfo culture) {
        if ((value is bool) && ((bool)value) == false)
            return true;
        else
            return false;
    }
    public object ConvertBack(object value, Type targetType, object parameter,
                              System.Globalization.CultureInfo culture) {
        throw new NotImplementedException();
    }
}
```

- **❸ ADD A CONVERTER THAT CONVERTS BOOLEANS TO VISIBILITY ENUMS.** You've already seen how you can make a control visible or invisible by setting its `Visibility` property to `Visible` or `Collapsed`. These values come from an enum in the *System.Windows* namespace called `Visibility`. Here's a converter that converts Boolean values to Visibility values:

```
using System.Windows;
using System.Windows.Data;

class BooleanVisibilityConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter,
                          System.Globalization.CultureInfo culture) {
        if ((value is bool) && ((bool)value) == true)
            return Visibility.Visible;
        else
            return Visibility.Collapsed;
    }
    public object ConvertBack(object value, Type targetType, object parameter,
                              System.Globalization.CultureInfo culture) {
        throw new NotImplementedException();
    }
}
```

- **❹ MODIFY YOUR BASIC STOPWATCH CONTROL TO USE THE CONVERTERS.** Modify *BasicStopwatch.xaml* to add instances of these converters as static resources:

```
<viewmodel:BooleanVisibilityConverter x:Key="visibilityConverter"/>
<viewmodel:BooleanNotConverter x:Key="notConverter"/>
```

Now you can bind the controls' `IsEnabled` and `Visibility` properties to the ViewModel's `Running` property:

```
<StackPanel Orientation="Horizontal">
    <Button IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
            Click="StartButton_Click" Margin="0,0,5,0">Start</Button>
    <Button IsEnabled="{Binding Running}" Click="StopButton_Click"
            Margin="0,0,5,0">Stop</Button>
    <Button Click="ResetButton_Click" Margin="0,0,5,0">Reset</Button>
    <Button IsEnabled="{Binding Running}" Click="LapButton_Click">Lap</Button>
</StackPanel>
<TextBlock Text="Stopwatch is running"
           Visibility="{Binding Running, Converter={StaticResource visibilityConverter}}"/>
```

This enables the Start button only if the stopwatch is not running.

This causes a TextBlock to become visible when the stopwatch is running.

NOTE

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Build an analog stopwatch using the same ViewModel

The MVVM pattern **decouples** the View from the ViewModel, and the ViewModel from the Model. This is really useful if you need to make changes to one of the layers. Because of that decoupling, you can be very confident that the changes you make will not cause the "shotgun surgery" effect and ripple into the other layers. So did we do a good job decoupling the stopwatch program's View from its ViewModel? There's one way to be sure: let's build an entirely new View without changing the existing classes in the ViewModel. The only change you'll need in the C# code **is a new converter in the ViewModel** that converts minutes and seconds into angles.

NOTE

Remember how you used the data classes you built for Jimmy's Comics in Chapter 14 and reused them to create a Split App without making any changes? This is the same idea.

DO THIS!

- **❶ ADD A CONVERTER TO CONVERT TIME TO ANGLES.**
  Add the `AngleConverter` class to the *ViewModel* folder. You'll use it for the hands on the face.

```
using System.Windows.Data;
class AngleConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter,
                          System.Globalization.CultureInfo culture) {
        double parsedValue;
        if ((value != null)
            && double.TryParse(value.ToString(), out parsedValue)
            && (parameter != null))
            switch (parameter.ToString()) {
                case "Hours":
                    return parsedValue * 30;
                case "Minutes":
                case "Seconds":
                    return parsedValue * 6;
            }
        return 0;
    }
    public object ConvertBack(object value, Type targetType, object parameter,
                              System.Globalization.CultureInfo culture) {
        throw new NotImplementedException();
    }
}
```

An hour value ranges from 0 to 11, so to convert to an angle it's multiplied by 30.

Minutes and seconds range from 0 to 60, so the angle conversion means multiplying by 6.

- **❷ ADD THE NEW USERCONTROL.**
  Add a **new WPF user control called `AnalogStopwatch` to the *View* folder** and add the `ViewModel` namespace to the `<UserControl>` tag. Also, change the design width and height:

```
d:DesignHeight="300"
d:DesignWidth="400"
xmlns:viewmodel="clr-namespace:Stopwatch.ViewModel">
```

And add the ViewModel, two converters, and a style to the user control's static resources.

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Key="viewModel"/>
    <viewmodel:BooleanNotConverter x:Key="notConverter"/>
    <viewmodel:AngleConverter x:Key="angleConverter"/>
</UserControl.Resources>
```

- **❸ ADD THE FACE AND HANDS TO THE GRID.**
  Modify the `<Grid>` tag to add the stopwatch face, using four rectangles for hands.

```
<Grid x:Name="baseGrid" DataContext="{StaticResource ResourceKey=viewModel}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="400"/>
    </Grid.ColumnDefinitions>
    <Ellipse Width="300" Height="300" Stroke="Black" StrokeThickness="2">
        <Ellipse.Fill>
            <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                <GradientStop Color="#FFB03F3F"/>
                <GradientStop Color="#FFE4CECE" Offset="1"/>
            </LinearGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="2" Height="150" Fill="Black">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-60"/>
                <RotateTransform Angle="{Binding Seconds,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Seconds}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="4" Height="100" Fill="Black">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-50"/>
                <RotateTransform Angle="{Binding Minutes,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Minutes}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="1" Height="150" Fill="Yellow">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-60"/>
                <RotateTransform Angle="{Binding LapSeconds,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Seconds}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="2" Height="100" Fill="Yellow">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-50"/>
                <RotateTransform Angle="{Binding LapMinutes,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Minutes}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Ellipse Width="10" Height="10" Fill="Black"/>
</Grid>
```

Setting the column width keeps it from expanding to fill whatever container it's in.

This is the face of the stopwatch. It has a black outline and a grayish gradient background.

Here's the second hand. It's a long, thin rectangle with a translate and rotate transform.

Here's the minute hand.

Every control can have one RenderTransform section.

There are two yellow hands for the lap time.

The TransformGroup tag lets you apply multiple transforms to the same control.

This draws an extra circle in the middle to cover up where the hands overlap. Since it's at the bottom of the Grid, it's drawn last and ends up on top.

The stopwatch face is filled with a gradient brush, just like the background you used in *Save the Humans*.

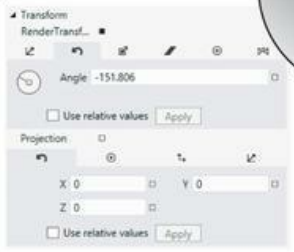
Each hand is transformed twice. It starts out centered in the face, so the first transform shifts it up so that it's in position to rotate.

```
<TranslateTransform Y="-60"/>
```

```
<RotateTransform Angle="{Binding Seconds,
    Converter={StaticResource ResourceKey=angleConverter},
    ConverterParameter=Seconds}"/>
```

The second transform rotates the hand to the correct angle. The `Angle` property of the rotation is bound to seconds or minutes in the ViewModel, and uses the angle converter to convert it to an angle.

Every control can have one `RenderTransform` element that changes how it's displayed. This can include rotating, moving to an offset, skewing, scaling its size up or down, and more.



You used transforms in *Save the Humans* to change the shape of the ellipses in the enemy to make it look like an alien.

Your stopwatch will start ticking as soon as you add the second hand, because it creates an instance of the ViewModel as a static resource to render the control in the designer. The designer may stop it updating, but you can restart it by switching away from the designer window and back again.

- **❹ ADD THE BUTTONS TO THE STOPWATCH.**
  Since the ViewModel is the same, the buttons should work the same. Add the same buttons to *AnalogStopwatch.xaml* that you used for the basic stopwatch:

```
<StackPanel Orientation="Horizontal" VerticalAlignment="Bottom">
    <Button IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
    Click="StartButton_Click" Margin="0,0,5,0">Start</Button>
    <Button IsEnabled="{Binding Running}"
                      Click="StopButton_Click" Margin="0,0,5,0">Stop</Button>
    <Button Click="ResetButton_Click" Margin="0,0,5,0">Reset</Button>
    <Button IsEnabled="{Binding Running}" Click="LapButton_Click">Lap</Button>
</StackPanel>
```

Here's the code-behind for *AnalogStopwatch.xaml.cs*:

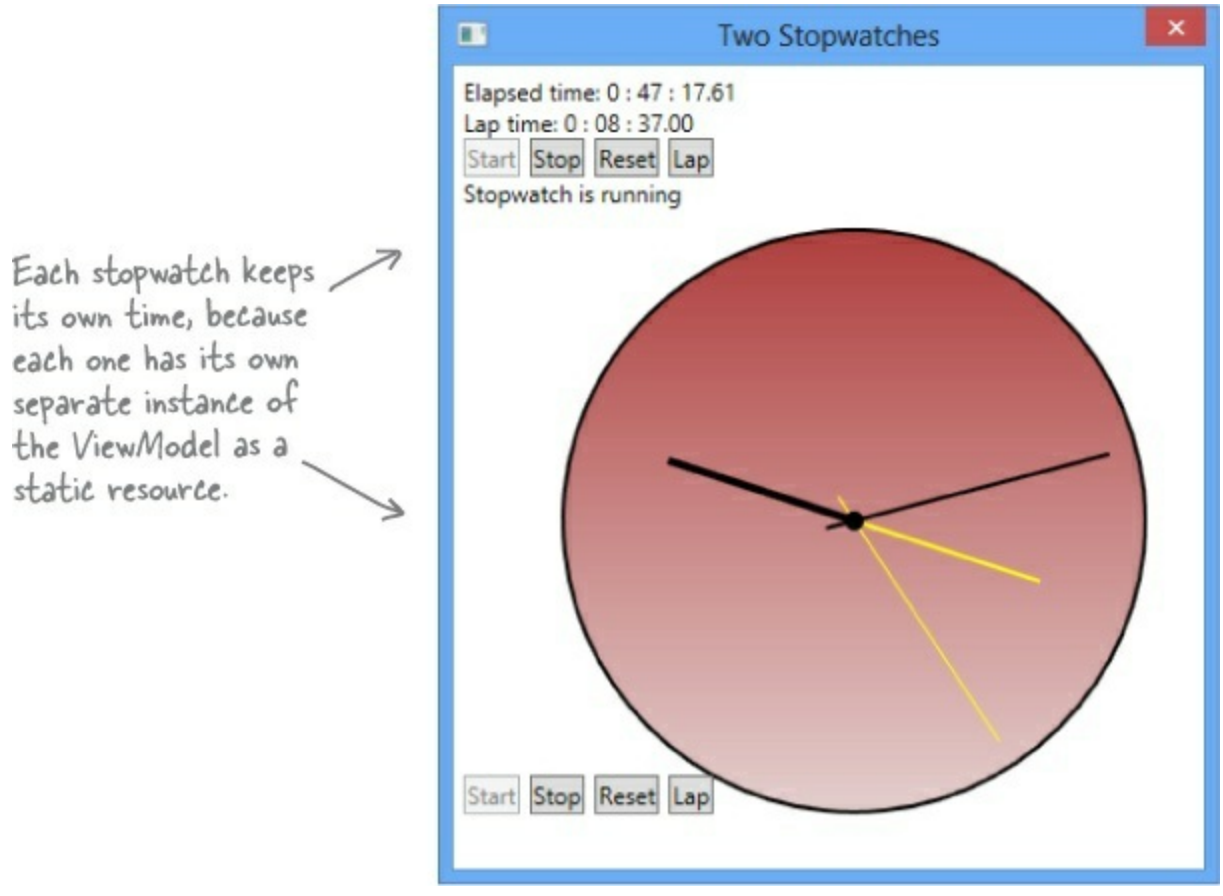
```
ViewModel.StopwatchViewModel viewModel;

public AnalogStopwatch() {
    InitializeComponent();

    viewModel = FindResource("viewModel") as ViewModel.StopwatchViewModel;
```

```
}

private void StartButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Start();
}

private void StopButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Stop();
}

private void ResetButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Reset();
}

private void LapButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Lap();
}
```

- **❺ UPDATE THE MAIN WINDOW TO SHOW BOTH STOPWATCHES.**
  Now you just need to modify your *MainWindow.xaml* to add an `AnalogStopwatch` control:

```
<Window x:Class="Stopwatch.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Two Stopwatches" Height="450" Width="400" ResizeMode="NoResize"
        xmlns:view="clr-namespace:Stopwatch.View">
    <Grid>
        <StackPanel>
            <view:BasicStopwatch Margin="5"/>
            <view:AnalogStopwatch Margin="5"/>
        </StackPanel>
    </Grid>
</Window>
```

Run your app. Now you have two stopwatch controls on the page.



Try changing the ViewModel to make the _stopwatchModel field static. What does this change about how the stopwatch app behaves? Can you figure out why that happens?

UI controls can be instantiated with C# code, too

You already know that your XAML code instantiates classes in the `Windows.UI` namespace, and you even used the Watch window in the IDE back in Chapter 10 to explore them. But what if you want to create controls from inside your code? Well, controls are just objects, so you can create them and work with them just like you would with any other object. Go ahead and **modify the code-behind to add markings to the face of your analog stopwatch**.

```
public sealed partial class AnalogStopwatch : UserControl {
    public AnalogStopwatch() {
        InitializeComponent();

        viewModel = FindResource("viewModel") as ViewModel.StopwatchViewModel;
        AddMarkings();
    }

    private void AddMarkings() {
        for (int i = 0; i < 360; i += 3) {
            Rectangle rectangle = new Rectangle();
            rectangle.Width = (i % 30 == 0) ? 3 : 1;
            rectangle.Height = 15;
            rectangle.Fill = new SolidColorBrush(Colors.Black);
            rectangle.RenderTransformOrigin = new Point(0.5, 0.5);

            TransformGroup transforms = new TransformGroup();
            transforms.Children.Add(new TranslateTransform() { Y = -140 });
            transforms.Children.Add(new RotateTransform() { Angle = i });
            rectangle.RenderTransform = transforms;
            baseGrid.Children.Add(rectangle);
        }
    }
    // ... the button event handlers stay the same
```

Modify the constructor to call a method that adds the markings.

This statement uses the % modulo operator to make the marks for the hours thicker than the ones for the minutes. `i % 30` returns 0 only if `i` is divisible by 30.

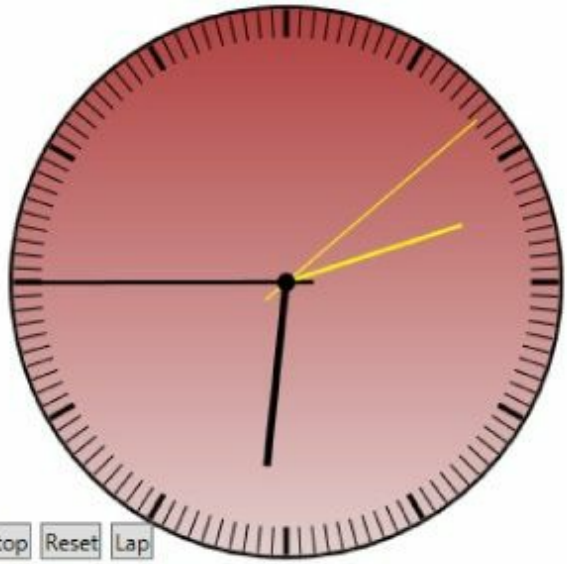
This creates instances of the same Rectangle object that you created with the <Rectangle> tag.

Flip back to the XAML for the hour and minute hands. This code sets up exactly the same transform, except instead of binding the Angle property it sets it to a value.

Controls like Grid, StackPanel, and Canvas have a `Children` collection with references to all the other controls contained inside them. You can add controls to the grid with its `Add()` method and remove all controls by calling its `Clear()` method. You add transforms to a `TransformGroup` the same way.

You used a `Binding` object to set up data binding in C# code back in Chapter 11. Can you figure out how to remove the XAML to create the Rectangle controls for the hour and minute hands and replace it with C# code to do the same thing?

Now that you added the markings to the stopwatch, the ref will make all the right calls.



Which team will dominate the conference and win the Objectville Trophy? Nobody's sure. All we know is that Joe, Bob, and Ed will be betting on it!

For the next few projects, you'll need to download the bee images from the your class website. Make sure that you add the images to your project so they're in the top-level folder, just like you did with the Jimmy's Comics app. You'll also need to select each image file in the Solution Explorer and use the Properties window to set the "Build Action" to `Content` and "Copy to Output Directory" to `Copy always`. Here's what it looks like when you did it for the Jimmy's Comics app:

Make sure you do this for `Bee animation 1.png`, `Bee animation 2.png`, `Bee animation 3.png`**, and** `Bee animation 4.png`.

Create a user control to animate a picture

Let's encapsulate all the frame-by-frame animation code. **Add a WPF user control called *AnimatedImage* to your View folder**. It has very little XAML — all the intelligence is in the code-behind. Here's everything inside the `<UserControl>` tag in the XAML:

```
<Grid>
    <Image x:Name="image" Stretch="Fill"/>
</Grid>
```

The work is done in the code-behind. Notice its overloaded constructor that calls the `StartAnimation()` method, which **creates storyboard and key frame animation objects** to animate the `Source` property of the Image control.

```
using System.Windows.Media.Animation;
using System.Windows.Media.Imaging;

public partial class AnimatedImage : UserControl {
    public AnimatedImage() {
        InitializeComponent();
    }

    public AnimatedImage(IEnumerable<string> imageNames, TimeSpan interval)
        : this() {
        StartAnimation(imageNames, interval);
    }

    public void StartAnimation(IEnumerable<string> imageNames, TimeSpan interval) {
        Storyboard storyboard = new Storyboard();
        ObjectAnimationUsingKeyFrames animation = new ObjectAnimationUsingKeyFrames();
        Storyboard.SetTarget(animation, image);
        Storyboard.SetTargetProperty(animation, new PropertyPath(Image.SourceProperty));

        TimeSpan currentInterval = TimeSpan.FromMilliseconds(0);
        foreach (string imageName in imageNames) {
            ObjectKeyFrame keyFrame = new DiscreteObjectKeyFrame();
            keyFrame.Value = CreateImageFromAssets(imageName);
            keyFrame.KeyTime = currentInterval;
            animation.KeyFrames.Add(keyFrame);
            currentInterval = currentInterval.Add(interval);
        }

        storyboard.RepeatBehavior = RepeatBehavior.Forever;
        storyboard.AutoReverse = true;
        storyboard.Children.Add(animation);
        storyboard.Begin();
    }
    private static BitmapImage CreateImageFromAssets(string imageFilename) {
        try {
            Uri uri = new Uri(imageFilename, UriKind.RelativeOrAbsolute);
            return new BitmapImage(uri);
        } catch (System.IO.IOException) {
            return new BitmapImage();
        }
    }
}
```

BitmapImage is in the Media.Imaging namespace. Storyboard and the other animation classes are in the Media.Animation namespace.

Every control must have a parameterless constructor if you want to create an instance of the control using XAML. You can still add overloaded constructors, but that's useful only if you're writing code to create the control.

The static `SetTarget()` and `SetTargetProperty()` methods from the `Storyboard` class set the target object being animated (`"image"`), and the property that will change (`Source`) using the PropertyPath() class.

Once the `Storyboard` object is set up and animations have been added to its `Children` collection, call its `Begin()` method to start the animation.

This is the same method you used in Chapter 14.

Make your bees fly around a page

Let's take your AnimatedImage control out for a test flight.

DO THIS!

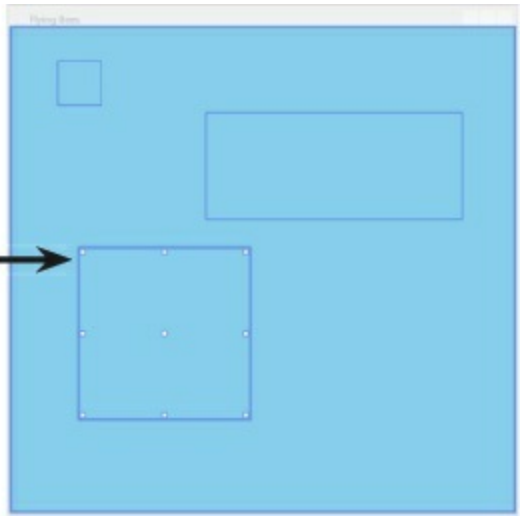
- **❶ REPLACE THE MAIN WINDOW WITH A WINDOW IN THE VIEW FOLDER.**
  Add a **Window to your *View* folder** called *FlyingBees.xaml*. Delete *MainWindow.xaml* from the project. Then **modify the StartupUri property in the <Application> tag App.xaml**:

```
StartupUri="View\FlyingBees.xaml"
```

- **❷ THE BEES WILL FLY AROUND A CANVAS CONTROL.**
  Here's the code for the window (you'll need to change the AnimatedBee namespace if you used a different project name). It uses a **Canvas control in *FlyingBees.xaml***. A Canvas control is a container, so it can contain other controls like a Grid or StackPanel. The difference is that a Canvas lets you set the coordinates of the controls using the `Canvas.Left` and `Canvas.Top` properties. You used a Canvas back in Chapter 1 to create the play area for *Save the Humans*. Here's the XAML for the *FlyingBees.xaml* window:

```
<Window x:Class="AnimatedBee.View.FlyingBees"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
               xmlns:view="clr-namespace:AnimatedBee.View"
        Title="Flying Bees" Height="600" Width="600">
    <Grid>
        <Canvas Background="SkyBlue">
            <view:AnimatedImage Canvas.Left="55" Canvas.Top="40"
                        x:Name="firstBee" Width="50" Height="50"/>
            <view:AnimatedImage Canvas.Left="80" Canvas.Top="260"
                        x:Name="secondBee" Width="200" Height="200"/>
            <view:AnimatedImage Canvas.Left="230" Canvas.Top="100"
                        x:Name="thirdBee" Width="300" Height="125"/>
        </Canvas>
    </Grid>
</Window>
```

The AnimatedImage control is invisible until its `CreateFrameImages()` method is called, so the controls in the Canvas will show up only as outlines. You can select them using the Document Outline. Try dragging the controls around the canvas to see the `Canvas.Left` and `Canvas.Top` properties change.



- **❸ ADD THE CODE-BEHIND FOR THE PAGE.**
  You'll need this `using` statement for the namespace that contains `Storyboard` and `DoubleAnimation`:

```
using System.Windows.Media.Animation;
```

Now you can **modify the constructor in *FlyingBees.xaml.cs*** to start up the bee animation. Let's also create a `DoubleAnimation` to animate the `Canvas.Left` property. Compare the code for creating a storyboard and animation to the XAML code with `<DoubleAnimation>` earlier in the chapter.

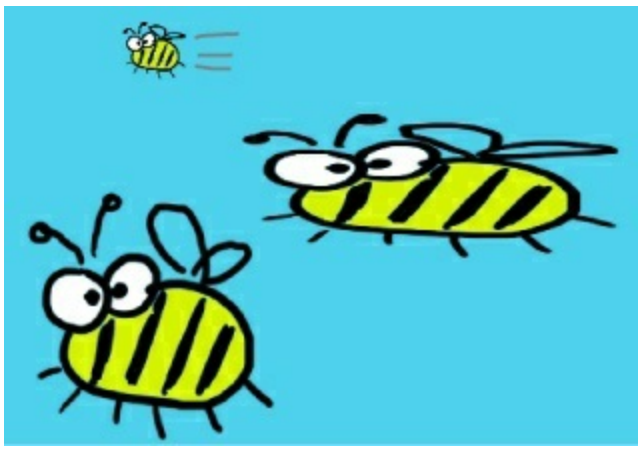
```
public FlyingBees() {
    this.InitializeComponent();

    List<string> imageNames = new List<string>();
    imageNames.Add("Bee animation 1.png");
    imageNames.Add("Bee animation 2.png");
    imageNames.Add("Bee animation 3.png");
    imageNames.Add("Bee animation 4.png");

    firstBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(50));
    secondBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(10));
    thirdBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(100));

    Storyboard storyboard = new Storyboard();
    DoubleAnimation animation = new DoubleAnimation();
    Storyboard.SetTarget(animation, firstBee);
    Storyboard.SetTargetProperty(animation, new PropertyPath(Canvas.LeftProperty));
    animation.From = 50;
    animation.To = 450;
    animation.Duration = TimeSpan.FromSeconds(3);
    animation.RepeatBehavior = RepeatBehavior.Forever;
    animation.AutoReverse = true;
    storyboard.Children.Add(animation);
    storyboard.Begin();
}
```

The CreateFrameImages() method takes a sequence of asset names and a TimeSpan to set the rate that the frames are updated.

Instead of using a <Storyboard> tag and a <DoubleAnimation> tag like earlier in the chapter, you can create the Storyboard and DoubleAnimation objects and set their properties in code.

The `Storyboard` is garbage-collected after the animation completes. You can see this for yourself by using Make Object ID to watch it and clicking ↻ to refresh it after the animation ends.

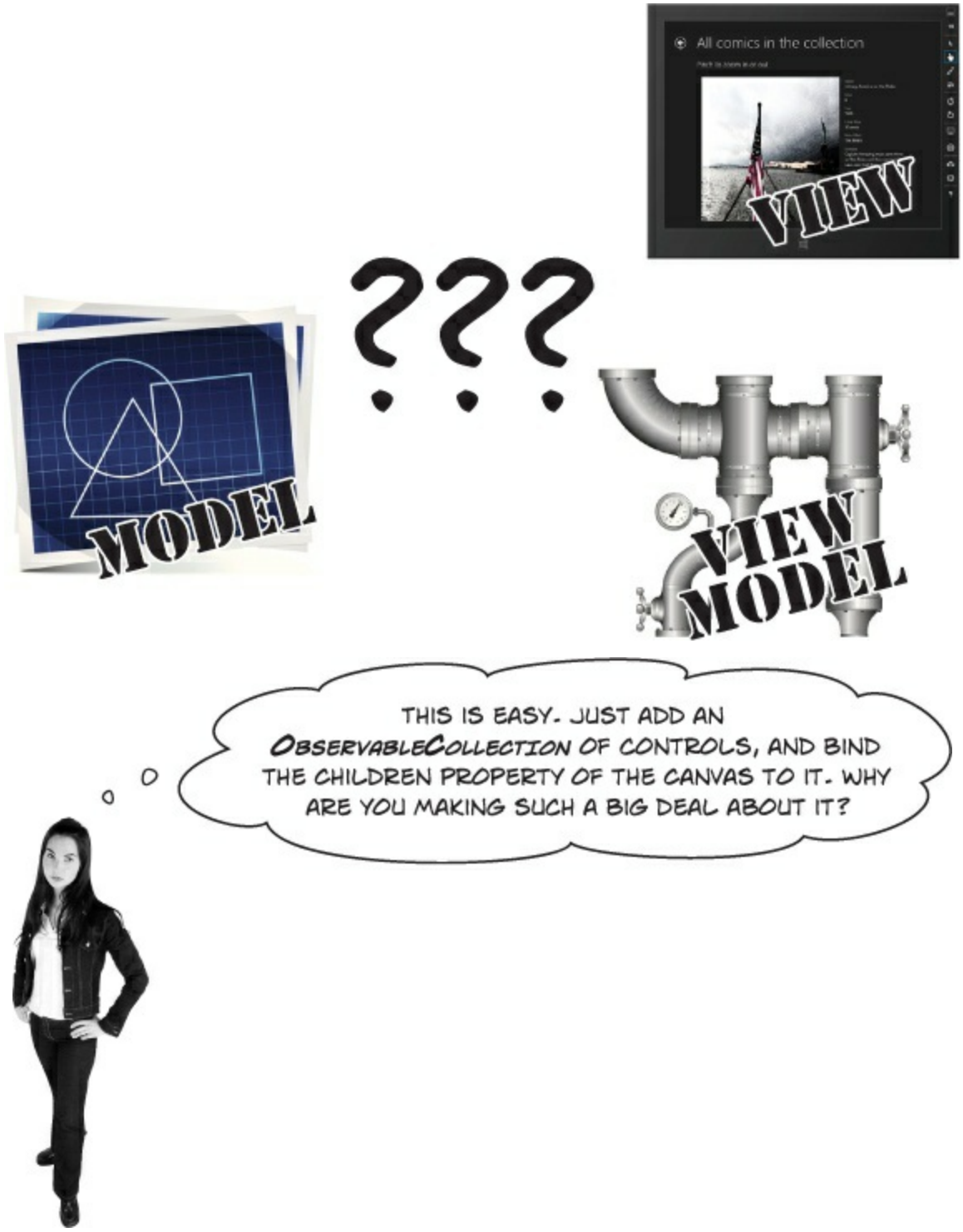
Run your program. Now you can see three bees flapping their wings. You gave them different intervals, so they flap at different rates because their timers are waiting for different timespans before changing frames. The top bee has its Canvas. Left property animated from 50 to 450 and back, which causes it to move around the page. Take a close look at the properties that are set on the DoubleAnimation object and compare them with the XAML properties you used earlier in the chapter.



Something's not right about this project. Can you spot it?

Something's not right: there's nothing in your *Model* or *ViewModel* folder, and you're creating dummy data in the View. That's not MVVM!

If we wanted to add more bees, we'd have to create more controls in the View and then initialize them individually. What if we want different sizes or kinds of bees? Or other things to be animated? If we had a Model that was optimized for data, it would be a lot easier. How can we make this project follow the MVVM pattern?



That won't work. Data binding doesn't work with container controls' `Children` property — and for good reason.

Data binding is built to work with **attached properties**, which are the properties that show up in the XAML code. The `Canvas` object *does* have a public `Children` property, but if you try to set it using XAML (`Children="{Binding ...}"`) your code ***won't compile***.

However, you already know how to bind a collection of objects to a XAML control, because you did that with ListView and GridView controls using the `ItemsSource` property. We can take advantage of that data binding to add child controls to a Canvas.

Use ItemsPanelTemplate to bind controls to a Canvas

When you used the `ItemsSource` property to bind items to a ListView, GridView, or ListBox, it didn't matter which one you were binding to, because the `ItemsSource` property always worked the same way. If you were going to build three classes that had exactly the same behavior, you would put that behavior in a base class and have the three classes extend it, right? Well, the Microsoft team did exactly the same thing when they built the selector controls. The ListView, GridView, and ListBox all extend a class called `Selector`, which is a subclass of **the `ItemsControl` class that displays a collection of items**.

- ❶ We're going to use its `ItemsPanel` property to **set up a template for the panel that controls the layout of the items**. Start by adding the `ViewModel` namespace to *FlyingBees.xaml*:

```
xmlns:viewmodel="clr-namespace:AnimatedBee.ViewModel"
```

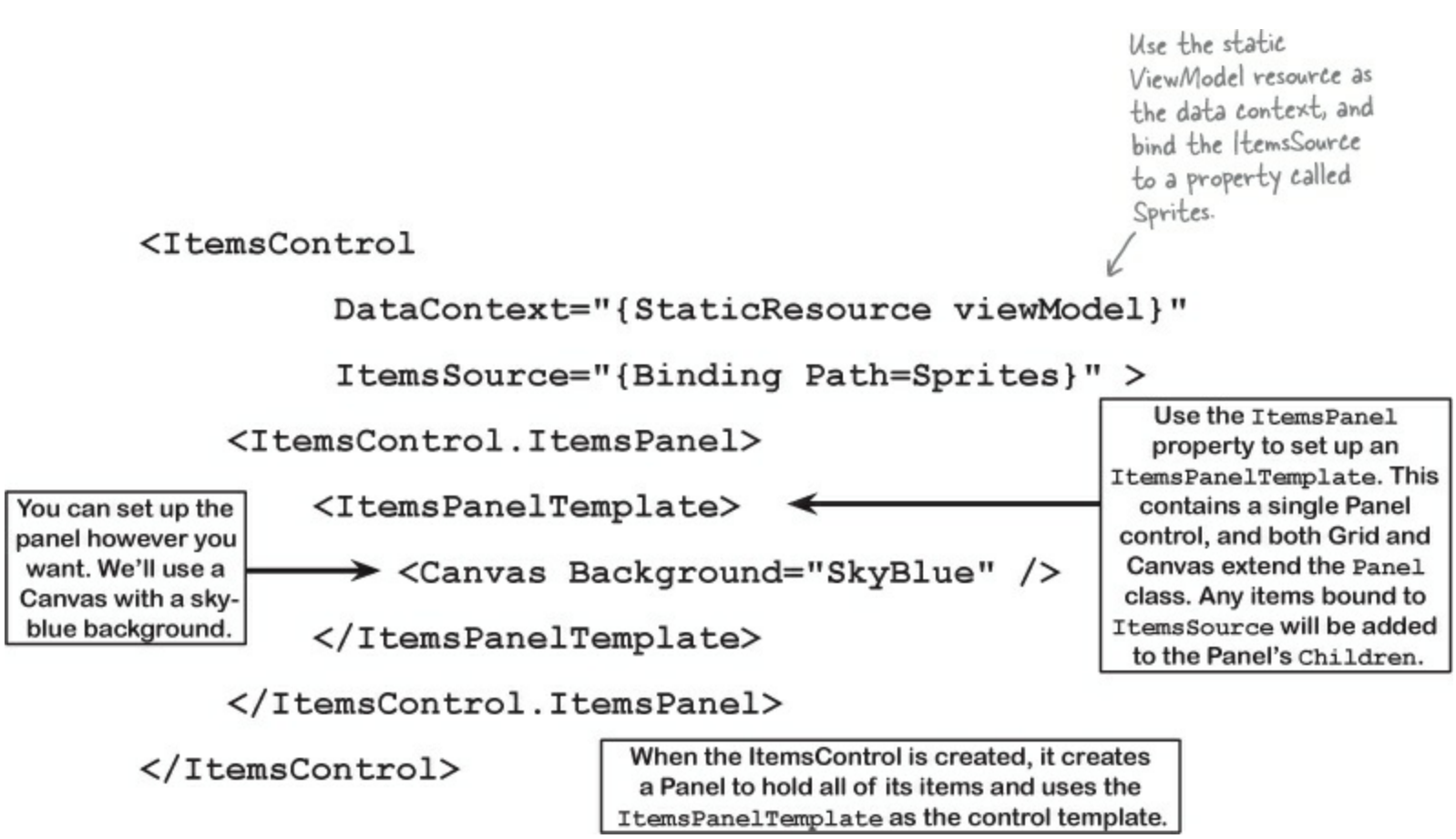
If you used a different project name, change AnimatedBee to the correct namespace.

- ❷ Next, **add an *empty* class called `BeeViewModel` to your *ViewModel* folder**, and then add an instance of that class as a static resource to *FlyingBees.xaml*:

```
<viewmodel:BeeViewModel x:Key="viewModel"/>
```

Edit *FlyingBees.xaml.cs* and **delete all the additional code that you added to the `FlyingBees()` constructor** in the `FlyingBees` control. Make sure that you ***don't delete*** the `InitializeComponent()` method!

- ❸ Here's the XAML for the ItemsControl. Open *FlyingBees.xaml*, **delete the `<Canvas>` tag** you added, and **replace it with this ItemsControl**:



- ❹ Create a **new class in the *View* folder called `BeeHelper`**. Make sure it's a static class, because it'll have only static methods to help your ViewModel manage its bees.

> **THE FACTORY METHOD PATTERN**
>
> MVVM is just one of many design patterns. One of the most common — and most useful — patterns is the factory method pattern, where you have a "factory" method that creates objects. The factory method is usually static, and the name often ends with "Factory" so it's obvious what's going on.

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media.Animation;

static class BeeHelper {
    public static AnimatedImage BeeFactory(
                double width, double height, TimeSpan flapInterval) {
        List<string> imageNames = new List<string>();
        imageNames.Add("Bee animation 1.png");
        imageNames.Add("Bee animation 2.png");
        imageNames.Add("Bee animation 3.png");
        imageNames.Add("Bee animation 4.png");

        AnimatedImage bee = new AnimatedImage(imageNames, flapInterval);
        bee.Width = width;
        bee.Height = height;
        return bee;
    }

    public static void SetBeeLocation(AnimatedImage bee, double x, double y) {
        Canvas.SetLeft(bee, x);
        Canvas.SetTop(bee, y);
    }

    public static void MakeBeeMove(AnimatedImage bee,
                                   double fromX, double toX, double y) {
        Canvas.SetTop(bee, y);
        Storyboard storyboard = new Storyboard();
        DoubleAnimation animation = new DoubleAnimation();
        Storyboard.SetTarget(animation, bee);
        Storyboard.SetTargetProperty(animation,
                                     new PropertyPath(Canvas.LeftProperty));
        animation.From = fromX;
        animation.To = toX;
        animation.Duration = TimeSpan.FromSeconds(3);
        animation.RepeatBehavior = RepeatBehavior.Forever;
        animation.AutoReverse = true;
        storyboard.Children.Add(animation);
        storyboard.Begin();
    }
}
```

This factory method creates bee controls. It makes sense to keep this in the View, because it's all UI-related code.

When you take a small block of code that's reused a lot and put it in its own (often static) method, it's sometimes called a helper method. Putting helper methods in a static class with a name that ends with "Helper" makes your code easier to read.

This is the same code that was in the page's constructor. Now it's in a static helper method.

- ❺ Here's the code for the empty `BeeViewModel` class that you added to the *ViewModel* folder. By moving the UI-specific code to the View, we can keep the code in the ViewModel simple and specific to managing bee-related logic.

This will come in handy in the last lab.

All XAML controls inherit from the `UIElement` base class in the `System.Windows` namespace. We explicitly used the namespace (`System.Windows.UIElement`) in the body of the class instead of adding a `using` statement to limit the amount of UI-related code we added to the ViewModel.

We used `UIElement` because it's the most abstract class that all the sprites extend. For some projects, a subclass like `FrameworkElement` may be more appropriate, because that's where many properties are defined, including `Width`, `Height`, `Opacity`, `HorizontalAlignment`, etc.

When the AnimatedImage control is added to the _sprites ObservableCollection that's bound to the ItemsControl's ItemsSource property, the control is added to the item panel, which is created based on the ItemsPanelTemplate.

```
using View;
using System.Collections.ObjectModel;
using System.Collections.Specialized;

class BeeViewModel {
    private readonly ObservableCollection<System.Windows.UIElement>
                _sprites = new ObservableCollection<System.Windows.UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    public BeeViewModel() {
        AnimatedImage firstBee =
            BeeHelper.BeeFactory(50, 50,
                                 TimeSpan.FromMilliseconds(50));
        _sprites.Add(firstBee);

        AnimatedImage secondBee =
            BeeHelper.BeeFactory(200, 200, TimeSpan.FromMilliseconds(10));
        _sprites.Add(secondBee);

        AnimatedImage thirdBee =
            BeeHelper.BeeFactory(300, 125, TimeSpan.FromMilliseconds(100));
        _sprites.Add(thirdBee);

        BeeHelper.MakeBeeMove(firstBee, 50, 450, 40);
        BeeHelper.SetBeeLocation(secondBee, 80, 260);
        BeeHelper.SetBeeLocation(thirdBee, 230, 100);
    }
}
```

We're taking two steps to encapsulate the Sprites property. The backing field is marked readonly so it can't be overwritten later, and we expose it as an INotifyCollectionChanged property so other classes can only observe it but not modify it.

A sprite is the term for any 2D image or animation that gets incorporated into a larger game or animation.

You're changing properties and adding animations on the controls after they were added to the `ObservableCollection`. Why does that work?

- ❻ Run your app. It should look exactly the same as before, but now the behavior is split across the layers, with UI-specific code in the View and code that deals with bees and moving in the ViewModel.

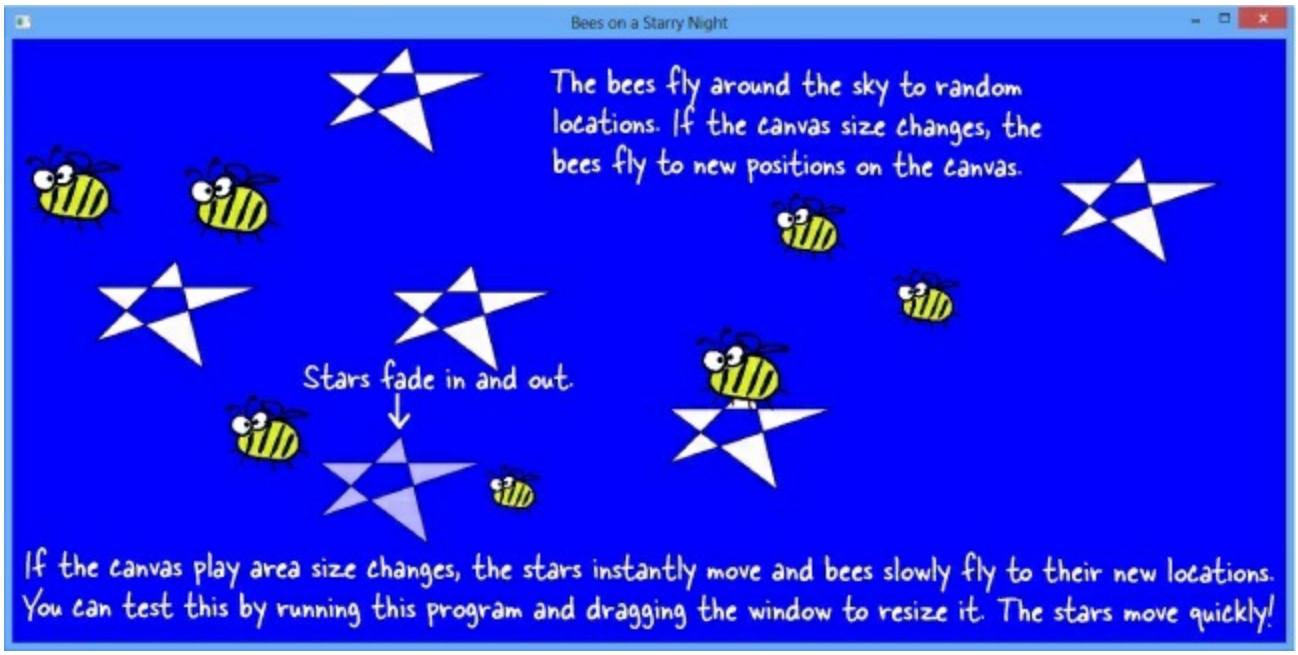
THE READONLY KEYWORD

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 be modified only in its declaration or in the constructor.

LONG EXERCISE

This is the last exercise in the book. Your job is to build a program that animates bees and stars. There's a lot of code to write, but you're up to the task...and once you have this working, you'll have all the tools you need to build a complete video game. *(Can you guess what's in Lab #3?)*

- **❶ HERE'S THE APP YOU'LL CREATE.**
  Bees with flapping wings fly around a dark blue canvas, while behind them, stars fade in and out. You'll build a View that contains the bees, stars, and page to display them; a Model that keeps track of where they are and fires off events when bees move or stars change; and a ViewModel to connect the two together.



- **❷ CREATE A NEW WPF APPLICATION PROJECT.**
  Create a new project called StarryNight. Next, **add the *Model*, *View*, and *ViewModel* folders**. Once that's done, you'll need to add **an empty class called `BeeStarViewModel`** to the *ViewModel* folder.
- **❸ CREATE A NEW WINDOW IN THE VIEW FOLDER.**
  Delete *MainWindow.xaml*. Then **add a window in the *View* folder** called *BeesOnAStarryNight. xaml*. Add the namespace to the top-level tag in the *BeesOnAStarryNight.xaml* (it should match your project's name, `StarryNight`):

```
xmlns:viewmodel="clr-namespace:StarryNight.ViewModel"
```

Add the ViewModel as a static resource and change the page name:

```
<Window.Resources>
    <viewmodel:BeeStarViewModel x:Key="viewModel"/>
</Window.Resources>
```

The XAML for the page is **exactly the same** as *FlyingBees.xaml* in the last project, except the Canvas control's background is **`Blue`** and it has a `SizeChanged` event handler:

```
<Canvas Background="Blue" SizeChanged="SizeChangedHandler" />
```

The **`SizeChanged`** event is fired when a control changes size, with **`EventArgs`** properties for the new size.

Then modify the `<Application>` tag in *App.xaml* so the application starts with the new window:

```
StartupUri="View\BeesOnAStarryNight.xaml"
```

Visual Studio comes with a fantastic tool to help you experiment with shapes! Fire up Blend for Visual Studio 2013 and use the pen, pencil, and toolbox to create XAML shapes that you can copy and paste into your C# projects.

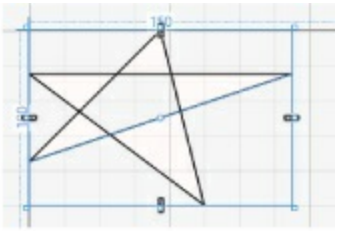
> **NOTE**
>
> The code in step 4 won't compile until you add the PlayAreaSize property to the ViewModel in step 9. You can use the IDE to generate a property stub for it for now.

- **❹ ADD CODE-BEHIND FOR THE PAGE AND THE APP.**
  Add the `SizeChanged` event handler to *BeesOnAStarryNight.xaml.cs* in the *View* folder:

```
ViewModel.BeeStarViewModel viewModel;

public BeesOnAStarryNight() {
    InitializeComponent();

    viewModel = FindResource("viewModel") as ViewModel.BeeStarViewModel;
}

private void SizeChangedHandler(object sender, SizeChangedEventArgs e) {
    viewModel.PlayAreaSize = new Size(e.NewSize.Width, e.NewSize.Height);
}
```

- **❺ ADD THE ANIMATEDIMAGE CONTROL TO THE VIEW FOLDER.**
  Go back to the *View* folder and add the AnimatedImage control. This is exactly the same control from earlier in the chapter. Make sure you **add the image files** for the animation frames to the project and update each file's Build Action to `Content` and its Copy to Output Directory to `Copy always`.
- **❻ ADD A USER CONTROL CALLED STARCONTROL TO THE VIEW FOLDER.**
  This control draws a star. It also has two storyboards, one to fade in and one to fade out. **Add methods called `FadeIn()` and `FadeOut()`** to the code-behind to trigger the storyboards.

> **NOTE**
>
> **A Polygon control uses a set of points to draw a polygon. This UserControl uses it to draw a star.**



```
<UserControl
    // The usual XAML code that the IDE generates is fine,
    // no extra namespaces are needed for this User Control.
    >

    <UserControl.Resources>
        <Storyboard x:Key="fadeInStoryboard">
            <DoubleAnimation From="0" To="1" Storyboard.TargetName="starPolygon"
                             Storyboard.TargetProperty="Opacity" Duration="0:0:1.5" />
        </Storyboard>
        <Storyboard x:Key="fadeOutStoryboard">
           <DoubleAnimation From="1" To="0" Storyboard.TargetName="starPolygon"
                            Storyboard.TargetProperty="Opacity" Duration="0:0:1.5" />
        </Storyboard>
    </UserControl.Resources>

    <Grid>
        <Polygon Points="0,75 75,0 100,100 0,25 150,25" Fill="Snow"
                 Stroke="Black" x:Name="starPolygon"/>
    </Grid>
</UserControl>
```

You'll need to add public FadeIn() and FadeOut() methods to the code-behind that starts these storyboards. That's how the stars will fade in and out.

This polygon draws the star. You can replace it with other shapes to experiment with how they work.

There are even more shapes beyond ellipses, rectangles, and polygons: http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh465055.aspx

- **❼ ADD THE BEESTARHELPER CLASS TO THE VIEW.**
  Here's a useful helper class. It's got some familiar tools and a couple of new ones. Put it in the *View* folder.

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media.Animation;
using System.Windows.Shapes;

static class BeeStarHelper {
    public static AnimatedImage BeeFactory(double width, double height, TimeSpan flapInterval) {
        List<string> imageNames = new List<string>();
        imageNames.Add("Bee animation 1.png");
        imageNames.Add("Bee animation 2.png");
        imageNames.Add("Bee animation 3.png");
        imageNames.Add("Bee animation 4.png");

        AnimatedImage bee = new AnimatedImage(imageNames, flapInterval);
        bee.Width = width;
        bee.Height = height;
        return bee;
    }

    public static void SetCanvasLocation(UIElement control, double x, double y) {
        Canvas.SetLeft(control, x);
        Canvas.SetTop(control, y);
    }

    public static void MoveElementOnCanvas(UIElement uiElement, double toX, double toY) {
        double fromX = Canvas.GetLeft(uiElement);
        double fromY = Canvas.GetTop(uiElement);

        Storyboard storyboard = new Storyboard();
        DoubleAnimation animationX = CreateDoubleAnimation(uiElement,
                    fromX, toX, new PropertyPath(Canvas.LeftProperty));
        DoubleAnimation animationY = CreateDoubleAnimation(uiElement,
                    fromY, toY, new PropertyPath(Canvas.TopProperty));
        storyboard.Children.Add(animationX);
        storyboard.Children.Add(animationY);
        storyboard.Begin();
    }

    public static DoubleAnimation CreateDoubleAnimation(UIElement uiElement,
                                    double from, double to, PropertyPath propertyToAnimate) {
        DoubleAnimation animation = new DoubleAnimation();
        Storyboard.SetTarget(animation, uiElement);
        Storyboard.SetTargetProperty(animation, propertyToAnimate);
        animation.From = from;
        animation.To = to;
        animation.Duration = TimeSpan.FromSeconds(3);
        return animation;
    }

    public static void SendToBack(StarControl newStar)
    {
        Canvas.SetZIndex(newStar, -1000);
    }
}
```

Canvas has `SetLeft()` and `GetLeft()` methods to set and get the X position of a control. The `SetTop()` and `GetTop()` methods set and get the Y position. They work even after a control is added to the Canvas.

We added a helper called CreateDoubleAnimation() that creates a three-second DoubleAnimation. This method uses it to move a UIElement from its current location to a new point by animating its Canvas.Left and Canvas.Top properties.

"Z Index" means the order the controls are layered on a panel. A control with a higher Z index is drawn on top of one with a lower Z index.

- **❽ ADD THE BEE, STAR, AND EVENTARGS CLASSES TO THE MODEL.**

Your model needs to keep track of the bees' positions and sizes, and the stars' positions, and it will fire off events so the ViewModel knows whenever there's a change to a bee or a star.

```
using System.Windows;
class Bee {
    public Point Location { get; set; }
    public Size Size { get; set; }
    public Rect Position { get { return new Rect(Location, Size); } }
    public double Width { get { return Position.Width; } }
    public double Height { get { return Position.Height; } }

    public Bee(Point location, Size size) {
        Location = location;
        Size = size;
    }
}
```

```
using System.Windows;
class Star {
    public Point Location {
        get; set;
    }

    public Star(Point location) {
        Location = location;
    }
}
```

Once you get your program working, try adding a Boolean Rotating property to the Star class and use it to make some of your stars slowly spin around.

```
using System.Windows;
class BeeMovedEventArgs : EventArgs {
    public Bee BeeThatMoved { get; private set; }
    public double X { get; private set; }
    public double Y { get; private set; }

    public BeeMovedEventArgs(Bee beeThatMoved, double x, double y) {
        BeeThatMoved = beeThatMoved;
        X = x;
        Y = y;
    }
}
```

The model will fire events that use these EventArgs to tell the ViewModel when changes happen.

```
using System.Windows;
class StarChangedEventArgs : EventArgs {
    public Star StarThatChanged { get; private set; }
    public bool Removed { get; private set; }

    public StarChangedEventArgs(Star starThatChanged, bool removed) {
        StarThatChanged = starThatChanged;
        Removed = removed;
    }
}
```

THE POINT, SIZE, AND RECT STRUCTS

The System.Windows namespace has several very useful structs. Point uses X and Y double properties to store a set of coordinates. Size has two double properties too, Width and Height, and also a special Empty value. Rect stores two coordinates for the top-left and bottom-right corner of a rectangle. It has a lot of useful methods to find its width, height, intersection with other Rects, and more.

The Rect struct has several overloaded constructors, and methods that let you extract its width, height, size, and location (either as a Point or individual X and Y double coordinates). The Points property on the Polygon control is a collection of Point structs.

- **⑨ ADD THE BEESTARMODEL CLASS TO THE MODEL.**
  We've filled in the private fields and a couple of useful methods. Your job is to finish building the `BeeStarModel` class.

```
using System.Windows;

class BeeStarModel {
    public static readonly Size StarSize = new Size(150, 100);

    private readonly Dictionary<Bee, Point> _bees = new Dictionary<Bee, Point>();
    private readonly Dictionary<Star, Point> _stars = new Dictionary<Star, Point>();
    private Random _random = new Random();

    public BeeStarModel() {
        _playAreaSize = Size.Empty;
    }

    public void Update() {
        MoveOneBee();
        AddOrRemoveAStar();
    }

    private static bool RectsOverlap(Rect r1, Rect r2) {
        r1.Intersect(r2);
        if (r1.Width > 0 || r1.Height > 0)
            return true;
        return false;
    }

    public Size PlayAreaSize {
        // Add a backing field, and have the set accessor call CreateBees() and CreateStars()
    }
    private void CreateBees() {
        // If the play area is empty, return. If there are already bees, move each of them.
        // Otherwise, create between 5 and 15 randomly sized bees (40 to 150 pixels), add
        // it to the _bees collection, and fire the BeeMoved event.
    }
    private void CreateStars() {
        // If the play area is empty, return. If there are already stars,
        // set each star's location to a new point and fire the StarChanged
        // event, otherwise call CreateAStar() between 5 and 10 times.
    }
    private void CreateAStar() {
        // Find a new non-overlapping point, add a new Star object to the
        // _stars collection, and fire the StarChanged event.
    }
    private Point FindNonOverlappingPoint(Size size) {
        // Find the upper-left corner of a rectangle that doesn't overlap any bees or stars.
        // You'll need to try random Rects, then use LINQ queries to find any bees or stars
        // that overlap (the RectsOverlap() method will be useful).
    }
    private void MoveOneBee(Bee bee = null) {
        // If there are no bees, return. If the bee parameter is null, choose a random bee,
        // otherwise use the bee argument. Then find a new non-overlapping point, update the bee's
        // location, update the _bees collection, and then fire the OnBeeMoved event.
    }
    private void AddOrRemoveAStar() {
        // Flip a coin (_random.Next(2) == 0) and either create a star using CreateAStar() or
        // remove a star and fire OnStarChanged. Always create a star if there are <= 5, remove
        // one if >= 20. _stars.Keys.ToList()[_random.Next(_stars.Count)] will find a random star.
    }
    // You'll need to add the BeeMoved and StarChanged events and methods to call them.
    // They use the BeeMovedEventArgs and StarChangedEventArgs classes.
}
```

You can use readonly to create a constant struct value.

Size.Empty is a value of Size that's reserved for an empty size. You'll use it only to create bees and stars when the play area is resized.

The ViewModel will use a timer to call this Update() method periodically.

This method checks two Rect structs and returns true if they overlap each other using the Rect.Intersect() method.

PlayAreaSize is a property.

If the method's tried 1,000 random locations and hasn't found one that doesn't overlap, the play area has probably run out of space, so just return any point.

You can debug your app with the simulator to make sure it works with different screen sizes and orientations.

- **⑩ ADD THE BEESTARVIEWMODEL CLASS TO THE VIEWMODEL.**
  Fill in the commented methods. You'll need to look closely at how the Model works and what the View expects. The helper

methods will also come in very handy.

We wanted to make sure that `DispatcherTimer` and `UIElement` are the only classes from the `Windows.UI.Xaml` namespace that we used in the ViewModel. The `using` keyword lets you use = to declare a single member in another namespace.

```
using View;
using Model;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using System.Windows;
using DispatcherTimer = Windows.UI.Xaml.DispatcherTimer;
using UIElement = Windows.UI.Xaml.UIElement;

class BeeStarViewModel {
    private readonly ObservableCollection<UIElement>
                _sprites = new ObservableCollection<UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    private readonly Dictionary<Star, StarControl> _stars = new Dictionary<Star, StarControl>();
    private readonly List<StarControl> _fadedStars = new List<StarControl>();

    private BeeStarModel _model = new BeeStarModel();

    private readonly Dictionary<Bee, AnimatedImage> _bees = new Dictionary<Bee, AnimatedImage>();

    private DispatcherTimer _timer = new DispatcherTimer();

    public Size PlayAreaSize { /* get and set accessors return and set _model.PlayAreaSize */ }

    public BeeStarViewModel() {
        // Hook up the event handlers to the BeeStarModel's BeeMoved and StarChanged events,
        // and start the timer ticking every two seconds.
    }
    void timer_Tick(object sender, object e) {
        // Every time the timer ticks, find all StarControl references in the _fadedStars
        // collection and remove each of them from _sprites, then call the BeeViewModel's
        // Update() method to tell it to update itself.
    }
    void BeeMovedHandler(object sender, BeeMovedEventArgs e) {
        // The _bees dictionary maps Bee objects in the Model to AnimatedImage controls
        // in the view. When a bee is moved, the BeeViewModel fires its BeeMoved event to
        // tell anyone listening which bee moved and its new location. If the _bees
        // dictionary doesn't already contain an AnimatedImage control for the bee, it needs
        // to create a new one, set its canvas location, and update both _bees and _sprites.
        // If the _bees dictionary already has it, then we just need to look up the corresponding
        // AnimatedImage control and move it on the canvas to its new location with an animation.
    }
    void StarChangedHandler(object sender, StarChangedEventArgs e) {
        // The _stars dictionary works just like the _bees one, except that it maps Star objects
        // to their corresponding StarControl controls. The EventArgs contains references to
        // the Star object (which has a Location property) and a Boolean to tell you if the star
        // was removed. If it is then we want it to fade out, so remove it from _stars, add it
        // to _fadedStars, and call its FadeOut() method (it'll be removed from _sprites the next
        // time the Update() method is called, which is why we set the timer's tick interval to
        // be greater than the StarControl's fade out animation).
        //
        // If the star is not being removed, then check to see if _stars contains it - if so, get
        // the StarControl reference; if not, you'll need to create a new StarControl, fade it in,
        // add it to _sprites, and send it to back so the bees can fly in front of it. Then set
        // the canvas location for the StarControl.
    }
}
```

When you set the new Canvas location, the control is updated—even if it's already on the Canvas. This is how the stars move themselves around when the play area is resized.

LONG EXERCISE SOLUTION

Here are the filled-in methods in the `BeeStarModel` class.

```
using System.Windows;

class BeeStarModel {
    public static readonly Size StarSize = new Size(150, 100);

    private readonly Dictionary<Bee, Point> _bees = new Dictionary<Bee, Point>();
    private readonly Dictionary<Star, Point> _stars = new Dictionary<Star, Point>();
    private Random _random = new Random();

    public BeeStarModel() {
        _playAreaSize = Size.Empty;
    }

    public void Update() {
        MoveOneBee();
        AddOrRemoveAStar();
    }

    private static bool RectsOverlap(Rect r1, Rect r2) {
        r1.Intersect(r2);
        if (r1.Width > 0 || r1.Height > 0)
            return true;
        return false;
    }

    private Size _playAreaSize;
    public Size PlayAreaSize {
        get { return _playAreaSize; }
        set
        {
            _playAreaSize = value;
            CreateBees();
            CreateStars();
        }
    }

    private void CreateBees() {
        if (PlayAreaSize == Size.Empty) return;

        if (_bees.Count() > 0) {
            List<Bee> allBees = _bees.Keys.ToList();
            foreach (Bee bee in allBees)
                MoveOneBee(bee);
        } else {
            int beeCount = _random.Next(5, 10);
            for (int i = 0; i < beeCount; i++) {
                int s = _random.Next(50, 100);
                Size beeSize = new Size(s, s);
                Point newLocation = FindNonOverlappingPoint(beeSize);
                Bee newBee = new Bee(newLocation, beeSize);
                _bees[newBee] = new Point(newLocation.X, newLocation.Y);
                OnBeeMoved(newBee, newLocation.X, newLocation.Y);
            }
        }
    }
```

We gave these to you.

Whenever the PlayAreaSize property changes, the Model updates the _playAreaSize backing field and then calls CreateBees() and CreateStars(). This lets the ViewModel tell the Model to adjust itself whenever the size changes—which will happen if you run the program on a tablet and change the orientation.

If there are already bees, move each of them. MoveOneBee() will find a new nonoverlapping location for each bee and fire a BeeMoved event.

If there aren't any bees in the model yet, this creates new Bee objects and sets their locations. Any time a bee is added or changes, we need to fire a BeeMoved event.

```
private void CreateStars() {
    if (PlayAreaSize == Size.Empty) return;

    if (_stars.Count > 0) {
        foreach (Star star in _stars.Keys) {
            star.Location = FindNonOverlappingPoint(StarSize);
            OnStarChanged(star, false);
        }
    } else {
        int starCount = _random.Next(5, 10);
        for (int i = 0; i < starCount; i++)
            CreateAStar();
    }
}

private void CreateAStar() {
    Point newLocation = FindNonOverlappingPoint(StarSize);
    Star newStar = new Star(newLocation);
    _stars[newStar] = new Point(newLocation.X, newLocation.Y);
    OnStarChanged(newStar, false);
}

private Point FindNonOverlappingPoint(Size size) {
    Rect newRect = new Rect();
    bool noOverlap = false;
    int count = 0;
    while (!noOverlap) {
        newRect = new Rect(_random.Next((int)PlayAreaSize.Width - 150),
            _random.Next((int)PlayAreaSize.Height - 150),
            size.Width, size.Height);

        var overlappingBees =
            from bee in _bees.Keys
            where RectsOverlap(bee.Position, newRect)
            select bee;

        var overlappingStars =
            from star in _stars.Keys
            where RectsOverlap(
                new Rect(star.Location.X, star.Location.Y, StarSize.Width, StarSize.Height),
                newRect)
            select star;

        if ((overlappingBees.Count() + overlappingStars.Count() == 0) || (count++ > 1000))
            noOverlap = true;
    }
    return new Point(newRect.X, newRect.Y);
}

private void MoveOneBee(Bee bee = null) {
    if (_bees.Keys.Count() == 0) return;
    if (bee == null) {
        int beeCount = _stars.Count;
        List<Bee> bees = _bees.Keys.ToList();
        bee = bees[_random.Next(bees.Count)];
    }
    bee.Location = FindNonOverlappingPoint(bee.Size);
    _bees[bee] = bee.Location;
    OnBeeMoved(bee, bee.Location.X, bee.Location.Y);
}
```

If there are already stars, we just set each existing star's location to a new point on the PlayArea and fire the StarChanged event. It's up to the ViewModel to handle that event and move the corresponding control.

This creates a random Rect and then checks if it overlaps. We gave it a 250-pixel gap on the right and a 150-pixel gap on the bottom so the stars and bees don't leave the play area.

These LINQ queries call RectsOverlap() to find any bees or stars that overlap the new Rect. If either return value has a count, the new Rect overlaps something.

If this iterated 1,000 times, it means we're probably out of nonoverlapping spots in the play area and need to break out of an infinite loop.

LONG EXERCISE SOLUTION

The last few members of the `BeeStarModel` class.

Flip a coin by choosing either 0 or 1 at random, but always create a star if there are under 5 and remove if 20 or more.

```
    private void AddOrRemoveAStar() {
        if (((_random.Next(2) == 0) || (_stars.Count <= 5)) && (_stars.Count < 20 ))
            CreateAStar();
        else {
            Star starToRemove = _stars.Keys.ToList()[_random.Next(_stars.Count)];
            _stars.Remove(starToRemove);
            OnStarChanged(starToRemove, true);
        }
    }

    public event EventHandler<BeeMovedEventArgs> BeeMoved;

    private void OnBeeMoved(Bee beeThatMoved, double x, double y)
    {
        EventHandler<BeeMovedEventArgs> beeMoved = BeeMoved;
        if (beeMoved != null)
        {
            beeMoved(this, new BeeMovedEventArgs(beeThatMoved, x, y));
        }
    }

    public event EventHandler<StarChangedEventArgs> StarChanged;

    private void OnStarChanged(Star starThatChanged, bool removed)
    {
        EventHandler<StarChangedEventArgs> starChanged = StarChanged;
        if (starChanged != null)
        {
            starChanged(this, new StarChangedEventArgs(starThatChanged, removed));
        }
    }
}
```

Every time the Update() method is called, we want to either add or remove a star. The CreateAStar() method already creates stars. If we're removing a star, we just remove it from _stars and fire a StarChanged event.

These are typical event handlers and methods to fire them.

Here are the filled-in methods of the `BeeStarViewModel` class.

```
using View;
using Model;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using System.Windows;
using DispatcherTimer = System.Windows.Threading.DispatcherTimer;
using UIElement = System.Windows.UIElement;

class BeeStarViewModel {
    private readonly ObservableCollection<UIElement>
                _sprites = new ObservableCollection<UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    private readonly Dictionary<Star, StarControl> _stars = new Dictionary<Star, StarControl>();
    private readonly List<StarControl> _fadedStars = new List<StarControl>();

    private BeeStarModel _model = new BeeStarModel();

    private readonly Dictionary<Bee, AnimatedImage> _bees
                                        = new Dictionary<Bee, AnimatedImage>();

    private DispatcherTimer _timer = new DispatcherTimer();
```

We gave these to you.

```
    public Size PlayAreaSize {
        get { return _model.PlayAreaSize; }
        set { _model.PlayAreaSize = value; }
    }

    public BeeStarViewModel() {
        _model.BeeMoved += BeeMovedHandler;
        _model.StarChanged += StarChangedHandler;

        _timer.Interval = TimeSpan.FromSeconds(2);
        _timer.Tick += timer_Tick;
        _timer.Start();
    }

    void timer_Tick(object sender, object e) {
        foreach (StarControl starControl in _fadedStars)
            _sprites.Remove(starControl);

        _model.Update();
    }

    void BeeMovedHandler(object sender, BeeMovedEventArgs e) {
        if (!_bees.ContainsKey(e.BeeThatMoved)) {
            AnimatedImage beeControl = BeeStarHelper.BeeFactory(
                        e.BeeThatMoved.Width, e.BeeThatMoved.Height, TimeSpan.FromMilliseconds(20));
            BeeStarHelper.SetCanvasLocation(beeControl, e.X, e.Y);
            _bees[e.BeeThatMoved] = beeControl;
            _sprites.Add(beeControl);
        } else {
            AnimatedImage beeControl = _bees[e.BeeThatMoved];
            BeeStarHelper.MoveElementOnCanvas(beeControl, e.X, e.Y);
        }
    }

    void StarChangedHandler(object sender, StarChangedEventArgs e) {
        if (e.Removed) {
            StarControl starControl = _stars[e.StarThatChanged];
            _stars.Remove(e.StarThatChanged);
            _fadedStars.Add(starControl);
            starControl.FadeOut();
        } else {
            StarControl newStar;
            if (_stars.ContainsKey(e.StarThatChanged))
                newStar = _stars[e.StarThatChanged];
            else {
                newStar = new StarControl();
                _stars[e.StarThatChanged] = newStar;
                newStar.FadeIn();
                BeeStarHelper.SendToBack(newStar);
                _sprites.Add(newStar);
            }
            BeeStarHelper.SetCanvasLocation(
                        newStar, e.StarThatChanged.Location.X, e.StarThatChanged.Location.Y);
        }
    }
}
```

The _fadedStars collection contains the controls that are currently fading and will be removed the next time the ViewModel's Update() method is called.

If a star is being added, it needs to have its FadeIn() method called. If it's already there, it's just being moved because the play area size changed. Either way, we want to move it to its new location on the Canvas.

LONG EXERCISE SOLUTION

Here are the methods for the StarControl code-behind:

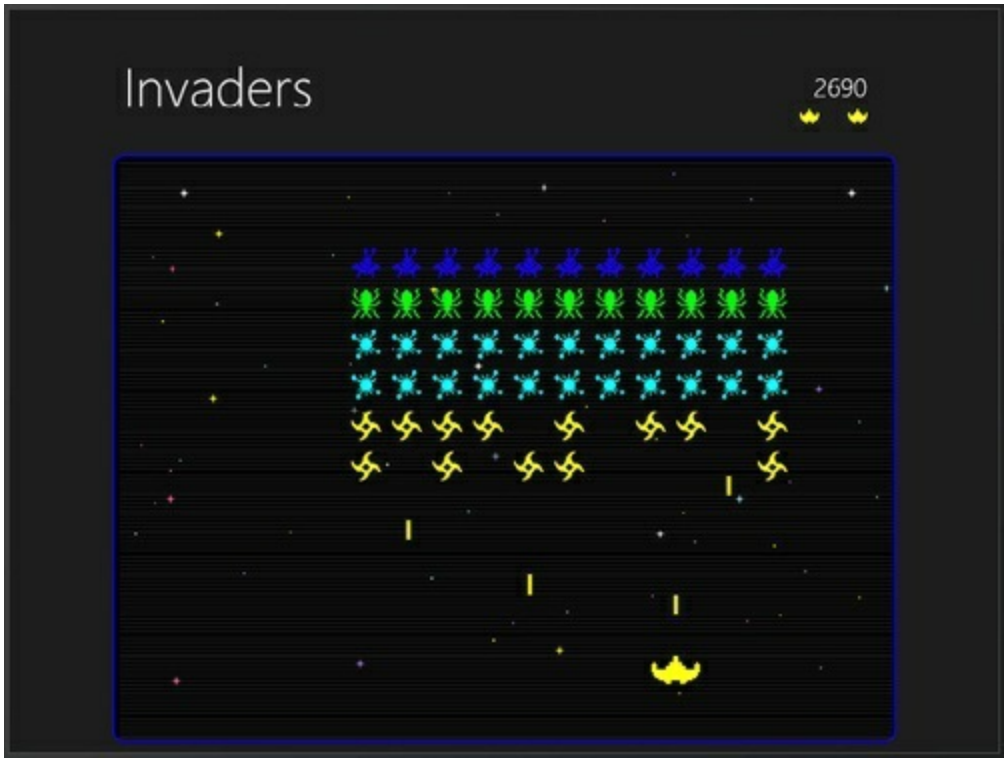
```
using System.Windows.Media.Animation;

public partial class StarControl : UserControl {
    public StarControl()
    {
        InitializeComponent();
    }

    public void FadeIn() {
        Storyboard fadeInStoryboard = FindResource("fadeInStoryboard") as Storyboard;
        fadeInStoryboard.Begin();
    }

    public void FadeOut() {
        Storyboard fadeOutStoryboard = FindResource("fadeOutStoryboard") as Storyboard;
        fadeOutStoryboard.Begin();
    }
}
```

The ViewModel's `PlayAreaSize` property just passes through to the property on the Model — but the Model's `PlayAreaSize` set accessor calls methods that fire `BeeMoved` and `StarChanged` events. So when the screen resolution changes: 1) the Canvas fires its `SizeChanged` event, which 2) updates the ViewModel's `PlayAreaSize` property, which 3) updates the Model's property, which 4) calls methods to update bees and stars, which 5) fire `BeeMoved` and `StarChanged` events, which 6) trigger the ViewModel's event handlers, which 7) update the `Sprites` collection, which 8) update the controls on the Canvas. This is an example of loose coupling, where there's no single, central object to coordinate things. This is a very stable way to build software because each object doesn't need to have explicit knowledge of how the other objects work. It just needs to know one small job: handle an event, fire an event, call a method, set a property, etc.

> **NOTE**
>
> If you've done a good job with separation of concerns, your designs often tend to naturally end up being loosely coupled.

You've got all the tools to do Lab #3 and build Invaders!

We saved the best for last. In the last lab in the book, you'll build your own version of Space Invaders, the grandfather of video games. And while the lab is aimed at Windows Store apps, if you finished the *Bees on a Starry Night* project — and you understood it all — then you have the knowledge and know-how to build a WPF version of the Invaders game. Almost everything in the lab applies to WPF. The only thing that's different is how the user controls the ship. Windows Store apps have advanced gesture events that process touch and mouse input, but WPF windows don't support those events. You'll need to use the WPF Window object's KeyUp and KeyDown events. Luckily, you've already got a good example. Flip back to the Key Game in Chapter 4 — your Invaders game can handle keystrokes in exactly the same way.


Invaders
2690

Congratulations! (But you're not done yet...)

Did you finish that last exercise? Did you understand everything that was going on? If so, then **congratulations** — you've learned a whole lot of C#, and probably in less time than you'd expected! The world of programming awaits you.

Still, there are a few things that you should do before you move on to the last lab, if you really want to make sure all the information you put in your brain stays there.

Take one last look through Save the Humans.

If you did everything we asked you to do, you've built *Save the Humans* twice, once at the beginning of the book and again before you started Chapter 10. Even the second time around, there were parts of it that seemed like magic. But when it comes to programming, **there is no magic**. So take one last pass through the code you built. You'll be surprised at how much you understand! There's almost nothing that seals a lesson into your brain like positive reinforcement.

Talk about it with your friends.

Humans are social animals, and when you talk through things you've learned with your social circle you do a better job of retaining them. And these days, "talking" means social networking, too! Plus, you've really accomplished something here. Go ahead and **claim your bragging rights**!

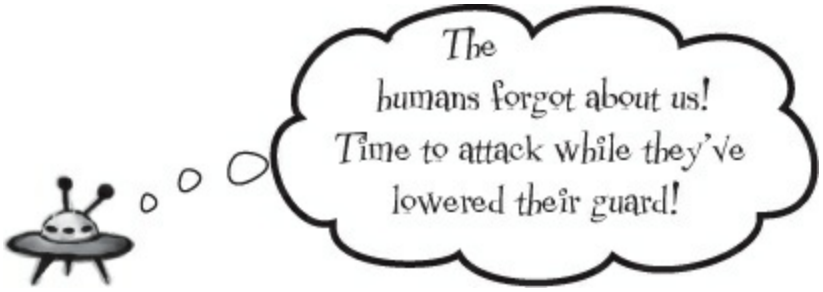
Take a break. Even better, take a nap.

Your brain has absorbed a lot of information, and sometimes the best thing you can do to "lock in" all that new knowledge is to sleep on it. There's a lot of neuroscience research that shows that information absorption is significantly improved **after a good night's sleep**. So give your brain a well-deserved rest!

When it comes to programming, there is no magic. Every program works because it was built to work, and all code can be understood.

NOTE

...but it's a lot easier to understand code if the programmer used good design patterns and object-oriented programming principles.



Index

A NOTE ON THE DIGITAL INDEX

A link in an index entry is displayed as the section title in which that entry appears. Because some sections have multiple index markers, it is not unusual for an entry to have several links to the same section. Clicking on any link will take you directly to the place in the text in which the marker appears.

Symbols

& (ampersand), Set up conditions and see if they're true, Wait, wait! What did that say?, ...more basics..., ...more basics...

& (logical AND) operator, ...more basics...

&& operator, Set up conditions and see if they're true, Wait, wait! What did that say?, ...more basics...

* (asterisk), A few useful types, Let's create some instances!, When you cast a value that's too big, C# will adjust it automatically, When you call a method, the arguments must be compatible with the types of the parameters, Combining = with an operator

*= (multiplication and assignment) operator, A few useful types, Let's create some instances!, When you call a method, the arguments must be compatible with the types of the parameters, Combining = with an operator

multiplication operator, converting types, When you cast a value that's too big, C# will adjust it automatically

@ (at sign), preceding filenames, Write text to a file in three simple steps, Things you can do with Directory:

~ (bitwise complement) operator, ...more basics...

: (colon), Use a colon to inherit from a base class, Now you can create an instance of NectarStinger that does both jobs

implementing an interface, Now you can create an instance of NectarStinger that does both jobs

using to inherit from base class, Use a colon to inherit from a base class

?: (conditional) operator, Add the stopwatch ViewModel, ...more basics..., #6. Equality, IEquatable, and Equals()

{ } (curly brackets), Let's take a closer look at your code, Let's take a closer look at your code, Loops perform an action over and over, Make each button do something, Create a project for your guys, Build the Part y Planner version 2.0, The Swindler launches another diabolical plan

code for classes or methods in, Make each button do something

grouping sttements into code blocks, Let's take a closer look at your code, Let's take a closer look at your code

leaving out for code blocks, Build the Part y Planner version 2.0

matching up using the IDE, Loops perform an action over and over

using to pass variables to string in StreamWriter, The Swindler launches another diabolical plan

. (dot) operator, A few useful types

\\ (double backslash), escaping backslash in strings, Things you can do with Directory:

= (equals sign), You have to assign values to variables before you use them, if/else statements make decisions, if/else statements make decisions, Set up conditions and see if they're true, Use logical operators to check conditions, ...more basics..., #3. Namespaces and assemblies, #6. Equality, IEquatable, and Equals()

assignment operator, You have to assign values to variables before you use them, if/else statements make decisions, ...more basics...

combining with logical operators, #3. Namespaces and assemblies

== (equality) operator, Set up conditions and see if they're true, #6. Equality, IEquatable, and Equals()

= versus == operator, if/else statements make decisions, Use logical operators to check conditions

! (exclamation mark), A few useful types, Set up conditions and see if they're true, When you call a method, the arguments must be compatible with the types of the parameters, Controls are objects, just like any other object, How you'll build the beehive management system

!= (inequality) operator, Set up conditions and see if they're true, Controls are objects, just like any other object

NOT operator, A few useful types, When you call a method, the arguments must be compatible with the types of the parameters, How you'll build the beehive management system

> (greater than) operator, Set up conditions and see if they're true

<< (left shift) operator, #3. Namespaces and assemblies

< (less than) operator, Set up conditions and see if they're true

- (minus sign), A few useful types, A few useful types, When you cast a value that's too big, C# will adjust it automatically, When you call a method, the arguments must be compatible with the types of the parameters, Combining = with an operator

- - (decrement) operator, A few useful types

-= (subtraction and assignment) operator, When you call a method, the arguments must be compatible with the types of the parameters, Combining = with an operator

subtraction operator, A few useful types, When you cast a value that's too big, C# will adjust it automatically

\n (line feed character), Make each button do something, So what did you just build?, The variable's type determines what kind of data it can store, Build a program that uses a dictionary, Write text to a file in three simple steps, Things you can do with Directory:

?? (null coalescing) operator, ...more basics...

=> operator, #9. Anonymous types, anonymous methods, and lambda expressions

in lambda expressions, #9. Anonymous types, anonymous methods, and lambda expressions

| (pipe symbol), Set up conditions and see if they're true, You can help Brian out by building a program to manage his excuses, ...more basics..., ...more basics...

logical OR operator, ...more basics...

|| (OR) operator, Set up conditions and see if they're true, You can help Brian out by building a program to manage his excuses, ...more basics...

+ (plus sign), Add timers to manage the gameplay, A few useful types, A few useful types, A few useful types, A few useful types, When you cast a value that's too big, C# will adjust it automatically, Use a standard name when you add a method to raise an event, ...more basics..., #3. Namespaces and assemblies

+= (addition and assignment) operator, Add timers to manage the gameplay, A few useful types, Use a standard name when you add a method to raise an event

addition operator, A few useful types

addition or string concatentation, conversion of types with, When you cast a value that's too big, C# will adjust it automatically

++ (increment) operator, A few useful types, ...more basics...

string concatenation operator, A few useful types, #3. Namespaces and assemblies

>> (right shift) operator, #3. Namespaces and assemblies

\r (return character), Build a program that uses a dictionary, Things you can do with Directory:

; (semicolon), ending statements, The IDE helps you code, Make each button do something

/ (slash), A few useful types, Use the debugger to see your variables change, Make each button do something, Rebuild your app for Windows Desktop, Wait, wait! What did that say?, When you cast a value that's too big, C# will adjust it automatically, #2. The Basics, ...more basics..., Use the debugger to see your variables change

/* and */ enclosing multiline comments, ...more basics...

comments beginning with //, Make each button do something

comments beginning with ///, Rebuild your app for Windows Desktop, Wait, wait! What did that say?

comments surrounded with /* and */ or //, Use the debugger to see your variables change, Use the debugger to see your variables change

division operator, A few useful types

division operator, converting types, When you cast a value that's too big, C# will adjust it automatically

/// (triple-slash), denoting XML comments, #2. The Basics

[ ] (square brackets), A special case: arrays, Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!, #7. Using yield return to create enumerable objects

using to access elements, Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!

using to retrieve object from list, array, or dictionary, #7. Using yield return to create enumerable objects

usng to declare and initialize arrays, A special case: arrays

\t (tab character), The variable's type determines what kind of data it can store, Write text to a file in three simple steps, Things you can do with Directory:

^ (XOR) operator, ...more basics..., #6. Equality, IEquatable, and Equals()

A

About popup control, using Settings charm to open, Use the Settings charm to open a SettingsFlyout

abstract classes, An abstract class is like a cross between a class and an interface, An abstract class is like a cross between a class and an interface, An abstract method doesn't have a body

Fireside Chat, An abstract method doesn't have a body

usefulness, An abstract class is like a cross between a class and an interface

abstraction, Build up your class model by starting general and getting more specific, An abstract method doesn't have a body

as principle of OOP, An abstract method doesn't have a body

general versus specific, Build up your class model by starting general and getting more specific

abstract keyword, An abstract method doesn't have a body

abstract methods, An abstract class is like a cross between a class and an interface, An abstract method doesn't have a body

access modifiers, There's more than just public and private, There's more than just public and

private, There’s more than just public and private, There’s more than just public and private, There’s more than just public and private, There’s more than just public and private, Access modifiers change visibility, Access modifiers change visibility, ...so what did I just do?

internal, There’s more than just public and private

private, There’s more than just public and private

protected, There’s more than just public and private

protected versus private or public, Access modifiers change visibility

public, There’s more than just public and private

scope, Access modifiers change visibility

sealed, There’s more than just public and private

addition and assignment operator (+=)., Add timers to manage the gameplay (see + (plus sign), under Symbols; compound operators)

addition operator., Add timers to manage the gameplay (see + (plus sign), under Symbols)

Adventure Game program, C# Lab: A Day at the Races (see labs, #2 The Quest)

Albahari, Ben, Did you know that C# and the .NET Framework can...

Albahari, Joe, Add semantic zoom to Jimmy’s app, #8. Refactoring, Did you know that C# and the .NET Framework can...

aliens, , Aliens attack!, Aliens attack!, Your game is now playable, It’s all Just Code: Under the hood, C# Lab Invaders

gastronomy, Aliens attack!

saving Earth from, C# Lab Invaders

allocate, defined, IDisposable makes sure your objects are disposed of properly

allocated resources, IDisposable makes sure your objects are disposed of properly

ambiguity, avoiding, An abstract method doesn’t have a body

AND operator., ...more basics... (see & (ampersand), under Symbols)

AngleConverter class, Build an analog stopwatch using the same ViewModel, Build an analog stopwatch using the same ViewModel

animal inheritance program, How would you design a zoo simulator?

animations, Add a method that does something, Use the IDE to create your own method, Fill in the code for your method, Finish the method and run your program, Wait, wait! What did that say?, Controls are objects, just like any other object, Visual states make controls respond to changes, Use DoubleAnimation to animate double values, Use object animations to animate object values, C# can build “real” animations, too, Make your bees fly around a page, Use ItemsPanelTemplate to bind controls to a Canvas, Make your bees fly around a page, Use ItemsPanelTemplate to bind controls to a Canvas

bouncing Label controls, Controls are objects, just like any other object

building program that animates bees and stars, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

building with C#, C# can build “real” animations, too, Make your bees fly around a page, Make your bees fly around a page

making bees fly around a page, Make your bees fly around a page, Make your bees fly around a page

code creating enemy bouncing animation (example), Use the IDE to create your own method, Finish the method and run your program

desktop apps, Wait, wait! What did that say?

generating method stub for AnimateEnemy( ) method (example), Add a method that does something, Fill in the code for your method

key frame, Use object animations to animate object values

using DoubleAnimation to animate double values, Use DoubleAnimation to animate double values

visual state changes for buttons, Visual states make controls respond to changes

anonymous, defined, Use the new keyword to create anonymous types

anonymous methods, #9. Anonymous types, anonymous methods, and lambda expressions

anonymous types, Use the new keyword to create anonymous types, Use the new keyword to create anonymous types, Jimmy saved a bunch of dough, #9. Anonymous types, anonymous methods, and lambda expressions

creating using new keyword, Use the new keyword to create anonymous types, Use the new keyword to create anonymous types

APIs, defined, The .NET Framework gives you the right tools for the job

AppBar controls, Build a slightly less simple text editor

AppendAllText( ) method, Use the built-in File and Directory classes to work with files and directories

Append( ), AppendFormat( ) and AppendLine( ), StringBuilder, #3. Namespaces and assemblies

Appliance project, A CoffeeMaker is also an Appliance, A CoffeeMaker is also an Appliance, Upcasting works with both objects and interfaces, Downcasting lets you turn your appliance back into a coffee maker, Upcasting and downcasting work with interfaces, too, Upcasting and downcasting work with interfaces, too

Appliance class, A CoffeeMaker is also an Appliance

downcasting, Downcasting lets you turn your appliance back into a coffee maker, Upcasting and downcasting work with interfaces, too

interfaces, Upcasting and downcasting work with interfaces, too

upcasting, Upcasting works with both objects and interfaces, Upcasting and downcasting work with interfaces, too

interfaces, Upcasting and downcasting work with interfaces, too

ApplicationData.Current.LocalFolder, Use async methods to find and open files

application life cycle, Windows Store apps, Use static resources to declare your objects in XAML

Application object, Use the IDE to explore app page navigation

application programming interfaces., The .NET Framework gives you the right tools for the job (see APIs)

AppName, changing for Windows Store app, Use properties to change how the controls look

apps., _, if/else statements make decisions, Build an app from the ground up

(see also Windows Store apps)

building from ground up, if/else statements make decisions, Build an app from the ground up

App.xaml.cs file, What you do in Visual Studio..., Use the IDE to explore app page navigation, Windows Store apps use events for process lifetime management

App.xaml file, Use the IDE to explore app page navigation, Add semantic zoom to Jimmy's app, The IDE's Split App template helps you build apps for navigating data

ArgumentException, Brian's code did something unexpected, Bees need an OutOfHoney exception

arguments, C# does some casting automatically, When you call a method, the arguments must be compatible with the types of the parameters

compatibility with types of parameters, When you call a method, the arguments must be compatible with the types of the parameters

arithmetic operators, A few useful types, When you cast a value that's too big, C# will adjust it automatically, C# does some casting automatically

automatic casting with, When you cast a value that's too big, C# will adjust it automatically, C# does some casting automatically

arrays, A special case: arrays, Arrays can contain a bunch of reference variables, too, Arrays can contain a bunch of reference variables, too, Controls are objects, just like any other object, You can find out if a class implements a certain interface with "is", We could use an array to create a deck of cards..., Arrays are hard to work with, Lists are more flexible than arrays, #7. Using yield return to create enumerable objects

containing reference variables, Arrays can contain a bunch of reference variables, too

difficulty in working with, Arrays are hard to work with

finding length, Arrays can contain a bunch of reference variables, too

of objects, Controls are objects, just like any other object

using to create deck of cards, We could use an array to create a deck of cards...

using [ ] to return object from, #7. Using yield return to create enumerable objects

versus Lists, Lists are more flexible than arrays

as keyword, is tells you what an object implements; as tells the compiler how to treat your object, Downcasting lets you turn your appliance back into a coffee maker, Upcasting and downcasting work with interfaces, too, Polymorphism means that one object can take many different forms, The stack vs. the heap: more on memory, "Captain" Amazing...not so much

illegal downcasting, Upcasting and downcasting work with interfaces, too

use with objects, "Captain" Amazing...not so much

using in downcasting, Downcasting lets you turn your appliance back into a coffee maker, Polymorphism means that one object can take many different forms

value types and, The stack vs. the heap: more on memory

assemblies, There's more than just public and private, #3. Namespaces and assemblies

Assets folder, Use the new keyword to create anonymous types

adding image files to, Use the new keyword to create anonymous types

assignment, Start with a blank application, You have to assign values to variables before you use them, Use logical operators to check conditions, Use logical operators to check conditions, ...more basics...

= operator, ...more basics...

values to variables, You have to assign values to variables before you use them

assignment operator (=), if/else statements make decisions

(see also = (equals sign), under Symbols)

asynchronous methods, Windows Store apps use await to be more responsive, Use async methods to find and open files, Use a Task to call one async method from another, C# programs can use await to be more responsive

using Task to call one from another, Use a Task to call one async method from another

using to find and open files, Use async methods to find and open files

async modifier, Windows Store apps use await to be more responsive, Build a slightly less simple text editor, Build a slightly less simple text editor, C# programs can use await to be more responsive

in OpenFile( ) and SaveFile( ) methods, Build a slightly less simple text editor

with await operator in method's delcaration, Build a slightly less simple text editor

attributes, If you want your class to be serializable, mark it with the [Serializable] attribute

automatic properties, You can find out if a class implements a certain interface with "is"

await operator, Windows Store apps use await to be more responsive, Build a slightly less simple text editor, Build a slightly less simple text editor, Use a Task to call one async method from another, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Handle exceptions with try and catch, Add process lifetime management to Jimmy's comics, C# programs can use await to be more responsive

inability to use in body of catch clause, Handle exceptions with try and catch

in OpenFile( ) and SaveFile( ) methods, Build a slightly less simple text editor

with async keyword in method declaration, Build a slightly less simple text editor

B

BackgroundWorker, using to make WinForms responsive, #4. Use BackgroundWorker to make your WinForms responsive

backing fields, Properties make encapsulation easier, Use a constructor to initialize private fields, Keep your eyes open for polymorphism in the next exercise!

Baseball Simulator project, Ever wish your objects could think for themselves?, The IDE generates event handlers for you automatically, The IDE generates event handlers for you automatically, An object can subscribe to an event..., Use a callback to control who's listening, Chapter 15, Chapter 15

callbacks, Use a callback to control who's listening

Fan class, The IDE generates event handlers for you automatically, Chapter 15

Pitcher class, The IDE generates event handlers for you automatically, Chapter 15

subscription and public events, An object can subscribe to an event...

base classes, When your classes use inheritance, you only need to write your code once, How would you design a zoo simulator?, Create the class hierarchy, Use a colon to inherit from a base class, Any place where you can use a base class, you can use one of its subclasses instead, A subclass can access its base class using the base keyword, When a base class has a constructor, your subclass needs one, too, Upcasting works with both objects and interfaces

building Animal base class for zoo simulator, How would you design a zoo simulator?

colon (:), Use a colon to inherit from a base class

constructors, When a base class has a constructor, your subclass needs one, too

extending, Create the class hierarchy

subclasses accessing with base keyword, A subclass can access its base class using the base keyword

upcasting, Upcasting works with both objects and interfaces

using subclasses instead, Any place where you can use a base class, you can use one of its subclasses instead

base keyword, A subclass can access its base class using the base keyword, Access modifiers change visibility

Basic Page template, Page layout starts with controls

Beehive Management System project, Build a beehive management system, How you'll build the beehive management system, How you'll build the beehive management system, Use inheritance to extend the bee management system, Use inheritance to extend the bee management system, Use inheritance to extend the bee management system, Let's get back to bee-sics, We can use inheritance to create classes for different types of bees, An interface tells a class that it must implement certain methods and properties, You can't instantiate an interface, but you can reference an interface, Interfaces can inherit from other interfaces, Bees need an OutOfHoney exception

building form, How you'll build the beehive management system

building Worker and Queen classes, How you'll build the beehive management system

class hierarchy with Worker and Queen classes, We can use inheritance to create classes for different types of bees

extending through inheritance, Use inheritance to extend the bee management system

interfaces, An interface tells a class that it must implement certain methods and properties, You can't instantiate an interface, but you can reference an interface, Interfaces can inherit from other interfaces

inheritance, Interfaces can inherit from other interfaces

references, You can't instantiate an interface, but you can reference an interface

making Worker class inherit from Bee class, Use inheritance to extend the bee management system

OutOfHoneyException, Bees need an OutOfHoney exception

updating form to instantiate bees, Use inheritance to extend the bee management system

bees, Build a beehive management system, Build a beehive management system, Use inheritance to extend the bee management system, Use more than one catch block to handle multiple types of exceptions, Exception avoidance: implement IDisposable to do your own cleanup, C# can build

“real” animations, too, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

accounting systems, Build a beehive management system, Use inheritance to extend the bee management system

animating, C# can build “real” animations, too

animating bees and stars, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

binary and decimal, converting between, The variable’s type determines what kind of data it can store

binary files, Let’s serialize and deserialize a deck of cards, Use a BinaryWriter to write binary data, You can read and write serialized files manually, too, Working with binary files can be tricky, Working with binary files can be tricky

comparing, You can read and write serialized files manually, too

hex dump, Working with binary files can be tricky

working with, Working with binary files can be tricky

writing, Use a BinaryWriter to write binary data

BinaryFormatter, Serialization lets you read or write a whole object graph all at once, Serialization lets you read or write a whole object graph all at once, Serialization lets you read or write a whole object graph all at once, If you want your class to be serializable, mark it with the [Serializable] attribute, Let’s serialize and deserialize a deck of cards, Let’s serialize and deserialize a deck of cards, Uh oh — the code’s still got problems..., Uh-oh — the code’s still got problems...

Deserialize() method, Serialization lets you read or write a whole object graph all at once, Let’s serialize and deserialize a deck of cards

Serializable attribute, If you want your class to be serializable, mark it with the [Serializable] attribute, Let’s serialize and deserialize a deck of cards

SerializationException, Uh oh — the code’s still got problems..., Uh-oh — the code’s still got problems...

Serialize( ) method, Serialization lets you read or write a whole object graph all at once

BinaryReader, Use BinaryReader to read the data back in

binary serialization versus data contract serialization, Build a slightly less simple text editor

BinaryWriter, Use a BinaryWriter to write binary data

binding., Make your bees fly around a page (see data binding)

Binding object, Data binding connects your XAML pages to your classes, Two-way binding can get or set the source property

binding path, Data binding connects your XAML pages to your classes, Use a data template to display objects, Use a data template to display objects

property type, Use a data template to display objects, Use a data template to display objects

Birthday Party project, Kathleen does birthday parties, too, We need a BirthdayParty class, Build the Part y Planner version 2.0, Build the Part y Planner version 2.0, Build the Part y Planner version 2.0, One more thing...can you add a $100 fee for parties over 12?, One more thing...can you add a $100 fee for parties over 12?, Now you're ready to finish the job for Kathleen!

adding controls to form, Build the Part y Planner version 2.0

adding fee for parties over 12 people, One more thing...can you add a $100 fee for parties over 12?

BirthdayParty.CalculateCost( ), One more thing...can you add a $100 fee for parties over 12?

BirthdayParty class, We need a BirthdayParty class

inheriting from Party class, Now you're ready to finish the job for Kathleen!

testing the program, Build the Part y Planner version 2.0

writing code to make controls work, Build the Part y Planner version 2.0

bitwise complement operator (~), ...more basics...

Blank App template, What you do in Visual Studio..., Start with a blank application, The IDE helps you code

Blend for Visual Studio 2012, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

blocks (of code), Add a method that does something, Let's take a closer look at your code, Use logical operators to check conditions, Build the Part y Planner version 2.0

leaving out curly brackets, Build the Part y Planner version 2.0

Boolean values, converters for, Converters can work with many different types, Converters can work with many different types

bool type, You have to assign values to variables before you use them, A few useful types, The variable's type determines what kind of data it can store, A variable is like a data to-go cup

true or false values, A few useful types

Border controls, XAML controls can contain text...and more

BottomAppBar property, Build a slightly less simple text editor

boxed objects and structs, The stack vs. the heap: more on memory, Nullable types help you make your programs more robust, "Captain" Amazing...not so much

boxed struct, "Captain" Amazing...not so much

break keyword in case statements, Use a switch statement to choose the right option, Use a switch statement to let your deck of cards read from a file or write itself out to one

breakpoints, Use the debugger to see your variables change, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Use the debugger to see your variables change, Use

the IDE’s debugger to ferret out exactly what went wrong in the Excuse Manager

inserting into code, Use the debugger to see your variables change, Use the debugger to see your variables change

knowing where to put, Use the IDE’s debugger to ferret out exactly what went wrong in the Excuse Manager, Use the IDE’s debugger to ferret out exactly what went wrong in the Excuse Manager

break statements, ...more basics...

Build menu (IDE), Where programs come from

Bullet Points, Where no object has gone before, Where no object has gone before, Generics can store any type, Bees need an OutOfHoney exception, Bees need an OutOfHoney exception, A callback is just a way to use delegates, A callback is just a way to use delegates

delegates, A callback is just a way to use delegates

event handlers, A callback is just a way to use delegates

exception handling, Bees need an OutOfHoney exception

Lists, Generics can store any type

reference variables, Where no object has gone before

try/catch blocks, Bees need an OutOfHoney exception

types, Where no object has gone before

Button controls, When you’re doing this..., if/else statements make decisions, if/else statements make decisions, Make each button do something, Rebuild your app for Windows Desktop, Rebuild your app for Windows Desktop, Wait, wait! What did that say?, When you create a new object from a class, it’s called an instance of that class, Build a form to interact with the guys, Build a form to interact with the guys, A few ideas for designing intuitive classes, Use data binding to build Sloppy Joe a better menu, Visual states make controls respond to changes, Build an app from the ground up, Build an app from the ground up, Use data binding to build Sloppy Joe a better menu

adding code to interact with objects, Build a form to interact with the guys

adding to form, Build a form to interact with the guys, A few ideas for designing intuitive classes

adding to page, When you're doing this...

adding to Windows Desktop app, Rebuild your app for Windows Desktop

Button class, When you create a new object from a class, it's called an instance of that class

changing properties for Windows Desktop app, Wait, wait! What did that say?

changing Text property in Properties window, Rebuild your app for Windows Desktop

Content property, if/else statements make decisions, Build an app from the ground up

making them do something, Make each button do something

MenuMaker project, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

naming using x:Name property, if/else statements make decisions, Build an app from the ground up

visual states, Visual states make controls respond to changes

buttons., Build a form to interact with the guys (see Button controls)

by keyword, Jimmy saved a bunch of dough

byte arrays, Things you can do with Directory:, C# can use byte arrays to move data around

moving text around in, C# can use byte arrays to move data around

byte order mark, Use Stream.Read() to read bytes from a stream

byte type, The variable's type determines what kind of data it can store, A variable is like a data to-go cup, When you cast a value that's too big, C# will adjust it automatically

casting int variable too large for, When you cast a value that's too big, C# will adjust it automatically

C

C#, Why you should learn C#, C# and the Visual Studio IDE make lots of things easy, What you do in Visual Studio..., What Visual Studio does for you..., Here's what you're going to build, Use the IDE to create your own method, Use a constructor to initialize private fields, #3. Namespaces and assemblies, Did you know that C# and the .NET Framework can...

and .NET Framework, capabilities of, Did you know that C# and the .NET Framework can...

application code, Here's what you're going to build

benefits of, Why you should learn C#

case in, Use a constructor to initialize private fields

code for AnimateEnemy( ) method (example), Use the IDE to create your own method

combining with XAML, What Visual Studio does for you...

files created by Visual Studio when creating new project, What you do in Visual Studio...

Microsoft reference for, #3. Namespaces and assemblies

using with Visual Studio IDE, capabilities of, C# and the Visual Studio IDE make lots of things easy

C# 5.0 in a Nutshell, Did you know that C# and the .NET Framework can...

Calculator program, The worst catch block EVER: catch-all plus comments, Temporary solutions are OK (temporarily)

temporary solution, Temporary solutions are OK (temporarily)

callbacks, Use a callback to control who's listening, MessageDialog uses the callback pattern

versus events, MessageDialog uses the callback pattern

camelCase, Use a constructor to initialize private fields

Candy Control System, You can use class and method names to make your code intuitive

Canvas control, Set up the grid for your page, Add controls to your grid, Use properties to change how the controls look, Controls make the game work, Controls make the game work, Controls make the game work, Make your bees fly around a page, Make your bees fly around a page, Use ItemsPanelTemplate to bind controls to a Canvas, Make your bees fly around a page, Make your bees fly around a page, Use ItemsPanelTemplate to bind controls to a Canvas

adding Ellipse control, Controls make the game work

adding to Windows Store app, Set up the grid for your page, Add controls to your grid

animating Canvas.Left property, Make your bees fly around a page, Make your bees fly around a page

binding controls to, using ItemsPanelTemplate, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

child controls, data binding and, Make your bees fly around a page, Make your bees fly around a page

dragging, changes to Left and Top properties, Controls make the game work

turning into gameplay area, Use properties to change how the controls look, Controls make the game work

Canvas controls, Controls make the game work, Controls make the game work

adding Ellipse control, Controls make the game work

dragging control around, changes to Left and Top properties, Controls make the game work

capitalization, Use a constructor to initialize private fields

Captain Amazing, ...and things are looking up back home!, Make an object serialize itself in its Dispose(), Make an object serialize itself in its Dispose(), "Captain" Amazing...not so much, Extending a fundamental type: string

case in C#, Use a constructor to initialize private fields

case sensitivity in C# and XAML, Start with a blank application, Set up the grid for your page

case statements, Use a switch statement to choose the right option

case statements, Use a switch statement to let your deck of cards read from a file or write itself out to one

(see also switch statements)

casting, Even when a number is the right size, you can't just assign it to any variable, Even when a number is the right size, you can't just assign it to any variable, When you cast a value that's too big, C# will adjust it automatically, When you cast a value that's too big, C# will adjust it automatically, When you cast a value that's too big, C# will adjust it automatically, C# does some casting automatically

arithmetic operators, automatic conversions with, When you cast a value that's too big, C# will adjust it automatically

automatic casting in C#, C# does some casting automatically

decimal value to int type, Even when a number is the right size, you can't just assign it to any variable

too-large value, automatic adjustment in C#, When you cast a value that's too big, C# will adjust it automatically

wrapping numbers, When you cast a value that's too big, C# will adjust it automatically

catch blocks, Handle exceptions with try and catch, What happens when a method you want to call is risky?, Use the debugger to follow the try/catch flow, If you have code that ALWAYS should run, use a finally block, Use more than one catch block to handle multiple types of exceptions, Bees need an OutOfHoney exception, The worst catch block EVER: catch-all plus comments, Handle exceptions with try and catch, What happens when a method you want to call is risky?, Use the debugger to follow the try/catch flow

following in debugger, Use the debugger to follow the try/catch flow, Use the debugger to follow the try/catch flow

letting your program keep running, The worst catch block EVER: catch-all plus comments

multiple, to handle multiple types of exceptions, Use more than one catch block to handle multiple types of exceptions

with no specified exceptions, If you have code that ALWAYS should run, use a finally block

chaining events, Use a standard name when you add a method to raise an event, One event, multiple handlers

Character Map (Charmap.exe), Let's serialize and deserialize a deck of cards, .NET uses Unicode to store characters and text

Charms, Use delegates to use the Windows settings charm

char type, The variable's type determines what kind of data it can store, A variable is like a data to-go cup, .NET uses Unicode to store characters and text

CheckBox controls, Make each button do something, Use logical operators to check conditions, Rebuild your app for Windows Desktop, Rebuild your app for Windows Desktop, Build the Part y Planner version 2.0

adding to Windows Desktop app, Rebuild your app for Windows Desktop

Birthday Party project, Build the Part y Planner version 2.0

changing Text and Checked properties in Properties window, Rebuild your app for Windows Desktop

CheckFileExists property, OpenFileDialog, Use built-in objects to pop up standard dialog boxes

CheckPathExists property, OpenFileDialog, Use built-in objects to pop up standard dialog boxes

child, How would you design a zoo simulator?

Children collection, XAML controls, XAML controls can contain text...and more

class diagrams, Mike gets an idea, Give your classes a natural structure, Class diagrams help you organize your classes so they make sense, You're going to build a program for Kathleen, Any place where you can use a base class, you can use one of its subclasses instead, How you'll build the beehive management system

DinnerParty class (example), You're going to build a program for Kathleen

moving up, not down, Any place where you can use a base class, you can use one of its subclasses instead

organizing classes to make sense, Class diagrams help you organize your classes so they make sense

private fields and types, How you'll build the beehive management system

using to plan classes, Give your classes a natural structure

classes, Anatomy of a program, Let's take a closer look at your code, Two classes can be in the same namespace, Two classes can be in the same namespace, Make each button do something, Use logical operators to check conditions, Use logical operators to check conditions, Use logical operators to check conditions, Rebuild your app for Windows Desktop, Rebuild your app for Windows Desktop, You can change your program's entry point, How Mike thinks about his problems, How Mike's car navigation system thinks about his problems, Mike gets an idea, You use a class to build an object, A little advice for the code exercises, Let's create some instances!, You can use class and method names to make your code intuitive, Give your classes a natural structure, Class diagrams help you organize your classes so they make sense, Class diagrams help you organize your classes so they make sense, Build a class to work with some guys, Create a project for your guys, A few ideas for designing intuitive classes, A few ideas for designing intuitive classes, Use encapsulation to control access to your class's methods and fields, Think of an object as a black box, We need a BirthdayParty class, One more thing...can you add a $100 fee for parties over 12?, Think about how to group the animals, Now you're ready to finish the job for Kathleen!, An interface tells a class that it must implement certain methods and properties, You can find out if a class implements a certain interface with "is", There's more than just public and private, There's more than just public and private, There's more than just public and private, There's more than just public and private, There's more than just public and private, There's more than just public and private, Some classes should never be instantiated, An abstract class is like a cross between a class and an interface, An abstract class is like a cross between a class and an interface, Like we said, some classes should never be instantiated, Lists make it easy to store collections of...anything, If you want your class to be serializable, mark it with the [Serializable] attribute, Nullable types help you make your programs more robust, Extension methods add new behavior to EXISTING classes, Connecting the dots

abstract, An abstract class is like a cross between a class and an interface (see abstract classes)

adding new class to desktop app, You can change your program's entry point

code between { } (curly braces), Make each button do something

collection, Lists make it easy to store collections of...anything

concrete, An abstract class is like a cross between a class and an interface

copying, Mike gets an idea

creating (example), Create a project for your guys

creating instances of, Let's create some instances!

creating using code snippets, Use logical operators to check conditions

designing, How Mike's car navigation system thinks about his problems, Class diagrams help you organize your classes so they make sense, Build a class to work with some guys, A few ideas for designing intuitive classes, We need a BirthdayParty class, Now you're ready to finish the job for Kathleen!

separation of concerns, Now you're ready to finish the job for Kathleen!

encapsulation, Use encapsulation to control access to your class's methods and fields, Think of an object as a black box

finding out if class implements specific interface, You can find out if a class implements a certain interface with “is”

inheritance., One more thing...can you add a $100 fee for parties over 12? (see inheritance)

internal, There’s more than just public and private

looking for common, Think about how to group the animals

members, There’s more than just public and private

methods, Let’s take a closer look at your code

namespaces, Two classes can be in the same namespace

naming, You can use class and method names to make your code intuitive

natural structure, Give your classes a natural structure

never instantiated, Some classes should never be instantiated

organizing, Class diagrams help you organize your classes so they make sense

partial, Use logical operators to check conditions

private, There’s more than just public and private

protected, There’s more than just public and private

public, There’s more than just public and private

required by interface to implement certain methods and properties, An interface tells a class that it must implement certain methods and properties

sealed, There’s more than just public and private, Extension methods add new behavior to EXISTING classes

serializable, If you want your class to be serializable, mark it with the [Serializable] attribute

similarities between, A few ideas for designing intuitive classes

statements in, Use logical operators to check conditions

static, A little advice for the code exercises

subscribing, Connecting the dots

using lines for adding methods from other namespaces, Rebuild your app for Windows Desktop

using to build objects, You use a class to build an object

versus structs, Nullable types help you make your programs more robust

why some should never be instantiated, Like we said, some classes should never be instantiated

class hierarchy, Build up your class model by starting general and getting more specific, Create the class hierarchy, We can use inheritance to create classes for different types of bees

Hive Simulator, We can use inheritance to create classes for different types of bees

class libraries, creating, #3. Namespaces and assemblies

clauses in LINQ queries, LINQ makes queries easy

Clone class, implementing IDisposable, Dispose() works with using; finalizers work with garbage collection, Dispose() works with using; finalizers work with garbage collection

Close buttons, Windows Store apps and, Use static resources to declare your objects in XAML

clowns, Let's create some instances!, Upcasting and downcasting work with interfaces, too, Upcasting and downcasting work with interfaces, too, Upcasting and downcasting work with interfaces, too, Delegates in action, User controls let you create your own controls

- Fingers the Clown, Upcasting and downcasting work with interfaces, too, Delegates in action, User controls let you create your own controls
- scary clown, Upcasting and downcasting work with interfaces, too

CLR (Common Language Runtime), The .NET Framework gives you the right tools for the job, Where no object has gone before

code, What Visual Studio does for you..., Use logical operators to check conditions, Mike gets an idea, A better solution...brought to you by objects!, One more thing...can you add a $100 fee for parties over 12?, When your classes use inheritance, you only need to write your code once, How would you design a zoo simulator?

- advice for code exercises, A better solution...brought to you by objects!
- automatically generated by IDE, What Visual Studio does for you..., Use logical operators to check conditions
- avoiding duplication, How would you design a zoo simulator?
- copying, Mike gets an idea
- repeating, One more thing...can you add a $100 fee for parties over 12?
- similar, When your classes use inheritance, you only need to write your code once

code blocks, Add a method that does something, Let's take a closer look at your code, Use logical operators to check conditions, Build the Part y Planner version 2.0

- leaving out curly brackets, Build the Part y Planner version 2.0

code snippets, Loops perform an action over and over, Use logical operators to check conditions, Use logical operators to check conditions, Class diagrams help you organize your classes so they make sense, A few ideas for designing intuitive classes

- rearranging to make working C# program, Use logical operators to check conditions
- using to create classes, Use logical operators to check conditions, Class diagrams help you organize your classes so they make sense, A few ideas for designing intuitive classes
- using to write for loops, Loops perform an action over and over

collection initializers, Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!, Collection initializers are similar to object initializers

collections, Arrays are hard to work with, Lists make it easy to store collections of...anything, Generics can store any type, Use a dictionary to store keys and values, And yet MORE collection types..., Two-way binding can get or set the source property, XAML controls can contain text...and more, Brian's code did something unexpected, .NET collections are already set up for LINQ, LINQ is versatile, Use join to combine two collections into one sequence, Jimmy saved a bunch of dough, #7. Using yield return to create enumerable objects, Brian's code did something unexpected

(see also listings of individual collection types)

binding to, with ObservableCollection, Two-way binding can get or set the source property

controls contained in another control, XAML controls can contain text...and more

dictionaries, Use a dictionary to store keys and values

exception from trying to access nonexistent element, Brian's code did something unexpected, Brian's code did something unexpected

generic, Generics can store any type

implementing IEnumerator<T> GetEnumerator( ), .NET collections are already set up for LINQ

indexers, #7. Using yield return to create enumerable objects

lists, Lists make it easy to store collections of...anything

performing calculations on, LINQ is versatile

queues and stacks, And yet MORE collection types...

using join to combine two collections into one query, Use join to combine two collections into one sequence, Jimmy saved a bunch of dough

Collection<T> interface, #7. Using yield return to create enumerable objects

colon operator, Now you can create an instance of NectarStinger that does both jobs

Color.FromArgb( ) method, Wait, wait! What did that say?

color gradient, adding to XAML control, Controls make the game work

color gradients, Use properties to change how the controls look

adding to Canvas control, Use properties to change how the controls look

colors, What Visual Studio does for you..., Wait, wait! What did that say?, Wait, wait! What did that say?

cycling through form's background colors in animation, Wait, wait! What did that say?

predefined, or making your own, Wait, wait! What did that say?

selecting color theme in Visual Studio, What Visual Studio does for you...

ComboBoxItem object, Use a data template to display objects, Use a data template to display objects

command-line arguments, Use Stream.Read() to read bytes from a stream

CommandsRequested event, Use delegates to use the Windows settings charm

comments, Use the debugger to see your variables change, Make each button do something, Rebuild your app for Windows Desktop, Wait, wait! What did that say?, #2. The Basics, ...more basics..., Use the debugger to see your variables change

adding to code, starting with //, Make each button do something

/* and */ enclosing multiline comments, ...more basics...

beginning with ///, Rebuild your app for Windows Desktop, Wait, wait! What did that say?

starting with /* or //, Use the debugger to see your variables change, Use the debugger to see your variables change

XML, #2. The Basics

Common Intermediate Language (IL), ...so what did I just do?

Common Language Runtime (CLR), The .NET Framework gives you the right tools for the job, Where no object has gone before

CommonStates group, Visual states make controls respond to changes

CompareTo( ) method, IComparable<Duck> helps your list sort its ducks

compiler errors, classes implementing interfaces, An interface tells a class that it must implement certain methods and properties

compiling programs, using Build menu in IDE, Where programs come from

compound operators, A few useful types, When you call a method, the arguments must be compatible with the types of the parameters, Combining = with an operator

concatenation operator (+), A few useful types, C# does some casting automatically

automatic type conversions with, C# does some casting automatically

concrete classes, An abstract class is like a cross between a class and an interface

conditional expressions., Set up conditions and see if they're true, Rename a variable

(see also conditional tests)

consolidating, Rename a variable

conditional operator (?:), Add the stopwatch ViewModel, ...more basics..., #6. Equality, IEquatable, and Equals()

conditional operators, Set up conditions and see if they're true

conditional tests, Set up conditions and see if they're true, Use logical operators to check conditions, Use logical operators to check conditions

resulting in infinite loops, Use logical operators to check conditions

console applications, Any place where you can use a base class, you can use one of its subclasses instead

Console.Error.WriteLine( ), Use Stream.Read() to read bytes from a stream

Console.WriteLine() method, Build an application to test the Farmer class

constants, You're going to build a program for Kathleen

constructors, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields, When a base class has a constructor, your subclass needs one, too, Add an overloaded Deck() constructor that reads a deck of cards in from a file, Use static resources to declare your objects in XAML, Modify MenuMaker to notify you when the GeneratedDate property changes, Use the debugger to follow the try/catch flow, Create a user control to animate a picture, Use static resources to declare your objects in XAML, Modify MenuMaker to notify you when the GeneratedDate property changes, Use the debugger to follow the try/catch flow, Create a user control to animate a picture

  base class and subclass, When a base class has a constructor, your subclass needs one, too

  building new with switch statement, Add an overloaded Deck() constructor that reads a deck of cards in from a file

  closer examination of, Use a constructor to initialize private fields

  exceptions in, Use the debugger to follow the try/catch flow, Use the debugger to follow the try/catch flow

  parameterless, Use static resources to declare your objects in XAML, Modify MenuMaker to notify you when the GeneratedDate property changes, Create a user control to animate a picture, Use static resources to declare your objects in XAML, Modify MenuMaker to notify you when the GeneratedDate property changes, Create a user control to animate a picture

  without parameters, Use a constructor to initialize private fields

container tags, What Visual Studio does for you...

ContentControl, Set up the grid for your page, Add controls to your grid, Add controls to your grid, Use properties to change how the controls look, Use properties to change how the controls look, Controls make the game work, What you'll do next, Use the IDE to create your own method

  adding to Windows Store app, Set up the grid for your page, Add controls to your grid

  creating new ContentControl object and adding method, What you'll do next, Use the IDE to create your own method

  Edit Template, Create Empty..., Use properties to change how the controls look, Controls make the game work

  grouping, using StackPanel, Add controls to your grid, Use properties to change how the controls look

content controls, XAML controls can contain text...and more

Content property, Use properties to change how the controls look, XAML controls can contain text...and more, XAML controls can contain text...and more

Button controls, Use properties to change how the controls look

XAML controls, XAML controls can contain text...and more, XAML controls can contain text...and more

Content property, Button controls, Add controls to your grid

continue statements, ...more basics...

ControlCollection object, Windows Forms use an object graph set up by the IDE

controls, Here's what you're going to build, Here's what you're going to build, Set up the grid for your page, Add controls to your grid, Add controls to your grid, Use properties to change how the controls look, Use properties to change how the controls look, Controls make the game work, Controls make the game work, Controls make the game work, Make the Start button work, Make the Start button work, Add code to make your controls interact with the player, Add code to make your controls interact with the player, When you're doing this..., The .NET Framework gives you the right tools for the job, Controls are objects, just like any other object, Use a constructor to initialize private fields, Windows Forms use an object graph set up by the IDE, Redesign the Go Fish! form as a Windows Store app page, Page layout starts with controls, Data binding connects your XAML pages to your classes, Two-way binding can get or set the source property, XAML controls can contain text...and more, XAML controls can contain text...and more, Finish the stopwatch app, Styles set properties on multiple controls, Use a resource dictionary to share resources between pages, Visual states make controls respond to changes, Use DoubleAnimation to animate double values, UI controls can be instantiated with C# code, too, Use ItemsPanelTemplate to bind controls to a Canvas, Page layout starts with controls, UI controls can be instantiated with C# code, too, Use ItemsPanelTemplate to bind controls to a Canvas

adding code to make controls interact with player, Make the Start button work, Add code to make your controls interact with the player

adding code to make them interact with player, Make the Start button work, Add code to make your controls interact with the player

adding to page, When you're doing this...

altering appearance of a type, using styles, Styles set properties on multiple controls

altering appearance of every control of a specific type, Use a resource dictionary to share resources between pages

binding to canvas using ItemsPanelTemplate, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

C# code for, Here's what you're going to build

creating UI controls with C# code, UI controls can be instantiated with C# code, too, UI controls can be instantiated with C# code, too

data binding, connecting XAML pages to classes, Data binding connects your XAML pages to your classes

displaying collections, data binding to collection, Two-way binding can get or set the source property

double properties, animation for, Use DoubleAnimation to animate double values

dragged from Toolbox onto page, XAML generated for, Set up the grid for your page, Add controls to your grid

dragging around Canvas, Controls make the game work, Controls make the game work

initialization on forms with InitializeComponent( ), Use a constructor to initialize private fields

in MVVM applications, Finish the stopwatch app

in .NET for Windows Store apps, The .NET Framework gives you the right tools for the job

making game work in Windows Store app, Use properties to change how the controls look, Controls make the game work

nesting inside other controls, XAML controls can contain text...and more

page layout starting with, Page layout starts with controls, Page layout starts with controls

program animating Label controls, Controls are objects, just like any other object

using properties to change look of, Add controls to your grid, Use properties to change how the controls look

visual states causing response to changes, Visual states make controls respond to changes

Windows Store app, on a page, Redesign the Go Fish! form as a Windows Store app page

WinForms apps, Windows Forms use an object graph set up by the IDE

XAML, containing text and more, XAML controls can contain text...and more

Controls property, Windows Forms use an object graph set up by the IDE, Use the IDE to explore the object graph

controls containing other controls, Windows Forms use an object graph set up by the IDE

Form class and, Use the IDE to explore the object graph

ControlTemplate, Controls make the game work

ControlTemplate, Add a splash screen and a tile

(see also templates)

Convert( ) and ConvertBack( ) methods, value converter, Converters automatically convert values for binding, Converters automatically convert values for binding

converters, Converters automatically convert values for binding, Converters automatically convert values for binding, Converters can work with many different types, Build an analog stopwatch using the same ViewModel, ...more basics..., Converters automatically convert values for binding,

Converters automatically convert values for binding, Converters can work with many different types, Build an analog stopwatch using the same ViewModel

automatically converting values for binding, Converters automatically convert values for binding, Converters automatically convert values for binding

converting minutes and seconds to angles, Build an analog stopwatch using the same ViewModel, Build an analog stopwatch using the same ViewModel

Convert.ToString( ) and Convert.ToInt32( ), ...more basics...

working with many different types, Converters can work with many different types, Converters can work with many different types

covariance, You can upcast an entire list using IEnumerable

CreateDirectory( ) method, Use the built-in File and Directory classes to work with files and directories

CreateFileAsync( ) method, Use async methods to find and open files

Create( ) method, Use the built-in File and Directory classes to work with files and directories

CryptoStream, Data can go through more than one stream

.csproj (project) files, Where programs come from

curly brackets., Create a project for your guys (see { }, under Symbols)

CurrentQueryResults property, Use the new keyword to create anonymous types

D

data, Strings don't always work for storing categories of data, LINQ can pull data from multiple sources

pulling data from multiple sources, LINQ can pull data from multiple sources

storing categories of, Strings don't always work for storing categories of data

data binding, Data binding connects your XAML pages to your classes, Two-way binding can get or set the source property, Two-way binding can get or set the source property, Use data binding to build Sloppy Joe a better menu, Use a data template to display objects, Use a data template to display objects, INotifyPropertyChanged lets bound objects send updates, Modify MenuMaker to notify you when the GeneratedDate property changes, Do you design for binding or for working with data?, MVVM lets you design for binding and data, User controls let you create your own controls, Converters automatically convert values for binding, Make your bees fly around a page, Use ItemsPanelTemplate to bind controls to a Canvas, Use data binding to build Sloppy Joe a better menu, INotifyPropertyChanged lets bound objects send updates, Modify MenuMaker to notify you when the GeneratedDate property changes, User controls let you create your own controls, Converters automatically convert values for binding, Make your bees fly around a page, Use ItemsPanelTemplate to bind controls to a Canvas

Canvas child controls and, Make your bees fly around a page, Make your bees fly around a page

connecting XAML pages to classes, Data binding connects your XAML pages to your classes

converters automatically convering values for, Converters automatically convert values for binding, Converters automatically convert values for binding

designing for, Do you design for binding or for working with data?

designing for binding and data handling with MVVM pattern, MVVM lets you design for binding and data

INotifyPropertyChanged, bound objects sending updates, INotifyPropertyChanged lets bound objects send updates, INotifyPropertyChanged lets bound objects send updates

public properties for Go Fish game conversion, Modify MenuMaker to notify you when the GeneratedDate property changes, Modify MenuMaker to notify you when the GeneratedDate property changes

RosterControl XAML control (example), User controls let you create your own controls, User controls let you create your own controls

to collections, with ObservableCollection, Two-way binding can get or set the source property

two way binding, getting or setting source property, Two-way binding can get or set the source property

using data template to display objects, Use a data template to display objects, Use a data template to display objects

using ItemsPanelTemplate to bind controls to a canvas, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

using to build Sloppy Joe's menu, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

data context, Data binding connects your XAML pages to your classes, Use data binding to build Sloppy Joe a better menu, Use static resources to declare your objects in XAML, User controls let you create your own controls, User controls let you create your own controls, Build the view for a simple stopwatch, Use ItemsPanelTemplate to bind controls to a Canvas, Use data binding to build Sloppy Joe a better menu, Use static resources to declare your objects in XAML, User controls let you create your own controls, User controls let you create your own controls, Build the view for a simple stopwatch, Use ItemsPanelTemplate to bind controls to a Canvas

RosterControl (example), User controls let you create your own controls, User controls let you create your own controls

setting for menu maker (example), Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

setting for StackPanel and its children, Use static resources to declare your objects in XAML, Use static resources to declare your objects in XAML

DataContract attribute, A data contract is an abstract definition of your object's data, The whole

object graph is serialized to XML

data contract serialization, Build a slightly less simple text editor, A data contract is an abstract definition of your object's data, A data contract is an abstract definition of your object's data, The whole object graph is serialized to XML, Stream some Guy objects to XML files, Take your Guy Serializer for a test drive, Stream some Guy objects to a file, Take your Guy Serializer for a test drive

  data contract, defined, A data contract is an abstract definition of your object's data

  disambiguation in, Take your Guy Serializer for a test drive, Take your Guy Serializer for a test drive

  sending some objects to app's local folder, Stream some Guy objects to XML files, Stream some Guy objects to a file

  using XML files, A data contract is an abstract definition of your object's data

  whole object graph serialized to XML, The whole object graph is serialized to XML

DataMember attribute, A data contract is an abstract definition of your object's data, The whole object graph is serialized to XML

DataModel folder, adding data classes to, The IDE's Split App template helps you build apps for navigating data

data template, using to display objects, Use a data template to display objects, Use a data template to display objects

Deadly Diamond of Death, An abstract method doesn't have a body

debugger, When your program throws an exception, .NET generates an Exception object

debugger, Use the debugger to see your variables change, Use the debugger to see your variables change, The debugger helps you track down and prevent exceptions in your code, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?, Use the debugger to follow the try/catch flow, Use the debugger to follow the try/catch flow, If you have code that ALWAYS should run, use a finally block, If you have code that ALWAYS should run, use a finally block, Use more than one catch block to handle multiple types of exceptions, Bees need an OutOfHoney exception, Delegates in action, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?, Use the debugger to follow the try/catch flow, Use the debugger to follow the try/catch flow, If you have code that should ALWAYS run, use a finally block

(see also exception handling)

Bullet Points, Bees need an OutOfHoney exception

catch blocks, Use the debugger to follow the try/catch flow, If you have code that ALWAYS should run, use a finally block, Use more than one catch block to handle multiple types of exceptions, Use the debugger to follow the try/catch flow

following flow, Use the debugger to follow the try/catch flow, Use the debugger to follow the try/catch flow

multiple, Use more than one catch block to handle multiple types of exceptions

with no specified exceptions, If you have code that ALWAYS should run, use a finally block

exploring delegates, Delegates in action

finally block, If you have code that ALWAYS should run, use a finally block, If you have code that should ALWAYS run, use a finally block

following try/catch flow, Use the debugger to follow the try/catch flow, Use the debugger to follow the try/catch flow

knowing where to put breakpoints, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager

Step Into command, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager

uses for, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?

using to, Use the debugger to see your variables change

using to, Use the debugger to see your variables change (see changes in variables)

Watch window, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, What happens when a method you want to call is risky?, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, What happens when a method you want to call is risky?

running methods in, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager

debugging, Rebuild your app for Windows Desktop, Windows Forms use an object graph set up by the IDE, The debugger helps you track down and prevent exceptions in your code, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager

Excuse Management program, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager

System.Diagnostics.Debug.WriteLine( ), Windows Forms use an object graph set up by the IDE

Windows Desktop app in IDE, Rebuild your app for Windows Desktop

Debug menu, Where programs come from, Use the debugger to see your variables change, Use the debugger to see your variables change, Use the debugger to see your variables change, Wait, wait! What did that say?

Continue, Use the debugger to see your variables change

Start Debugging, Where programs come from, Use the debugger to see your variables change

Step Over, Use the debugger to see your variables change

Stop Debugging, Wait, wait! What did that say?

decimal and binary numbers, converting between, The variable's type determines what kind of data it can store

decimal type, The variable's type determines what kind of data it can store, A variable is like a data to-go cup, Even when a number is the right size, you can't just assign it to any variable, You're going to build a program for Kathleen

attempting to assign decimal value to int variable, Even when a number is the right size, you can't just assign it to any variable

using for monetry values, You're going to build a program for Kathleen

decrement operator (- -), A few useful types

default property of controls, XAML controls can contain text...and more

deferred evaluation, LINQ is versatile

delegate, defined, Connecting event senders with event listeners

delegates, One event, multiple handlers, A delegate STANDS IN for an actual method, A delegate STANDS IN for an actual method, Delegates in action, Delegates in action, Use a callback to control who's listening, A callback is just a way to use delegates, A callback is just a way to use delegates, MessageDialog uses the callback pattern, Use delegates to use the Windows settings charm

callbacks and, A callback is just a way to use delegates

defined, A delegate STANDS IN for an actual method

delegate type, A delegate STANDS IN for an actual method

events, callbacks, and, MessageDialog uses the callback pattern

exploring in debugger, Delegates in action

hooking up to one event, Use a callback to control who's listening

in action, Delegates in action

multiple events, One event, multiple handlers

using the Windows settings charm, Use delegates to use the Windows settings charm

Delete( ) method, Use the built-in File and Directory classes to work with files and directories

DependencyProperty class, Two-way binding can get or set the source property

deployment package, Here's what you're going to build

deselecting controls for editing, Add controls to your grid, Use properties to change how the controls look

design, You can use class and method names to make your code intuitive, A few ideas for designing intuitive classes, Now you're ready to finish the job for Kathleen!

intuitive classes, A few ideas for designing intuitive classes

making code intuitive with class and method names, You can use class and method names to make your code intuitive

separation of concerns, Now you're ready to finish the job for Kathleen!

design patterns, MessageDialog uses the callback pattern, MessageDialog uses the callback pattern, MessageDialog uses the callback pattern, Do you design for binding or for working with data?, User controls let you create your own controls, User controls let you create your own controls, Finish the stopwatch app, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

(see also MVVM pattern)

Callback pattern, MessageDialog uses the callback pattern

Factory Method pattern, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

Model-View-Controller (MVC) pattern, User controls let you create your own controls

Model-View-ViewModel (MVVM) pattern, Do you design for binding or for working with data?, User controls let you create your own controls, Finish the stopwatch app

Observer pattern, MessageDialog uses the callback pattern

desktop applications, The .NET Framework gives you the right tools for the job

destructor, Your last chance to DO something... your object's finalizer

developer license, Start remote debugging

device-independent units, Use the grid system to lay out app pages

dialog boxes, Use built-in objects to pop up standard dialog boxes, Dialog boxes are just another WinForms control, Dialog boxes are objects, too, Things you can do with Directory:, Use file dialogs to open and save files (all with just a few lines of code)

  as objects, Dialog boxes are objects, too

  customized, Things you can do with Directory:

  file dialogs, Use file dialogs to open and save files (all with just a few lines of code)

  popping up, Use built-in objects to pop up standard dialog boxes

DialogResult, Use built-in objects to pop up standard dialog boxes, You can help Brian out by building a program to manage his excuses

  excuse management program, You can help Brian out by building a program to manage his excuses

dictionaries, Use a dictionary to store keys and values, Use a dictionary to store keys and values, Use a dictionary to store keys and values, Use a dictionary to store keys and values, The dictionary functionality rundown, The dictionary functionality rundown, Build a program that uses a dictionary, #7. Using yield return to create enumerable objects

  Add( ) method, Use a dictionary to store keys and values

  building program that uses, Build a program that uses a dictionary

  ContainsKey( ) method, Use a dictionary to store keys and values

  functionality rundown, The dictionary functionality rundown

  keys, Use a dictionary to store keys and values

  keys and values, The dictionary functionality rundown

  using [ ] to return object from, #7. Using yield return to create enumerable objects

Dinner Party Planning project, Kathleen is an event planner, What does the estimator do?, You're going to build a program for Kathleen, You're going to build a program for Kathleen, Kathleen's test drive, Each option should be calculated individually, Each option should be calculated individually, Each option should be calculated individually, It's easy to accidentally misuse your objects, Encapsulation means keeping some of the data in a class private, Use a constructor to initialize private fields, When your classes use inheritance, you only need to write your code once, Now you're ready to finish the job for Kathleen!

CalculateCostOfDecorations( ) method, It's easy to accidentally misuse your objects

cost estimate, What does the estimator do?

DinnerParty class, You're going to build a program for Kathleen, You're going to build a program for Kathleen

encapsulating fields in DinnerParty class, Encapsulation means keeping some of the data in a class private

fixing calculator, Use a constructor to initialize private fields

inheriting from Party class, Now you're ready to finish the job for Kathleen!

numericUpDown control, Each option should be calculated individually

options, caluclating individually, Each option should be calculated individually

recalculating new individual costs, Each option should be calculated individually

similarities between DinnerParty and BirthdayParty classes, When your classes use inheritance, you only need to write your code once

test drive, Kathleen's test drive

directories, Use the built-in File and Directory classes to work with files and directories, Use the built-in File and Directory classes to work with files and directories, Use the built-in File and Directory classes to work with files and directories

creating new, Use the built-in File and Directory classes to work with files and directories

deleting, Use the built-in File and Directory classes to work with files and directories

getting list of files, Use the built-in File and Directory classes to work with files and directories

Directory.GetFiles( ) method, You can help Brian out by building a program to manage his excuses

Disabled state (controls), Visual states make controls respond to changes

disambiguation, Take your Guy Serializer for a test drive, Take your Guy Serializer for a test drive

Dispose( ) method, IDisposable makes sure your objects are disposed of properly, Avoid filesystem errors with using statements, Exception avoidance: implement IDisposable to do your own cleanup, Exception avoidance: implement IDisposable to do your own cleanup, Dispose() works with using; finalizers work with garbage collection, Finalizers can't depend on stability, Make an object serialize itself in its Dispose(), Make an object serialize itself in its Dispose()

calling outside of using statement, Exception avoidance: implement IDisposable to do your own cleanup

finalizers, Finalizers can't depend on stability, Make an object serialize itself in its Dispose()

making object serialize in, Make an object serialize itself in its Dispose()

using statement, Dispose() works with using; finalizers work with garbage collection

DivideByZeroException, But the program isn't working!, All exception objects inherit from

Exception

division operator (/), A few useful types

DLL file extension, #3. Namespaces and assemblies

Document Outline window, Use properties to change how the controls look

modifying StackPanel and TextBlock controls, Use properties to change how the controls look

Document Outline window, modifying controls, Controls make the game work

documents library, accessing with Windows Store apps, KnownFolders helps you access high-profile folders, Take your Guy Serializer for a test drive

dot (.) operator, A few useful types

DoubleAnimation, Use DoubleAnimation to animate double values, Make your bees fly around a page, Make your bees fly around a page

animating Canvas.Left property, Make your bees fly around a page, Make your bees fly around a page

double type, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, A variable is like a data to-go cup

defined, The variable's type determines what kind of data it can store

downcasting, Downcasting lets you turn your appliance back into a coffee maker, Upcasting and downcasting work with interfaces, too, Upcasting and downcasting work with interfaces, too, Polymorphism means that one object can take many different forms

failure of, Upcasting and downcasting work with interfaces, too

interfaces, Upcasting and downcasting work with interfaces, too

using as keyword, Polymorphism means that one object can take many different forms

E

editors, Let's take a closer look at your code, Build a slightly less simple text editor

building less simple text editor, Build a slightly less simple text editor

evolution of code editors, Let's take a closer look at your code

Edit Style right-mouse menu, Use properties to change how the controls look, if/else statements make decisions, Build an app from the ground up

changing text style for TextBlock, Use properties to change how the controls look

Edit Text right-mouse menu, Add controls to your grid, Add controls to your grid, Use properties to change how the controls look, Use properties to change how the controls look

changing text for TextBlock control, Add controls to your grid, Use properties to change how the controls look

editing text for button in Windows Store app, Add controls to your grid, Use properties to change how the controls look

Ellipse controls, Controls make the game work, Controls make the game work, Dragging humans onto enemies ends the game, Make your enemies look like aliens

adding to Canvas, Controls make the game work, Controls make the game work

editing to make enemies look like aliens (example), Dragging humans onto enemies ends the game, Make your enemies look like aliens

ellipses, Use properties to change how the controls look, Controls make the game work

encapsulation, _, C# Lab: A Day at the Races, Encapsulation means keeping some of the data in a class private, Use encapsulation to control access to your class's methods and fields, Private fields and methods can only be accessed from inside the class, Mike's navigator program could use better encapsulation, Think of an object as a black box, A few ideas for encapsulating classes, Encapsulation keeps your data pristine, Use automatic properties to finish the class, Build the Part y Planner version 2.0, Access modifiers change visibility, An abstract method doesn't have a body, Use Stream.Read() to read bytes from a stream

as principle of OOP, An abstract method doesn't have a body

automatic properties, Use automatic properties to finish the class

benefits for classes, Private fields and methods can only be accessed from inside the class

better, using protected modifier, Access modifiers change visibility

BirthdayParty class (example), Build the Part y Planner version 2.0

defined, Encapsulation means keeping some of the data in a class private

example, Encapsulation keeps your data pristine

ideas for, A few ideas for encapsulating classes

Navigator program (example), Mike's navigator program could use better encapsulation

using to control access to class methods. fields, or properties, Use encapsulation to control access to your class's methods and fields

well encapsulated versus poorly encapsulated classes, Think of an object as a black box

encodings, A FileStream reads and writes bytes to a file, Things you can do with Directory:, Let's serialize and deserialize a deck of cards, .NET uses Unicode to store characters and text

Unicode, Let's serialize and deserialize a deck of cards, .NET uses Unicode to store characters and text

end tags, What Visual Studio does for you...

entry point for a program, Rebuild your app for Windows Desktop, You can change your program's

entry point, Any place where you can use a base class, you can use one of its subclasses instead

changing, You can change your program's entry point

enumerable objects, creating with yield return, #7. Using yield return to create enumerable objects

enumeration, Strings don't always work for storing categories of data

enums, Enums let you work with a set of valid values, Enums let you represent numbers with names, Enums let you represent numbers with names, Enums let you represent numbers with names, Enums let you represent numbers with names, Generics can store any type

big numbers, Enums let you represent numbers with names

building class that holds playing card, Enums let you represent numbers with names, Enums let you represent numbers with names

representing numbers with names, Enums let you represent numbers with names

versus Lists, Generics can store any type

equality, #6. Equality, IEquatable, and Equals()

== operator, IEquatable, and Equals( ), #6. Equality, IEquatable, and Equals()

equality operator (==), if/else statements make decisions, Set up conditions and see if they're true

Equals( ) method, #6. Equality, IEquatable, and Equals()

error handling, If you have code that ALWAYS should run, use a finally block

Error List window, What Visual Studio does for you..., Use the IDE to create your own method, Finish the method and run your program, The IDE helps you code, Let's take a closer look at your code

examining errors in, Use the IDE to create your own method, Finish the method and run your program

troubleshooting compiler errors, The IDE helps you code, Let's take a closer look at your code

errors, When you call a method, the arguments must be compatible with the types of the parameters, An interface tells a class that it must implement certain methods and properties, Avoid filesystem errors with using statements, But the program isn't working!

avoiding file system errors with using statements, Avoid filesystem errors with using statements

compiler errors and interfaces, An interface tells a class that it must implement certain methods and properties

DivideByZero, But the program isn't working!

invalid arguments, When you call a method, the arguments must be compatible with the types of the parameters

escape sequences, Make each button do something

EventHandler, Connecting the dots, Use a standard name when you add a method to raise an event, A

callback is just a way to use delegates

as type of delegate, A callback is just a way to use delegates

event handlers, Make the Start button work, Add code to make your controls interact with the player, Rebuild your app for Windows Desktop, Private fields and methods can only be accessed from inside the class, Private fields and methods can only be accessed from inside the class, Build the Part y Planner version 2.0, You can help Brian out by building a program to manage his excuses, Build a slightly less simple text editor, When an EVENT occurs...objects listen, One object raises its event, others listen for it..., Use a standard name when you add a method to raise an event, Use a standard name when you add a method to raise an event, Use a standard name when you add a method to raise an event, The IDE generates event handlers for you automatically, One event, multiple handlers, A callback is just a way to use delegates, Finish the stopwatch app, Respond to swipe and keyboard input, Respond to swipe and keyboard input, Handling user input, Finish the stopwatch app

adding, Use a standard name when you add a method to raise an event

adding for button in Windows Desktop app, Rebuild your app for Windows Desktop

adding to controls to interact with player, Make the Start button work, Add code to make your controls interact with the player

automatic, The IDE generates event handlers for you automatically

Bullet Points, A callback is just a way to use delegates

excuse management program, You can help Brian out by building a program to manage his excuses

for Birthday Party project controls, Build the Part y Planner version 2.0

for key presses, swipes, and taps in Invaders lab, Handling user input

hooking up, One event, multiple handlers

how they work, One object raises its event, others listen for it...

keyboard, for Invaders game, Respond to swipe and keyboard input

page root, for swipes and taps, Respond to swipe and keyboard input

private or public keyword with, Private fields and methods can only be accessed from inside the class

returning something other than void, Use a standard name when you add a method to raise an event

stopwatch app user control, Finish the stopwatch app, Finish the stopwatch app

TextChanged event handler for TextBox, Build a slightly less simple text editor

types of, Use a standard name when you add a method to raise an event

event keyword, Connecting the dots

events, INotifyPropertyChanged lets bound objects send updates, Modify MenuMaker to notify you when the GeneratedDate property changes, When an EVENT occurs...objects listen, When an EVENT occurs...objects listen, One object raises its event, others listen for it..., One object raises its event,

others listen for it..., Connecting the dots, Connecting the dots, Connecting the dots, Connecting the dots, Windows Forms use many different events, One event, multiple handlers, XAML controls use routed events, Create an app to explore routed events, Connecting event senders with event listeners, Connecting event senders with event listeners, An object can subscribe to an event..., An object can subscribe to an event..., A callback is just a way to use delegates, MessageDialog uses the callback pattern, User controls let you create your own controls, Events alert the rest of the app to state changes, Finish the stopwatch app, INotifyPropertyChanged lets bound objects send updates, Modify MenuMaker to notify you when the GeneratedDate property changes, XAML controls use routed events

(see also event handlers)
callbacks versus, MessageDialog uses the callback pattern
connecting senders with receivers, Connecting event senders with event listeners
creating app to explore routed events, Create an app to explore routed events
defined, When an EVENT occurs...objects listen
delegates, One event, multiple handlers, A callback is just a way to use delegates
forms, Windows Forms use many different events
how they work, One object raises its event, others listen for it...
Model communicating in MVVM apps, User controls let you create your own controls
naming methods when raising events, Connecting the dots
notifying bound controls of changes in Ovservable-Collection, INotifyPropertyChanged lets bound objects send updates, INotifyPropertyChanged lets bound objects send updates
objects subscribing to, An object can subscribe to an event...
raising, Modify MenuMaker to notify you when the GeneratedDate property changes, Connecting the dots, Modify MenuMaker to notify you when the GeneratedDate property changes
raising events with no handlers, Connecting the dots
reference variables, Connecting event senders with event listeners
routed events, use by XAML controls, XAML controls use routed events, XAML controls use routed events
stopwatch app Model, alerting rest of app to state changes, Events alert the rest of the app to state changes
subscription to, One object raises its event, others listen for it..., Connecting the dots, An object can subscribe to an event...
how it works, One object raises its event, others listen for it...
possible probelms with, An object can subscribe to an event...
subscribing classes, Connecting the dots
ViewModel, passing to View in MVVM apps, Finish the stopwatch app
exception, defined, When your program throws an exception, .NET generates an Exception object
exception handling, _, But the program isn't working!, But the program isn't working!, But the program isn't working!, When your program throws an exception, .NET generates an Exception object, When your program throws an exception, .NET generates an Exception object, When your program throws an exception, .NET generates an Exception object, Brian's code did something unexpected, All exception objects inherit from Exception, All exception objects inherit from Exception, All exception objects inherit from Exception, All exception objects inherit from Exception, Use the IDE's debugger

to ferret out exactly what went wrong in the Excuse Manager, Handle exceptions with try and catch, Handle exceptions with try and catch, Handle exceptions with try and catch, Handle exceptions with try and catch, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?, Use the debugger to follow the try/catch flow, If you have code that ALWAYS should run, use a finally block, If you have code that ALWAYS should run, use a finally block, If you have code that ALWAYS should run, use a finally block, If you have code that ALWAYS should run, use a finally block, One class throws an exception that a method in another class can catch, Bees need an OutOfHoney exception, Bees need an OutOfHoney exception, Exception avoidance: implement IDisposable to do your own cleanup, Exception avoidance: implement IDisposable to do your own cleanup, The worst catch block EVER: catch-all plus comments, Temporary solutions are OK (temporarily), Temporary solutions are OK (temporarily), Brian finally gets his vacation..., Make an object serialize itself in its Dispose(), Brian's code did something unexpected, What happens when a method you want to call is risky?, Use the debugger to follow the try/catch flow, If you have code that should ALWAYS run, use a finally block

(see also debugger)

Bullet Points, Bees need an OutOfHoney exception

catch block, Handle exceptions with try and catch, Handle exceptions with try and catch, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?

catching specific exception types, If you have code that ALWAYS should run, use a finally block, Exception avoidance: implement IDisposable to do your own cleanup

DivideByZeroException, But the program isn't working!, All exception objects inherit from Exception

dividing any number by zero, But the program isn't working!

Exception object generated when program throws exception, When your program throws an exception, .NET generates an Exception object

exceptions in constructors, Use the debugger to follow the try/catch flow, Use the debugger to follow the try/catch flow

finalizers, Make an object serialize itself in its Dispose()

finally block, If you have code that ALWAYS should run, use a finally block, If you have code that should ALWAYS run, use a finally block

FormatException, All exception objects inherit from Exception

handled versus unhandled exceptions, If you have code that ALWAYS should run, use a finally block

handling, not burying, The worst catch block EVER: catch-all plus comments

handling versus fixing, Temporary solutions are OK (temporarily)

IDisposable interface, implementing to do cleanup, Exception avoidance: implement IDisposable to do your own cleanup

IndexOutOfRangeException, All exception objects inherit from Exception

invisible to users, Brian finally gets his vacation...

NullReferenceException, But the program isn't working!

OverFlowException, All exception objects inherit from Exception

program stopping with exceptions, If you have code that ALWAYS should run, use a finally block

simple ideas for, Temporary solutions are OK (temporarily)

spotting exceptions, When your program throws an exception, .NET generates an Exception object

throwing and catching exceptions, One class throws an exception that a method in another class can catch

try block, Handle exceptions with try and catch, Handle exceptions with try and catch, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?

unexpected input, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?

unhandled exceptions, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager

using exceptions to find bugs, Brian's code did something unexpected, Brian's code did something unexpected

using statement, Bees need an OutOfHoney exception

why there are so many exceptions, When your program throws an exception, .NET generates an Exception object

Exception objects, When your program throws an exception, .NET generates an Exception object, When your program throws an exception, .NET generates an Exception object, All exception objects inherit from Exception, Use the Exception object to get information about the problem, Use more than one catch block to handle multiple types of exceptions, Bees need an OutOfHoney exception

inheriting from Exception class, All exception objects inherit from Exception

Message property, Use more than one catch block to handle multiple types of exceptions

using to get information about the problem, Use the Exception object to get information about the problem

Excuse Manager project, Trouble at work, You can help Brian out by building a program to manage his excuses, You can help Brian out by building a program to manage his excuses, You can help Brian out by building a program to manage his excuses, You can help Brian out by building a program to manage his excuses, You can help Brian out by building a program to manage his excuses, You can help Brian out by building a program to manage his excuses, Use Stream.Read() to read bytes from a

stream, Brian's running Windows 8, Build Brian a new Excuse Manager app, Brian's code did something unexpected, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Uh oh — the code's still got problems..., Brian's code did something unexpected, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Uh-oh — the code's still got problems...

building the form, You can help Brian out by building a program to manage his excuses

changing to use binary files with serialized Excuse objects, Use Stream.Read() to read bytes from a stream

code problems, Uh oh — the code's still got problems..., Uh-oh — the code's still got problems...

debugging, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager

DialogResult, You can help Brian out by building a program to manage his excuses

event handlers, You can help Brian out by building a program to manage his excuses

Folder button, You can help Brian out by building a program to manage his excuses

Random Excuse button, You can help Brian out by building a program to manage his excuses

rebuilding as Windows Store app, Build Brian a new Excuse Manager app

solution, You can help Brian out by building a program to manage his excuses

turning into Windows Store app, Brian's running Windows 8

unexpected user behavior, Brian's code did something unexpected, Brian's code did something unexpected

executables, Where programs come from

executables, #3. Namespaces and assemblies

Exists( ) method, Use the built-in File and Directory classes to work with files and directories

extend, How would you design a zoo simulator?

Extensible Application Markup Language., Brian's running Windows 8 (see XAML)

extension methods, Extension methods add new behavior to EXISTING classes, Extension methods add new behavior to EXISTING classes, Extending a fundamental type: string, .NET collections are already set up for LINQ

LINQ, .NET collections are already set up for LINQ

strings, Extending a fundamental type: string

F

Factory Method pattern, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

Farmer class (example), Encapsulation keeps your data pristine, Build an application to test the

Farmer class, Use automatic properties to finish the class, Use a constructor to initialize private fields

constructor, using to initialize private fields, Use a constructor to initialize private fields

fully encapsulating, Use automatic properties to finish the class

testing, Build an application to test the Farmer class

fields, Add a method that does something, Fill in the code for your method, An instance uses fields to keep track of things, An instance uses fields to keep track of things, Build a form to interact with the guys, Build a form to interact with the guys, Each option should be calculated individually, Encapsulation means keeping some of the data in a class private, Private fields and methods can only be accessed from inside the class, A few ideas for encapsulating classes, Properties make encapsulation easier, What if we want to change the feed multiplier?, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use the interface keyword to define an interface, Access modifiers change visibility

adding to form, Build a form to interact with the guys, Build a form to interact with the guys

backing fields, set by properties, Properties make encapsulation easier

initializing public fields, What if we want to change the feed multiplier?

interfaces, Use the interface keyword to define an interface

masking, Use a constructor to initialize private fields, Use a constructor to initialize private fields

objects using each other's fields, problems from, Each option should be calculated individually

private, Encapsulation means keeping some of the data in a class private, Use a constructor to initialize private fields, Use a constructor to initialize private fields

public, A few ideas for encapsulating classes

versus methods, An instance uses fields to keep track of things

versus properties, Access modifiers change visibility

with no access, Private fields and methods can only be accessed from inside the class

FIFO (First In, First Out), queues, A queue is FIFO — First In, First Out

File class, Use the built-in File and Directory classes to work with files and directories, Use file dialogs to open and save files (all with just a few lines of code), Use file dialogs to open and save files (all with just a few lines of code), .NET uses Unicode to store characters and text, .NET uses Unicode to store characters and text, .NET uses Unicode to store characters and text, C# can use byte arrays to move data around, C# can use byte arrays to move data around, You can read and write serialized files manually, too, You can read and write serialized files manually, too, Use Stream.Read() to read bytes from a stream, Use Stream.Read() to read bytes from a stream, Use Stream.Read() to read bytes from a stream, Use Stream.Read() to read bytes from a stream, Use Stream.Read() to read bytes from a stream, Use Stream.Read() to read bytes from a stream, Use Stream.Read() to read bytes from a stream, Use Stream.Read() to read bytes from a stream, Use

Stream.Read() to read bytes from a stream

Close( ) method, Use Stream.Read() to read bytes from a stream

Create( ) method, You can read and write serialized files manually, too

OpenWrite( ) method, You can read and write serialized files manually, too

ReadAllBytes( ) method, .NET uses Unicode to store characters and text, C# can use byte arrays to move data around, Use Stream.Read() to read bytes from a stream

ReadAllLines( ) method, Use Stream.Read() to read bytes from a stream

ReadAllText( ) method, Use file dialogs to open and save files (all with just a few lines of code), Use Stream.Read() to read bytes from a stream

static methods, Use Stream.Read() to read bytes from a stream

versus FileInfo class, Use Stream.Read() to read bytes from a stream

WriteAllBytes( ) method, .NET uses Unicode to store characters and text, C# can use byte arrays to move data around, Use Stream.Read() to read bytes from a stream

WriteAllLines( ) method, Use Stream.Read() to read bytes from a stream

WriteAllText( ) method, Use file dialogs to open and save files (all with just a few lines of code), .NET uses Unicode to store characters and text, Use Stream.Read() to read bytes from a stream

file dialogs, Use file dialogs to open and save files (all with just a few lines of code), Use the FileIO class to read and write files

causing WinForms apps to become unresponsive, Use the FileIO class to read and write files

FileInfo class, Use the built-in File and Directory classes to work with files and directories, Use Stream.Read() to read bytes from a stream

versus File class, Use Stream.Read() to read bytes from a stream

file I/O, Brian runs into file trouble, Use the FileIO class to read and write files

FileIO class, Use the FileIO class to read and write files

Windows Store apps, Brian runs into file trouble

filenames, @ in front of, Write text to a file in three simple steps

FileNotFoundException, Exception avoidance: implement IDisposable to do your own cleanup

FileOpenPicker object, Use the FileIO class to read and write files

files, The IDE helps you code, .NET uses streams to read and write data, Use the built-in File and Directory classes to work with files and directories, Use the built-in File and Directory classes to work with files and directories, Use the built-in File and Directory classes to work with files and directories, Use the built-in File and Directory classes to work with files and directories, Writing files usually involves making a lot of decisions

(see also streams)

appending text to, Use the built-in File and Directory classes to work with files and directories

finding out if exists, Use the built-in File and Directory classes to work with files and directories

getting information about, Use the built-in File and Directory classes to work with files and directories

reading from or writing to, Use the built-in File and Directory classes to work with files and directories

unsaved files denoted by * (asterisk) in IDE, The IDE helps you code

writing, Writing files usually involves making a lot of decisions

FileSavePicker object, Use the file pickers to locate file paths

FileStreams, Different streams read and write different things, A FileStream reads and writes bytes to a file, Write text to a file in three simple steps, Things you can do with Directory:, Use a BinaryWriter to write binary data, Use BinaryReader to read the data back in, Use Stream.Read() to read bytes from a stream

BinaryReader, Use BinaryReader to read the data back in

BinaryWriter, Use a BinaryWriter to write binary data

created and managed by StreamWriter, Write text to a file in three simple steps, Things you can do with Directory:

reading and writing bytes to file, A FileStream reads and writes bytes to a file

versus StreamReader and StreamWriter, Use Stream.Read() to read bytes from a stream

Filter property, Dialog boxes are just another WinForms control, Dialog boxes are objects, too, Use file dialogs to open and save files (all with just a few lines of code)

OpenFileDialog object, Dialog boxes are just another WinForms control, Use file dialogs to open and save files (all with just a few lines of code)

SaveFileDialog object, Dialog boxes are objects, too

finalizers, Your last chance to DO something... your object's finalizer, When EXACTLY does a finalizer run?, Finalizers can't depend on stability, Finalizers can't depend on stability, Make an object serialize itself in its Dispose(), Make an object serialize itself in its Dispose(), Make an object serialize itself in its Dispose()

depending on references being valid, Finalizers can't depend on stability

Dispose( ) method, Finalizers can't depend on stability, Make an object serialize itself in its Dispose()

exceptions thrown in, Make an object serialize itself in its Dispose()

fields and methods, Make an object serialize itself in its Dispose()

garbage collection, When EXACTLY does a finalizer run?

finally block, If you have code that ALWAYS should run, use a finally block, If you have code that ALWAYS should run, use a finally block, Bees need an OutOfHoney exception, Exception avoidance: implement IDisposable to do your own cleanup, If you have code that should ALWAYS run, use a finally block

getting with using statements, Bees need an OutOfHoney exception

try/finally, Exception avoidance: implement IDisposable to do your own cleanup

float type, The variable's type determines what kind of data it can store, A variable is like a data to-go cup, When you cast a value that's too big, C# will adjust it automatically

adding int type to, conversion with + operatotr, When you cast a value that's too big, C# will adjust it automatically

FlowLayoutPanel, Use file dialogs to open and save files (all with just a few lines of code), Windows Forms use an object graph set up by the IDE

Controls property, Windows Forms use an object graph set up by the IDE

focused state, animating, Use DoubleAnimation to animate double values

folders., Use the built-in File and Directory classes to work with files and directories, KnownFolders helps you access high-profile folders

(see also directories; files)

high-profile, accessing with KnownFolders, KnownFolders helps you access high-profile folders

foreach loops, Lists shrink and grow dynamically, Generics can store any type, Add a ToString() method to your Card object, too, A stack is LIFO — Last In, First Out, Add the new queries to Jimmy's app

accessing all members in stack of queue, A stack is LIFO — Last In, First Out

from clause in LINQ queries compared to, Add the new queries to Jimmy's app

lists, Lists shrink and grow dynamically, Generics can store any type

using IEnumerable<T>, Add a ToString() method to your Card object, too

for loops, Loops perform an action over and over, Use logical operators to check conditions, Wait, wait! What did that say?

Form1 form, programs without, Any place where you can use a base class, you can use one of its

subclasses instead

FormatException, All exception objects inherit from Exception

Form object, Windows Forms use an object graph set up by the IDE, Use the IDE to explore the object graph

forms, Build a form to interact with the guys, Build a form to interact with the guys, Build a form to interact with the guys, A few ideas for designing intuitive classes, Objects use references to talk to each other, Windows Forms use many different events

- adding buttons, Build a form to interact with the guys, A few ideas for designing intuitive classes
- adding method, Build a form to interact with the guys
- adding variables, Build a form to interact with the guys
- as objects, Objects use references to talk to each other
- events, Windows Forms use many different events

Frame object, Use the IDE to explore app page navigation

Frame property, XAML Pages, Use the IDE to explore app page navigation

from clause, LINQ is simple, but your queries don't have to be, Add the new queries to Jimmy's app, LINQ can combine your results into groups

fully qualified names, Anatomy of a program

functions, An abstract method doesn't have a body

G

Game Over text, adding to Windows Store game, Controls make the game work, Controls make the game work

garbage collection, If there aren't any more references, your object gets garbage-collected, Where no object has gone before, Your last chance to DO something... your object's finalizer, When EXACTLY does a finalizer run?, Make an object serialize itself in its Dispose()

- code that automatically triggers, caution with, Your last chance to DO something... your object's finalizer
- finalizers, When EXACTLY does a finalizer run?

GC.Collect( ) method, When EXACTLY does a finalizer run?, Make an object serialize itself in its Dispose()

GDI+ graphics, Brian's running Windows 8

generic collections, Generics can store any type, Generics can store any type, And yet MORE collection types...

generic data types, Generics can store any type

get accessor, Properties make encapsulation easier, Use a constructor to initialize private fields, Get

a little practice using interfaces, You can find out if a class implements a certain interface with “is”

interface properties, You can find out if a class implements a certain interface with “is”

interfaces with get accessor without set accssor, Get a little practice using interfaces

GetFiles( ) method, Use the built-in File and Directory classes to work with files and directories

GetLastAccessTime( ) method, Use the built-in File and Directory classes to work with files and directories

GetLastWriteTime( ) method, Use the built-in File and Directory classes to work with files and directories

GetType( ) method, Type class, #5. The Type class and GetType()

Go Fish! card game, Build a program that uses a dictionary

Go To Definition, IDisposable makes sure your objects are disposed of properly, Use async methods to find and open files

finding information about class not in your project, Use async methods to find and open files

goto statements, ...more basics...

GPS navigation system, How Mike’s car navigation system thinks about his problems

gradients, Use properties to change how the controls look

adding to Canvas control, Use properties to change how the controls look

gradients, adding to XAML control, Controls make the game work

graphical user interface, A better solution...brought to you by objects! (see GUI)

greater than operator (>), Set up conditions and see if they’re true

grids for Windows Store app page, Start with a blank application, Set up the grid for your page, Set up the grid for your page, Add controls to your grid, Use the grid system to lay out app pages, XAML controls can contain text...and more

adding controls to, Set up the grid for your page, Add controls to your grid

setting up, Start with a blank application, Set up the grid for your page

StackPanel versus, XAML controls can contain text...and more

GridView controls, Use semantic zoom to navigate your data, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

implementing semantic zoom, Use semantic zoom to navigate your data

GroupBox control, We need a BirthdayParty class

group by clause, Combine Jimmy’s values into groups, Jimmy saved a bunch of dough

group clause, Jimmy saved a bunch of dough

group keyword, LINQ can combine your results into groups, Combine Jimmy’s values into groups

GUI (Graphical User Interface), A better solution...brought to you by objects!, Here's what your GUI should look like

labs, #1 A Day at the Races, Here's what your GUI should look like

guys (Two Guys project), Build a class to work with some guys, A few ideas for designing intuitive classes

GZipStream object, Different streams read and write different things

H

Handled property, RoutedEventArgs object, XAML controls use routed events, XAML controls use routed events

Head First Labs website, downloading solutions from, A better solution...brought to you by objects!

heap, Thanks for the memory, What's on your program's mind, The stack vs. the heap: more on memory

versus stack, The stack vs. the heap: more on memory

Hebrew letters, .NET uses Unicode to store characters and text

heights and widths, Windows Store app page, Rows and columns can resize to match the page size, Rows and columns can resize to match the page size

"Hello World" program, building from command line, ...so what did I just do?

hexadecimal, Let's serialize and deserialize a deck of cards, Working with binary files can be tricky, Use file streams to build a hex dumper

working with, Use file streams to build a hex dumper

hex dump, Working with binary files can be tricky, Use file streams to build a hex dumper, StreamReader and StreamWriter will do just fine (for now)

StreamReader and StreamWriter, StreamReader and StreamWriter will do just fine (for now)

using file streams to build hex dumper, Use file streams to build a hex dumper

Hide and Seek game, Keep your eyes open for polymorphism in the next exercise!

hiding methods, A subclass can hide methods in the superclass, Use different references to call hidden methods, Use different references to call hidden methods, Use the override and virtual keywords to inherit behavior

overriding versus, A subclass can hide methods in the superclass

using different reference to call hidden methods, Use different references to call hidden methods

using new keyword, Use different references to call hidden methods

hierarchy, Build up your class model by starting general and getting more specific, Create the class hierarchy, Create the class hierarchy

creating class hierarchy, Create the class hierarchy

defined, Create the class hierarchy

HitTestVisible property, Add code to make your controls interact with the player

HorizontalAlignment property, controls, Add controls to your grid, Use properties to change how the controls look

house model exercise, Keep your eyes open for polymorphism in the next exercise!, Keep your eyes open for polymorphism in the next exercise!

playing hide-and-seek, Keep your eyes open for polymorphism in the next exercise!

hovering over a variable during debugging, Use the debugger to see your variables change

I

IClown interface, Get a little practice using interfaces, Upcasting and downcasting work with interfaces, too, Access modifiers change visibility

access modifiers, Access modifiers change visibility

extending, Upcasting and downcasting work with interfaces, too

ICollection<T> interface, LINQ is versatile

IComparable interface, IComparable<Duck> helps your list sort its ducks

IComparer interface, Use IComparer to tell your List how to sort, Create an instance of your comparer object, Create an instance of your comparer object, IComparer can do complex comparisons, IComparer can do complex comparisons

complex comparisons, IComparer can do complex comparisons

creating instance, Create an instance of your comparer object

multiple classes, Create an instance of your comparer object

SortBy field, IComparer can do complex comparisons

IDE (Integrated Development Environment), Why you should learn C#, When you're doing this..., ...the IDE does this, Where programs come from, Where programs come from

(see also Visual Studio IDE)

creating solutions (.sln files), Where programs come from

editing program files, Where programs come from

making changes in, and IDE changes to default files, ...the IDE does this

what it does in application development, When you're doing this...

IDisposable interface, IDisposable makes sure your objects are disposed of properly, Avoid filesystem errors with using statements, Exception avoidance: implement IDisposable to do your own cleanup, Exception avoidance: implement IDisposable to do your own cleanup, Dispose() works with

using; finalizers work with garbage collection, Finalizers can't depend on stability

avoiding exceptions, Exception avoidance: implement IDisposable to do your own cleanup

Dispose( ) as alternative to finalizers, Finalizers can't depend on stability

streams implementing, Avoid filesystem errors with using statements

IEnumerable interface, Add a ToString() method to your Card object, too, You can upcast an entire list using IEnumerable, LINQ can pull data from multiple sources, .NET collections are already set up for LINQ, LINQ is versatile, #7. Using yield return to create enumerable objects

foreach loops using, Add a ToString() method to your Card object, too

ICollection<T> interface and, LINQ is versatile

upcasting entire list with, You can upcast an entire list using IEnumerable

IEnumerator interface, #7. Using yield return to create enumerable objects

IEnumerator<T> interface, #7. Using yield return to create enumerable objects

IEquatable<T> interface, #6. Equality, IEquatable, and Equals()

if/else statements, if/else statements make decisions, Make each button do something, Set up conditions and see if they're true, Set up conditions and see if they're true, Use logical operators to check conditions, Use logical operators to check conditions, Build the Part y Planner version 2.0, Writing files usually involves making a lot of decisions

checking CakeWriting.Length (example), Build the Part y Planner version 2.0

practice with, Make each button do something, Set up conditions and see if they're true, Use logical operators to check conditions, Use logical operators to check conditions

seting up conditions and checking if they're true, Set up conditions and see if they're true

if statements, Give your classes a natural structure, When you call a method, the arguments must be compatible with the types of the parameters, Writing files usually involves making a lot of decisions, Rename a variable

consolidating conditional expressions, Rename a variable

in CandyController class method (example), Give your classes a natural structure

IL (Intermediate Language), ...so what did I just do?

increment operator (++), A few useful types, ...more basics...

index (arrays), A special case: arrays

indexers, #7. Using yield return to create enumerable objects

IndexOutOfRangeException, When your program throws an exception, .NET generates an Exception object, All exception objects inherit from Exception

inequality operator (!=), Set up conditions and see if they're true, Controls are objects, just like any other object

infinite loops, Use logical operators to check conditions

inheritance, One more thing...can you add a $100 fee for parties over 12?, Build up your class model by starting general and getting more specific, How would you design a zoo simulator?, How would you design a zoo simulator?, How would you design a zoo simulator?, Think about how to group the animals, Create the class hierarchy, Create the class hierarchy, Use a colon to inherit from a base class, We know that inheritance adds the base class fields, properties, and methods to the subclass..., We know that inheritance adds the base class fields, properties, and methods to the subclass..., Any place where you can use a base class, you can use one of its subclasses instead, Any place where you can use a base class, you can use one of its subclasses instead, Any place where you can use a base class, you can use one of its subclasses instead, Use the override and virtual keywords to inherit behavior, A subclass can access its base class using the base keyword, When a base class has a constructor, your subclass needs one, too, Now you're ready to finish the job for Kathleen!, Use inheritance to extend the bee management system, We can use inheritance to create classes for different types of bees, Interfaces can inherit from other interfaces, The RoboBee 4000 can do a worker bee's job without using valuable honey, An abstract method doesn't have a body, An abstract method doesn't have a body, Extension methods add new behavior to EXISTING classes

(see also interfaces)

as principle of OOP, An abstract method doesn't have a body

base class method subclass needs to modify, We know that inheritance adds the base class fields, properties, and methods to the subclass...

building class model from general to more specific, Build up your class model by starting general and getting more specific

classes you can't inherit from, Extension methods add new behavior to EXISTING classes

class hierarchy, Hive Simulator, We can use inheritance to create classes for different types of bees

class that contains entry point, Any place where you can use a base class, you can use one of its subclasses instead

constructors for base class and subclass, When a base class has a constructor, your subclass needs one, too

creating class hierarchy, Create the class hierarchy

designing zoo simulator, How would you design a zoo simulator?

each subclass extending its base class, Create the class hierarchy

interface, Interfaces can inherit from other interfaces

interface, class implementing, The RoboBee 4000 can do a worker bee's job without using valuable honey

looking for classes with much in common, Think about how to group the animals

multiple, An abstract method doesn't have a body

Party base class for DinnerParty and BirthdayParty classes, Now you're ready to finish the job for

Kathleen!

passing instance of subclass, Any place where you can use a base class, you can use one of its subclasses instead

subclass accessing base class using base keyword, A subclass can access its base class using the base keyword

subclasses, We know that inheritance adds the base class fields, properties, and methods to the subclass...

terminology, How would you design a zoo simulator?

using colon to inherit from a base class, Use a colon to inherit from a base class

using override and virtual keywords to inherit behavior, Use the override and virtual keywords to inherit behavior

using subclass in place of base class, Any place where you can use a base class, you can use one of its subclasses instead

using to avoid duplicate code in subclasses, How would you design a zoo simulator?

using to extend bee management system (example), Use inheritance to extend the bee management system

inherit, defined, Build up your class model by starting general and getting more specific

InitialDirectory propery, OpenFileDialog, Dialog boxes are just another WinForms control, Use file dialogs to open and save files (all with just a few lines of code)

initialization, There's an easier way to initialize objects

InitializeComponent( ) method, Use a constructor to initialize private fields

INotifyPropertyChanged interface, INotifyPropertyChanged lets bound objects send updates, User controls let you create your own controls, INotifyPropertyChanged lets bound objects send updates, User controls let you create your own controls

instances, When you create a new object from a class, it's called an instance of that class, When you create a new object from a class, it's called an instance of that class, A little advice for the code exercises, An instance uses fields to keep track of things, An instance uses fields to keep track of things, Let's create some instances!, Create a project for your guys

creating, Let's create some instances!, Create a project for your guys

defined, When you create a new object from a class, it's called an instance of that class

fields, An instance uses fields to keep track of things

keeping track of things, An instance uses fields to keep track of things

requirement for, non-static versus static methods, A little advice for the code exercises

instantiation, interfaces, You can't instantiate an interface, but you can reference an interface

integers, using in code, Objects use variables, too

Integrated Development Environment (IDE)., Why you should learn C# (see IDE; Visual Studio IDE)

IntelliSense (in Visual Studio), The IDE helps you code

interface keyword, Use the interface keyword to define an interface

interfaces, An interface tells a class that it must implement certain methods and properties, An interface tells a class that it must implement certain methods and properties, An interface tells a class that it must implement certain methods and properties, Use the interface keyword to define an interface, Use the interface keyword to define an interface, Use the interface keyword to define an interface, Use the interface keyword to define an interface, Now you can create an instance of NectarStinger that does both jobs, Now you can create an instance of NectarStinger that does both jobs, Classes that implement interfaces have to include ALL of the interface's methods, Get a little practice using interfaces, Get a little practice using interfaces, Get a little practice using interfaces, You can't instantiate an interface, but you can reference an interface, You can't instantiate an interface, but you can reference an interface, You can find out if a class implements a certain interface with "is", You can find out if a class implements a certain interface with "is", Interfaces can inherit from other interfaces, is tells you what an object implements; as tells the compiler how to treat your object, Upcasting works with both objects and interfaces, Upcasting and downcasting work with interfaces, too, Upcasting and downcasting work with interfaces, too, Upcasting and downcasting work with interfaces, too, Upcasting and downcasting work with interfaces, too, Upcasting and downcasting work with interfaces, too, Upcasting and downcasting work with interfaces, too, Access modifiers change visibility, Access modifiers change visibility, Access modifiers change visibility, An abstract class is like a cross between a class and an interface, An abstract method doesn't have a body, An abstract method doesn't have a body, An abstract method doesn't have a body, Keep your eyes open for polymorphism in the next exercise!, Generics can store any type, Extension methods add new behavior to EXISTING classes

- abstract classes and, An abstract class is like a cross between a class and an interface, An abstract method doesn't have a body
- abstract methods in, An abstract method doesn't have a body
- allowing use of class in more than one situation, Now you can create an instance of NectarStinger that does both jobs
- avoiding ambiguity with, An abstract method doesn't have a body
- colon operator, Now you can create an instance of NectarStinger that does both jobs
- compiler errors, An interface tells a class that it must implement certain methods and properties
- containing statements, Upcasting and downcasting work with interfaces, too
- defining using interface keyword, Use the interface keyword to define an interface
- downcasting, Upcasting and downcasting work with interfaces, too
- easy way to implement, Upcasting and downcasting work with interfaces, too
- extending, Extension methods add new behavior to EXISTING classes
- fields, Use the interface keyword to define an interface

finding out if class implements specific interface, You can find out if a class implements a certain interface with “is”

generic, for working with collections, Generics can store any type

get accessor without a set accssor, Get a little practice using interfaces

IHidingPlace (example), Keep your eyes open for polymorphism in the next exercise!

implementing, Classes that implement interfaces have to include ALL of the interface’s methods

inheriting from other interfaces, Interfaces can inherit from other interfaces

is keyword, You can find out if a class implements a certain interface with “is”, is tells you what an object implements; as tells the compiler how to treat your object

naming, Use the interface keyword to define an interface

new keyword, You can’t instantiate an interface, but you can reference an interface

object references versus interface references, Access modifiers change visibility

public, Use the interface keyword to define an interface

public void method, Get a little practice using interfaces

references, You can’t instantiate an interface, but you can reference an interface, Access modifiers change visibility

why use, Access modifiers change visibility

requiring class to implement methods and properties, An interface tells a class that it must implement certain methods and properties

similarity to contracts, Upcasting and downcasting work with interfaces, too

upcasting, Upcasting works with both objects and interfaces, Upcasting and downcasting work with interfaces, too

void method, Get a little practice using interfaces

why use, Upcasting and downcasting work with interfaces, too, Access modifiers change visibility

Intermediate Language (IL), ...so what did I just do?

internal access modifier, There’s more than just public and private, Handling user input, ...so what did I just do?

Internet Explorer (IE), About option, Use delegates to use the Windows settings charm

int type, You have to assign values to variables before you use them, The variable’s type determines what kind of data it can store, A variable is like a data to-go cup, 10 pounds of data in a 5-pound bag, Even when a number is the right size, you can’t just assign it to any variable, When you cast a value that’s too big, C# will adjust it automatically, When you cast a value that’s too big, C# will adjust it automatically, When you call a method, the arguments must be compatible with the types of the parameters, Objects use variables, too, Objects use variables, too

adding to float type, conversion with + operatotr, When you cast a value that's too big, C# will adjust it automatically

assigning value, Objects use variables, too

attempting to assign decimal value to int variable, Even when a number is the right size, you can't just assign it to any variable

casting int variable (too big) to byte, When you cast a value that's too big, C# will adjust it automatically

declaring, Objects use variables, too

no automatic conversion to string, When you call a method, the arguments must be compatible with the types of the parameters

invalid arguments error, When you call a method, the arguments must be compatible with the types of the parameters

IRandomAccessStream, Use async methods to find and open files

IsHitTestVisible property, Your game is now playable, XAML controls use routed events, XAML controls use routed events, Create an app to explore routed events, Create an app to explore routed events

is keyword, You can find out if a class implements a certain interface with "is", The RoboBee 4000 can do a worker bee's job without using valuable honey, is tells you what an object implements; as tells the compiler how to treat your object, Downcasting lets you turn your appliance back into a coffee maker

as keyword versus, is tells you what an object implements; as tells the compiler how to treat your object

checking class or interface sublcassed or implemented, The RoboBee 4000 can do a worker bee's job without using valuable honey

IStorageFolder interface, Use async methods to find and open files, Use async methods to find and open files

methods to work with its files, Use async methods to find and open files

IStorageItem interface, Use async methods to find and open files

IsVisible property, Converters can work with many different types, Converters can work with many different types

items in a list, Use a data template to display objects, Use a data template to display objects

ItemsPanelTemplate, using to bind controls to a canvas, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

ItemsSource property, Make your bees fly around a page, Use ItemsPanelTemplate to bind controls to a Canvas, Make your bees fly around a page, Use ItemsPanelTemplate to bind controls to a Canvas

binding items to ListView, GridView, or ListBox controls, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

IValueConverter interface, Converters automatically convert values for binding, Converters automatically convert values for binding

J

join clause, Use join to combine two collections into one sequence, Jimmy saved a bunch of dough, Jimmy saved a bunch of dough, Jimmy saved a bunch of dough

jump statements, ...more basics...

K

Kathleen's Birthday Party Planner., Kathleen does birthday parties, too (see Birthday Party project)

Kathleen's Party Planning program., Kathleen is an event planner (see Dinner Party Planning project)

keyboard event handlers, Respond to swipe and keyboard input, Handling user input

key frame animations, Use object animations to animate object values

key frames, defined, Use object animations to animate object values

keywords, When you call a method, the arguments must be compatible with the types of the parameters, Controls are objects, just like any other object, #3. Namespaces and assemblies

reference for C# keywords, #3. Namespaces and assemblies

KnownFolders class, KnownFolders helps you access high-profile folders

L

Label controls, Make each button do something, Rebuild your app for Windows Desktop, Rebuild your app for Windows Desktop, Controls are objects, just like any other object, Build the Part y Planner version 2.0

adding to Windows Desktop app, Rebuild your app for Windows Desktop

animating, Controls are objects, just like any other object

Birtyday Party project, Build the Part y Planner version 2.0

button updating, Make each button do something

changing properties in Properties window, Rebuild your app for Windows Desktop

labels for objects, Refer to your objects with reference variables (see reference variables)

labels, loop using goto statement and, ...more basics...

labelToChange properties, Windows Store apps use XAML to create UI objects, WPF applications use XAML to create UI objects

labs, C# Lab: A Day at the Races, You'll need three classes and a form, You'll need three classes and

a form, Initialize your arrays of Greyhound and Guy objects, Initialize your arrays of Greyhound and Guy objects, Initialize your arrays of Greyhound and Guy objects, Here's your application architecture, Here's your application architecture, Here's your application architecture, Here's your application architecture, Here's your application architecture, Here's what your GUI should look like, Here's what your GUI should look like, Here's what your GUI should look like, Here's what your GUI should look like, Placing bets, The Finished Product, C# Lab: The Quest, The design: building the form, The architecture: using the objects, Gameplay concerns are separated into the Game object, Building the Game class, Finding common behavior: movement, The Mover class source code, The Player class keeps track of the player, Write the Move() method for the Player, Write the Move() method for the Player, Bats, ghosts, and ghouls inherit from the Enemy class, Write the different Enemy subclasses, Write the different Enemy subclasses, Write the different Enemy subclasses, Write the different Enemy subclasses, Weapon inherits from Mover; each weapon inherits from Weapon, Different weapons attack in different ways, Different weapons attack in different ways, Different weapons attack in different ways, Potions implement the IPotion interface, Potions implement the IPotion interface, Potions implement the IPotion interface, The form brings it all together, The form's UpdateCharacters() method moves the PictureBoxes into position, The fun's just beginning!, C# Lab Invaders, Your mission: defend the planet against wave after wave of invaders, Your mission: defend the planet against wave after wave of invaders, The architecture of Invaders, Build out the object model for the Model, Building the InvadersModel class, Building the InvadersModel class, The InvadersModel methods, Filling out the InvadersModel class, Filling out the InvadersModel class, LINQ makes collision detection much easier, Build the Invaders page for the View, Maintain the play area's aspect ratio, Respond to swipe and keyboard input, An AnimatedImage control displays the ships, An AnimatedImage control displays the ships, Use the Settings charm to open a SettingsFlyout, Build the ViewModel, Handling user input, Build the InvadersViewModel methods, The View's updated when the timer ticks, The player's ship can move and die, "Shots fired!", And yet there's more to do...

#1 A Day at the Races, C# Lab: A Day at the Races, You'll need three classes and a form, You'll need three classes and a form, Initialize your arrays of Greyhound and Guy objects, Initialize your arrays of Greyhound and Guy objects, Initialize your arrays of Greyhound and Guy objects, Here's your application architecture, Here's your application architecture, Here's your application architecture, Here's your application architecture, Here's your application architecture, Here's what your GUI should look like, Here's what your GUI should look like, Here's what your GUI should look like, Here's what your GUI should look like, Placing bets, The Finished Product

application architecture, Here’s your application architecture

Bet class, Initialize your arrays of Greyhound and Guy objects

Bet object, Here’s what your GUI should look like

Betting Parlor groupbox, Placing bets

dogs array, Here’s your application architecture

finished executable, The Finished Product

Greyhound class, You’ll need three classes and a form

GUI, Here’s what your GUI should look like

Guy class, Initialize your arrays of Greyhound and Guy objects

Guy object, Here’s what your GUI should look like

guys array, Here’s your application architecture

PictureBox control, You’ll need three classes and a form, Here’s your application architecture, Here’s what your GUI should look like

RadioButton controls, Here’s your application architecture

this keyword, Initialize your arrays of Greyhound and Guy objects

#2 The Quest, C# Lab: The Quest, The design: building the form, The architecture: using the objects, Gameplay concerns are separated into the Game object, Building the Game class, Finding common behavior: movement, The Mover class source code, The Player class keeps track of the player, Write the Move() method for the Player, Write the Move() method for the Player, Bats, ghosts, and ghouls inherit from the Enemy class, Write the different Enemy subclasses, Write the different Enemy subclasses, Write the different Enemy subclasses, Write the different Enemy subclasses, Weapon inherits from Mover; each weapon inherits from Weapon, Different weapons attack in different ways, Different weapons attack in different ways, Different weapons attack in different ways, Potions implement the IPotion interface, Potions implement the IPotion interface, Potions implement the IPotion interface, The form brings it all together, The form’s UpdateCharacters() method moves the PictureBoxes into position, The fun’s just beginning!

Bat subclass, Write the different Enemy subclasses

BluePotion class, Potions implement the IPotion interface

Enemy class, Bats, ghosts, and ghouls inherit from the Enemy class

Enemy subclasses, Write the different Enemy subclasses

form, bringing it all together, The form brings it all together

form, building, The design: building the form

form delegating activity to Game object, Gameplay concerns are separated into the Game object

form, UpdateCharacters( ) method, The form's UpdateCharacters() method moves the PictureBoxes into position

Game class, Building the Game class

Ghost subclass, Write the different Enemy subclasses

Ghoul subclass, Write the different Enemy subclasses

ideas for improving the game, The fun's just beginning!

IPotion interface, Potions implement the IPotion interface

Mace subclass, Different weapons attack in different ways

Mover class, Finding common behavior: movement

Mover class source code, The Mover class source code

objects, Player, Enemy, Weapon, and Game, The architecture: using the objects

Player class, The Player class keeps track of the player

Player class Attack( ) method, Write the Move() method for the Player

Player class Move( ) method, Write the Move() method for the Player

RedPotion class, Potions implement the IPotion interface

Sword subclass, Different weapons attack in different ways

Weapon class, Weapon inherits from Mover; each weapon inherits from Weapon

Weapon subclasses, Different weapons attack in different ways

#3 Invaders, C# Lab Invaders, Your mission: defend the planet against wave after wave of invaders, Your mission: defend the planet against wave after wave of invaders, The architecture of Invaders, Build out the object model for the Model, Building the InvadersModel class, Building the InvadersModel class, The InvadersModel methods, Filling out the InvadersModel class, Filling out the InvadersModel class, LINQ makes collision detection much easier, Build the Invaders page for the View, Maintain the play area's aspect ratio, Respond to swipe and keyboard input, An AnimatedImage control displays the ships, An AnimatedImage control displays the ships, Use the Settings charm to open a SettingsFlyout, Build the ViewModel, Handling user input, Build the InvadersViewModel methods, The View's updated when the timer ticks, The player's ship can

move and die, “Shots fired!”, And yet there’s more to do...

additions, And yet there’s more to do...

architecture, The architecture of Invaders

building ViewModel, Build the ViewModel

control for big stars, An AnimatedImage control displays the ships

Game class, Building the InvadersModel class

Game class, filling out, Filling out the InvadersModel class

handling user input, Handling user input

InvadersHelper class for ViewModel, An AnimatedImage control displays the ships

InvadersModel class, Building the InvadersModel class

InvadersModel class, filling out, Filling out the InvadersModel class

InvadersModel class methods, The InvadersModel methods

Invaders page, building for the View, Build the Invaders page for the View

InvadersViewModel class methods, Build the InvadersViewModel methods

LINQ, LINQ makes collision detection much easier

maintaining play area’s aspect ratio, Maintain the play area’s aspect ratio

movements, Your mission: defend the planet against wave after wave of invaders

object model for the Model, Build out the object model for the Model

player’s ship, moving and dying, The player’s ship can move and die

responding to swipe and keyboard input, Respond to swipe and keyboard input

shots fired, “Shots fired!”

types of invaders, Your mission: defend the planet against wave after wave of invaders

using Settings charm to open About popup, Use the Settings charm to open a SettingsFlyout

View, updating when timer ticks, The View’s updated when the timer ticks

lambda expressions, #9. Anonymous types, anonymous methods, and lambda expressions, #9. Anonymous types, anonymous methods, and lambda expressions

Launched event handler, updating, Add process lifetime management to Jimmy’s comics

Length property, arrays, Arrays can contain a bunch of reference variables, too

less than operator (<), Set up conditions and see if they’re true

libraries, #3. Namespaces and assemblies

creating class libraries, #3. Namespaces and assemblies

LIFO (Last In, First Out), stacks, A stack is LIFO — Last In, First Out

line breaks., Make each button do something, XAML controls can contain text...and more

(see also \n; \r)

adding to XAML text controls, XAML controls can contain text...and more

LINQ (Language Integrated Query), _, LINQ can pull data from multiple sources, .NET collections are already set up for LINQ, .NET collections are already set up for LINQ, LINQ makes queries easy, LINQ is simple, but your queries don't have to be, LINQ is simple, but your queries don't have to be, LINQ is versatile, LINQ is versatile, LINQ is versatile, LINQ is versatile, Add the new queries to Jimmy's app, Add the new queries to Jimmy's app, Add the new queries to Jimmy's app, Add the new queries to Jimmy's app, Add the new queries to Jimmy's app, Add the new queries to Jimmy's app, LINQ can combine your results into groups, LINQ can combine your results into groups, LINQ can combine your results into groups, Combine Jimmy's values into groups, Use join to combine two collections into one sequence, Jimmy saved a bunch of dough, Jimmy saved a bunch of dough, Jimmy saved a bunch of dough, LINQ makes collision detection much easier, #10. LINQ to XML

combining results into groups, LINQ can combine your results into groups, Combine Jimmy's values into groups

complex queries with, LINQ is simple, but your queries don't have to be

deferred evaluation of queries, LINQ is versatile

difference from most of C# syntax, LINQ is versatile

extension methods, .NET collections are already set up for LINQ

from clause, Add the new queries to Jimmy's app, LINQ can combine your results into groups

Invaders lab, LINQ makes collision detection much easier

join queries, Jimmy saved a bunch of dough

LINQ to XML, #10. LINQ to XML

modifying items, LINQ is versatile

.NET collections, .NET collections are already set up for LINQ

orderby clause, Add the new queries to Jimmy's app, LINQ can combine your results into groups

performing calculations on collections, LINQ is versatile

pulling data from multiple sources, LINQ can pull data from multiple sources

queries, LINQ makes queries easy

queries, anatomy of, LINQ is simple, but your queries don't have to be

query statements, Add the new queries to Jimmy's app

select clause, Add the new queries to Jimmy's app

Take statement, Add the new queries to Jimmy's app

using join to combine two collections into one query, Use join to combine two collections into one sequence, Jimmy saved a bunch of dough

var keyword and, Jimmy saved a bunch of dough

where clause, Add the new queries to Jimmy's app

LINQPad, Add semantic zoom to Jimmy's app

ListBox controls, Page layout starts with controls, Use ItemsPanelTemplate to bind controls to a Canvas, Page layout starts with controls, Use ItemsPanelTemplate to bind controls to a Canvas

Windows Store Go Fish! app page, Page layout starts with controls, Page layout starts with controls

ListBoxItem object, Use a data template to display objects, Use a data template to display objects

lists, #7. Using yield return to create enumerable objects

using [ ] to return object from, #7. Using yield return to create enumerable objects

List<T> class, Lists make it easy to store collections of...anything, Lists are more flexible than arrays, Lists are more flexible than arrays, Lists shrink and grow dynamically, Lists shrink and grow dynamically, Generics can store any type, Generics can store any type, Collection initializers are similar to object initializers, Lists are easy, but SORTING can be tricky, Lists are easy, but SORTING can be tricky, IComparable<Duck> helps your list sort its ducks, IComparable<Duck> helps your list sort its ducks, Use IComparer to tell your List how to sort, Create an instance of your comparer object, Create an instance of your comparer object, IComparer can do complex comparisons, Update your foreach loops to let your Ducks and Cards print themselves, Add a ToString() method to your Card object, too, You can upcast an entire list using IEnumerable, You can build your own overloaded methods, A stack is LIFO — Last In, First Out

building class to store deck of cards and form using it, You can build your own overloaded methods

CompareTo( ) method, IComparable<Duck> helps your list sort its ducks

converting from stacks or queues, A stack is LIFO — Last In, First Out

creating, using collection initializer, Collection initializers are similar to object initializers

dynamically shrinking and growing, Lists shrink and grow dynamically

foreach loop, Lists shrink and grow dynamically, Update your foreach loops to let your Ducks and Cards print themselves

foreach loop using IEnumerable<T>, Add a ToString() method to your Card object, too

IComparable interface, IComparable<Duck> helps your list sort its ducks

IComparer interface, Use IComparer to tell your List how to sort

IComparer interface, complex comparisons with, IComparer can do complex comparisons

IComparer interface, creating an instance, Create an instance of your comparer object

IComparer interface, multiple implementations, Create an instance of your comparer object

sorting, Lists are easy, but SORTING can be tricky

Sort( ) method, Lists are easy, but SORTING can be tricky

storing any type, Generics can store any type

things you can do with, Lists are more flexible than arrays

upcasting, using IEnumerable<T>, You can upcast an entire list using IEnumerable

versus arrays, Lists are more flexible than arrays

versus enums, Generics can store any type

ListView controls, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu, Use the new keyword to create anonymous types, Use semantic zoom to navigate your data, Use ItemsPanelTemplate to bind controls to a Canvas, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu, Use ItemsPanelTemplate to bind controls to a Canvas

app managing Jimmy's comic collection, Use the new keyword to create anonymous types

data binding to properties in MenuMaker (example), Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

implementing semantic zoom, Use semantic zoom to navigate your data

populating using one-way data binding, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

ListViewItem object, Use a data template to display objects, Use a data template to display objects

literals, The variable's type determines what kind of data it can store, Where no object has gone before

logical operators, Set up conditions and see if they're true, ...more basics..., ...more basics..., #3. Namespaces and assemblies

&, |, and ^, ...more basics...

combining with =, #3. Namespaces and assemblies

using to check conditions, Set up conditions and see if they're true

long type, The variable's type determines what kind of data it can store, A variable is like a data to-go cup, C# does some casting automatically

converting to a string, C# does some casting automatically

loops, Loops perform an action over and over, Use logical operators to check conditions, Use logical operators to check conditions, Wait, wait! What did that say?, A stack is LIFO — Last In, First Out, ...more basics...

adding while and for loops to program, Use logical operators to check conditions, Use logical operators to check conditions

infinite loops, Use logical operators to check conditions

continue and break keywords, ...more basics...

foreach., A stack is LIFO — Last In, First Out (see foreach loops)

using for Windows desktop app animation, Wait, wait! What did that say?

lowercasing, Use a constructor to initialize private fields

M

Main( ) method, Rebuild your app for Windows Desktop, Your desktop app knows where to start

MainPage class, Windows Store apps use XAML to create UI objects, WPF applications use XAML to create UI objects

Margin property, Use properties to change how the controls look, Use the grid system to lay out app pages

Button controls, Use properties to change how the controls look

Grid control, Use the grid system to lay out app pages

Margin property, Button controls, Add controls to your grid

masking fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields

math operators, A few useful types

members (class), There's more than just public and private

memory, Thanks for the memory, The stack vs. the heap: more on memory

stack versus heap, The stack vs. the heap: more on memory

MemoryStreams, Different streams read and write different things

MessageBox.Show( ) method, You can change your program's entry point, Even when a number is the right size, you can't just assign it to any variable, C# does some casting automatically, Build a program that uses a dictionary

argument type not matching parameter type, C# does some casting automatically

conversion of \n character to line breaks, Build a program that uses a dictionary

MessageDialog object, Windows Store apps use await to be more responsive, C# programs can use await to be more responsive

Message property, Exception object, All exception objects inherit from Exception, Use more than one catch block to handle multiple types of exceptions

methods, You've set the stage for the game, You've set the stage for the game, What you'll do next, Add a method that does something, Add a method that does something, Use the IDE to create your own method, Anatomy of a program, Let's take a closer look at your code, Two classes can be in the same namespace, Make each button do something, Rebuild your app for Windows Desktop, Your desktop app knows where to start, Mike's Navigator class has methods to set and modify routes, Mike's Navigator class has methods to set and modify routes, Some methods have a return value, You use a class to build an object, A little advice for the code exercises, An instance uses fields to keep track of things, You can use class and method names to make your code intuitive, Build a form to interact with the guys, When you call a method, the arguments must be compatible with the types of the parameters, Objects use references to talk to each other, But is the RealName field REALLY protected?, Private fields and methods can only be accessed from inside the class, A few ideas for encapsulating classes, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Different animals make different noises, Create the class hierarchy, A subclass can override methods to change or replace methods it inherited, A subclass can hide methods in the superclass, Use different references to call hidden methods, Use the override and virtual keywords to inherit behavior, An interface tells a class that it must implement certain methods and properties, Classes that implement interfaces have to include ALL of the interface's methods, An abstract class is like a cross between a class and an interface, An abstract method doesn't have a body, Enums let you

represent numbers with names, Use out parameters to make a method return more than one value, Pass by reference using the ref modifier, Use optional parameters to set default values, Extension methods add new behavior to EXISTING classes, A delegate STANDS IN for an actual method, #8. Refactoring

abstract, An abstract class is like a cross between a class and an interface, An abstract method doesn't have a body

accessing private fields with public methods, Private fields and methods can only be accessed from inside the class

adding for form, Build a form to interact with the guys

adding from other namespaces, Rebuild your app for Windows Desktop

arguments matching types of parameters, When you call a method, the arguments must be compatible with the types of the parameters

calling most specific, Create the class hierarchy

calling on classes in same namespace, Two classes can be in the same namespace

code between { } (curly braces), Make each button do something

creating using IDE, You've set the stage for the game, Add a method that does something

defined, You've set the stage for the game, Add a method that does something, Let's take a closer look at your code

delegates standing in for, A delegate STANDS IN for an actual method

extension, Extension methods add new behavior to EXISTING classes (see extension methods)

extracting, #8. Refactoring

filling in code for, What you'll do next, Use the IDE to create your own method

get and set accessors versus, Use a constructor to initialize private fields

hidden, using different references to call, Use different references to call hidden methods

hiding versus overriding, A subclass can hide methods in the superclass

implementing interfaces, Classes that implement interfaces have to include ALL of the interface's methods

in desktop app class, Your desktop app knows where to start

interface requiring class to implement, An interface tells a class that it must implement certain methods and properties

naming, You can use class and method names to make your code intuitive

Navigator class (example), Mike's Navigator class has methods to set and modify routes

object, You use a class to build an object

optional parameters, using to set default values, Use optional parameters to set default values

overloaded, Enums let you represent numbers with names (see overloaded methods)

overriding, Different animals make different noises, A subclass can override methods to change or replace methods it inherited

passing arguments by reference, Pass by reference using the ref modifier

private, But is the RealName field REALLY protected?

public, A few ideas for encapsulating classes

public, capitalization in names, Use a constructor to initialize private fields

returning more than one value with out parameters, Use out parameters to make a method return more than one value

return values, Mike's Navigator class has methods to set and modify routes

signature, Use a constructor to initialize private fields

static., A little advice for the code exercises (see static methods)

this keyword with, Objects use references to talk to each other

using override and virtual keywords, Use the override and virtual keywords to inherit behavior

versus fields, An instance uses fields to keep track of things

with no return value, Use a constructor to initialize private fields

Microsoft Download Center, Use the Remote Debugger to sideload your app

Microsoft reference for C#, #3. Namespaces and assemblies

mileage and reimbursement calculator, When you call a method, the arguments must be compatible with the types of the parameters

Model, MVVM lets you design for binding and data, User controls let you create your own controls, Events alert the rest of the app to state changes, Finish the stopwatch app

(see also MVVM pattern)

rules for MVVM apps, Finish the stopwatch app

stopwatch app, events alerting app to state changes, Events alert the rest of the app to state changes

using Model statement at top of ViewModel classes, User controls let you create your own controls

Model-View-Controller (MVC) pattern, User controls let you create your own controls

Model-View-ViewModel pattern., _ (see MVVM pattern)

monetary values, decimal type for, You're going to build a program for Kathleen

monitors, different, Windows Store apps on, Use the grid system to lay out app pages

multiple inheritance, An abstract method doesn't have a body

multiplication operator., A few useful types (see * (asterisk), under Symbols)

MVC (Model-View-Controller) pattern, User controls let you create your own controls

MVVM (Model-View-ViewModel) pattern, _, Do you design for binding or for working with data?, MVVM lets you design for binding and data, Use the MVVM pattern to start building the basketball roster app, User controls let you create your own controls, User controls let you create your own controls, User controls let you create your own controls, User controls let you create your own controls, User controls let you create your own controls, MVVM means thinking about the state of the app, Start building the stopwatch app's Model, Finish the stopwatch app, Finish the stopwatch app, Build an analog stopwatch using the same ViewModel, Build an analog stopwatch using the same ViewModel, Make your bees fly around a page, Use ItemsPanelTemplate to bind controls to a Canvas, The architecture of Invaders, User controls let you create your own controls, Build an analog stopwatch using the same ViewModel, Build an analog stopwatch using the same ViewModel, Make your bees fly around a page, Use ItemsPanelTemplate to bind controls to a Canvas

- animating bees and stars, program for, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas
- debate between Model and ViewModel, User controls let you create your own controls
- decisions about implementation, Finish the stopwatch app
- decoupling of components, Build an analog stopwatch using the same ViewModel, Build an analog stopwatch using the same ViewModel
- designing for binding and data, MVVM lets you design for binding and data
- designing for binding or working with data, Do you design for binding or for working with data?
- dividing up concerns of the program, User controls let you create your own controls
- enabling easier handling of code in future, User controls let you create your own controls
- image animation and, Make your bees fly around a page, Make your bees fly around a page
- Invaders game, The architecture of Invaders
- Model communicating with rest of app, User controls let you create your own controls
- rules for building apps, Finish the stopwatch app
- state of the app, MVVM means thinking about the state of the app
- stopwatch, analog, building with ViewModel, Build an analog stopwatch using the same ViewModel, Build an analog stopwatch using the same ViewModel
- stopwatch for BasketballRoster project, Start building the stopwatch app's Model
- user controls, User controls let you create your own controls, User controls let you create your own controls
- using to start building basketball roster app, Use the MVVM pattern to start building the basketball roster app

N

\n (line feed character), Make each button do something, So what did you just build?, The variable's type determines what kind of data it can store, Build a program that uses a dictionary, Write text to a file in three simple steps, Things you can do with Directory:

Name box, Properties window, Add controls to your grid, Use properties to change how the controls look

namespaces, The .NET Framework gives you the right tools for the job, Anatomy of a program, Two classes can be in the same namespace, Use logical operators to check conditions, Rebuild your app for Windows Desktop, Your desktop app knows where to start, You can build your own overloaded methods, Use static resources to declare your objects in XAML, Take your Guy Serializer for a test drive, #3. Namespaces and assemblies, Use static resources to declare your objects in XAML, Take your Guy Serializer for a test drive

- and assemblies, #3. Namespaces and assemblies
- classes in, Two classes can be in the same namespace
- generated by IDE for Windows Desktop app, Rebuild your app for Windows Desktop, Your desktop app knows where to start
- in C# programs, Anatomy of a program, You can build your own overloaded methods
- reasons for using, Take your Guy Serializer for a test drive, Take your Guy Serializer for a test drive
- Windows Runtime and .NET Framework tools, The .NET Framework gives you the right tools for the job
- XML, Use static resources to declare your objects in XAML, Use static resources to declare your objects in XAML

NavigatedFrom event handler, Add process lifetime management to Jimmy's comics

navigating data, building apps for, The IDE's Split App template helps you build apps for navigating data

navigation, page-based, in Windows Store apps, Jimmy could use some help

Navigation project, How Mike thinks about his problems, Mike's Navigator class has methods to set and modify routes, Mike's navigator program could use better encapsulation

- better encapsulation for Route class, Mike's navigator program could use better encapsulation
- Navigator class, methods to set and modify routes, Mike's Navigator class has methods to set and modify routes

.NET Framework, Why you should learn C#, Finish the method and run your program, The .NET Framework gives you the right tools for the job, The .NET Framework gives you the right tools for the job, The .NET Framework gives you the right tools for the job, Anatomy of a program, Anatomy of a program, Use logical operators to check conditions, A little advice for the code exercises, Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!, Controls are objects, just like any other object, Use a constructor to initialize private fields, There's more than just public and private,

Lists make it easy to store collections of...anything, Generics can store any type, You can build your own overloaded methods, Build a program that uses a dictionary, And yet MORE collection types..., .NET uses streams to read and write data, Brian's running Windows 8, Two-way binding can get or set the source property, Modify MenuMaker to notify you when the GeneratedDate property changes, KnownFolders helps you access high-profile folders, Take your Guy Serializer for a test drive, When EXACTLY does a finalizer run?, A struct looks like an object..., Extension methods add new behavior to EXISTING classes, .NET collections are already set up for LINQ, Did you know that C# and the .NET Framework can..., Modify MenuMaker to notify you when the GeneratedDate property changes, Take your Guy Serializer for a test drive

built-in classes and assemblies, There's more than just public and private

collections, Lists make it easy to store collections of...anything, .NET collections are already set up for LINQ

events, raising, pattern for, Modify MenuMaker to notify you when the GeneratedDate property changes, Modify MenuMaker to notify you when the GeneratedDate property changes

for Windows Store apps, Brian's running Windows 8

garbage collection, When EXACTLY does a finalizer run?

generic collections, And yet MORE collection types...

generic interfaces for working with collections, Generics can store any type

KnownFolders class, KnownFolders helps you access high-profile folders, Take your Guy Serializer for a test drive, Take your Guy Serializer for a test drive

line breaks, adding with Environment.NewLine, Build a program that uses a dictionary

Math.Min( ) method, A little advice for the code exercises

namespaces, The .NET Framework gives you the right tools for the job, Anatomy of a program, Controls are objects, just like any other object

.NET for Windows Desktop, The .NET Framework gives you the right tools for the job

.NET for Windows Store apps, The .NET Framework gives you the right tools for the job

ObservableCollection<T> class, for data binding, Two-way binding can get or set the source property

overloaded methods, in built-in classes and objects, You can build your own overloaded methods

pre-built structures, Why you should learn C#

Random class, Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!

sealed classes, Extension methods add new behavior to EXISTING classes

streams, readinga and writing data, .NET uses streams to read and write data

structs, A struct looks like an object...

System namespace, use of, Use logical operators to check conditions

System.Windows.Forms namespace, Use a constructor to initialize private fields

using statement, to use animation code, Finish the method and run your program

using tools in C# code, Anatomy of a program

NetworkStreams, Different streams read and write different things

new keyword, Mike can use objects to solve his problem, Use different references to call hidden methods, You can't instantiate an interface, but you can reference an interface, Use the new keyword to create anonymous types, Use the new keyword to create anonymous types

interfaces, You can’t instantiate an interface, but you can reference an interface

using to create anonymous types, Use the new keyword to create anonymous types, Use the new keyword to create anonymous types

using when hiding methods, Use different references to call hidden methods

new statements, Thanks for the memory, A special case: arrays, Use a constructor to initialize private fields

creating array object, A special case: arrays

creating class instances, Thanks for the memory

using contructor with, Use a constructor to initialize private fields

Normal state (controls), Visual states make controls respond to changes

NOT operator (!), A few useful types

nullable types, Use nullable types when you need nonexistent values, Nullable types help you make your programs more robust

helping to make programs more robust, Nullable types help you make your programs more robust

null coalescing operator (??), ...more basics...

null keyword, Where no object has gone before

NullReferenceException, But the program isn’t working!

numbers, The variable’s type determines what kind of data it can store, The variable’s type determines what kind of data it can store, Enums let you represent numbers with names

converting between decimal and binary, The variable’s type determines what kind of data it can store

data types for, The variable’s type determines what kind of data it can store

representing with names, using enums, Enums let you represent numbers with names

NumericUpDown controls, So what did you just build?, When you call a method, the arguments must be compatible with the types of the parameters, Build the Part y Planner version 2.0

Birthday Party project, Build the Part y Planner version 2.0

O

ObjectAnimationUsingKeyFrames animation, Use object animations to animate object values

object graphs, Windows Forms use an object graph set up by the IDE, Use the IDE to explore the object graph, The whole object graph is serialized to XML

using IDE to explore, Use the IDE to explore the object graph

whole graph serialized to XML with data contract serialization, The whole object graph is serialized to XML

object initializers, There's an easier way to initialize objects, A few ideas for designing intuitive classes, What if we want to change the feed multiplier?, Use a constructor to initialize private fields

initializing public fields and properties in, What if we want to change the feed multiplier?

object-oriented programming (OOP), An abstract method doesn't have a body, Use Stream.Read() to read bytes from a stream

object references, versus interface references, Access modifiers change visibility

objects, , Mike can use objects to solve his problem, You use a class to build an object, A better solution...brought to you by objects!, A little advice for the code exercises, An instance uses fields to keep track of things, Thanks for the memory, Build a class to work with some guys, The variable's type determines what kind of data it can store, When you call a method, the arguments must be compatible with the types of the parameters, Objects use variables, too, Objects use variables, too, Objects use variables, too, Refer to your objects with reference variables, If there aren't any more references, your object gets garbage-collected, Objects use references to talk to each other, Where no object has gone before, Where no object has gone before, Controls are objects, just like any other object, Controls are objects, just like any other object, C# Lab: A Day at the Races, Each option should be calculated individually, It's easy to accidentally misuse your objects, Encapsulation means keeping some of the data in a class private, Interface references work just like object references, Upcasting works with both objects and interfaces, Downcasting lets you turn your appliance back into a coffee maker, But what exactly IS an object's state? What needs to be saved?, Serialization lets you read or write a whole object graph all at once, Your last chance to DO something... your object's finalizer, Values get copied; references get assigned, Structs are value types; objects are reference types, The stack vs. the heap: more on memory, "Captain" Amazing...not so much, Ever wish your objects could think for themselves?, Connecting the dots, XAML controls use routed events, An object can subscribe to an event..., Use object animations to animate object values, XAML controls use routed events

accessing fields inside object, Encapsulation means keeping some of the data in a class private

accidentally misusing, It's easy to accidentally misuse your objects

array of, iterating through, Controls are objects, just like any other object

assigning value, Objects use variables, too

as variables, Objects use variables, too

boxed, The stack vs. the heap: more on memory

building from classes, You use a class to build an object, Build a class to work with some guys

Guy objects (example), Build a class to work with some guys

controls as, Controls are objects, just like any other object

declaring, Objects use variables, too

downcasting, Downcasting lets you turn your appliance back into a coffee maker

encapsulation, C# Lab: A Day at the Races (see encapsulation)

event arguments, Connecting the dots

finalizers, Your last chance to DO something... your object's finalizer (see finalizers)

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

knowing when to respond, Ever wish your objects could think for themselves?

methods versus fields, An instance uses fields to keep track of things

null keyword, Where no object has gone before

object animations to animate object values, Use object animations to animate object values

object tree, XAML controls use routed events, XAML controls use routed events

object type, The variable's type determines what kind of data it can store, When you call a method, the arguments must be compatible with the types of the parameters

- assignments to variables, parameters, or fields with, When you call a method, the arguments must be compatible with the types of the parameters

reading entire with serialization, Serialization lets you read or write a whole object graph all at once

references, Interface references work just like object references

reference variables, Refer to your objects with reference variables (see reference variables)

states, But what exactly IS an object's state? What needs to be saved?

storage in heap memory, Thanks for the memory

subscribing to events, An object can subscribe to an event...

talking to other objects, Objects use references to talk to each other, Where no object has gone before

upcasting, Upcasting works with both objects and interfaces

using each other's fields, problem caused by, Each option should be calculated individually

using to program Navigator class (example), Mike can use objects to solve his problem, A better solution...brought to you by objects!, A little advice for the code exercises

value types versus, Values get copied; references get assigned

versus structs, Structs are value types; objects are reference types, "Captain" Amazing...not so much

ObservableCollection<T> collections, Two-way binding can get or set the source property, Use data binding to build Sloppy Joe a better menu, INotifyPropertyChanged lets bound objects send updates,

Use the new keyword to create anonymous types, Create an app to explore routed events, Use ItemsPanelTemplate to bind controls to a Canvas, Use data binding to build Sloppy Joe a better menu, INotifyPropertyChanged lets bound objects send updates, Create an app to explore routed events, Use ItemsPanelTemplate to bind controls to a Canvas

changes in, firing off event to tell bound controls, INotifyPropertyChanged lets bound objects send updates, INotifyPropertyChanged lets bound objects send updates

changing properties and adding animations to controls, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

using for MenuItems in MenuMaker project, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

Observer pattern, MessageDialog uses the callback pattern

on ... equals clause, Jimmy saved a bunch of dough

OnSuspending( ) event handler, Add process lifetime management to Jimmy's comics

OOP (object-oriented programming), An abstract method doesn't have a body, Use Stream.Read() to read bytes from a stream

OpenFileDialog control, Dialog boxes are just another WinForms control, Use file dialogs to open and save files (all with just a few lines of code)

OpenRead( ) method, Use the built-in File and Directory classes to work with files and directories

OpenWrite( ) method, Use the built-in File and Directory classes to work with files and directories

operators, A few useful types, Combining = with an operator, #3. Namespaces and assemblies

compound, Combining = with an operator

reference for C# operators, #3. Namespaces and assemblies

optional parameters, using to set default values, Use optional parameters to set default values

orderby clause, LINQ is simple, but your queries don't have to be, Add the new queries to Jimmy's app, LINQ can combine your results into groups

OriginalSource property, RoutedEventArgs object, XAML controls use routed events, XAML controls use routed events

OR operator., ...more basics... (see | (pipe symbol), under Symbols)

OR operator (||), You can help Brian out by building a program to manage his excuses

out parameters, Use out parameters to make a method return more than one value, Pass by reference using the ref modifier

making methods return multiple values, Use out parameters to make a method return more than one value

use by built-in value types' TryParse( ) method, Pass by reference using the ref modifier

Oven class, A CoffeeMaker is also an Appliance

OverFlowException, All exception objects inherit from Exception

overloaded constructors, The Swindler launches another diabolical plan, You can help Brian out by building a program to manage his excuses, Create a user control to animate a picture, Create a user control to animate a picture

excuse management program, You can help Brian out by building a program to manage his excuses

taking a Stream, The Swindler launches another diabolical plan

overloaded methods, Enums let you represent numbers with names, You can build your own overloaded methods

building your own, You can build your own overloaded methods

overriding methods, A subclass can override methods to change or replace methods it inherited, Any place where you can use a base class, you can use one of its subclasses instead, A subclass can hide methods in the superclass, Use the override and virtual keywords to inherit behavior, Use the override and virtual keywords to inherit behavior, An abstract method doesn't have a body

abstract class methods, An abstract method doesn't have a body

hiding versus, A subclass can hide methods in the superclass

override keyword, A subclass can override methods to change or replace methods it inherited, Any place where you can use a base class, you can use one of its subclasses instead, Use the override and virtual keywords to inherit behavior

using to inherit behavior, Use the override and virtual keywords to inherit behavior

P

page-based navigation, Windows Store apps, Jimmy could use some help

page header text, changing, Use properties to change how the controls look

Page object, Windows Store apps use XAML to create UI objects, Use data binding to build Sloppy Joe a better menu, WPF applications use XAML to create UI objects, Use data binding to build Sloppy Joe a better menu

creating instance of MenuMaker and using it for data context, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

page root event handlers for swipes and taps, Respond to swipe and keyboard input

pages, XAML controls can contain text...and more, Use a data template to display objects, Finish the stopwatch app, Use a data template to display objects

choices for design and creation of, Use a data template to display objects, Use a data template to display objects

in MVVM applications, Finish the stopwatch app

laying out, using Graid versus StackPanel, XAML controls can contain text...and more

parameterless constructors, Use static resources to declare your objects in XAML, Modify MenuMaker to notify you when the GeneratedDate property changes, Create a user control to animate a picture, Use static resources to declare your objects in XAML, Modify MenuMaker to notify you when the GeneratedDate property changes, Create a user control to animate a picture

parameters, Let's take a closer look at your code, Mike's Navigator class has methods to set and modify routes, So what did you just build?, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields

- capitalization, Use a constructor to initialize private fields
- masking fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields
- method, Let's take a closer look at your code

parent, How would you design a zoo simulator?

partial classes, Two classes can be in the same namespace, Use logical operators to check conditions

PascalCase, Use a constructor to initialize private fields

patterns., MessageDialog uses the callback pattern (see design patterns)

PictureBox controls, When you change things in the IDE, you're also changing your code, You'll need three classes and a form, Here's your application architecture, Here's what your GUI should look like, How would you design a zoo simulator?, Everything in the dungeon is a PictureBox, The form's UpdateCharacters() method moves the PictureBoxes into position

- labs, #1 A Day at the Races, You'll need three classes and a form, Here's your application architecture, Here's what your GUI should look like
- updating, The form's UpdateCharacters() method moves the PictureBoxes into position

pictures, user control to animate, Create a user control to animate a picture, Create a user control to animate a picture

pinch/zoom, Use semantic zoom to navigate your data

pixels, Page layout starts with controls, Use the grid system to lay out app pages, Use the grid system to lay out app pages

- in grid layout margins, Page layout starts with controls, Use the grid system to lay out app pages
- use of term in XAML layout, Use the grid system to lay out app pages

PointerOver state (controls), Visual states make controls respond to changes

PointerPressed event handler, Create an app to explore routed events, Create an app to explore routed events

polymorphism, An abstract method doesn't have a body, Polymorphism means that one object can take many different forms

as principle of OOP, An abstract method doesn't have a body

popping up dialog boxes, Use built-in objects to pop up standard dialog boxes

Pressed state (controls), Visual states make controls respond to changes, Use object animations to animate object values

animation of, Use object animations to animate object values

private access modifier, Get a little practice using interfaces, There's more than just public and private

private fields, Encapsulation means keeping some of the data in a class private, Use a constructor to initialize private fields, Use a constructor to initialize private fields

declaring, Use a constructor to initialize private fields

initializing with constructors, Use a constructor to initialize private fields

private methods, But is the RealName field REALLY protected?, Use a constructor to initialize private fields

calculating intermediate costs in Dinner Party calculator, Use a constructor to initialize private fields

Program class, Rebuild your app for Windows Desktop, Your desktop app knows where to start

code for desktop app stored in, Your desktop app knows where to start

Main( ) method, Rebuild your app for Windows Desktop

Programmer Reference for C#, #3. Namespaces and assemblies

programming, C# and the Visual Studio IDE make lots of things easy, What Visual Studio does for you..., What you'll do next, Use the IDE to create your own method

benefits of using C# with Visual Studio IDE, C# and the Visual Studio IDE make lots of things easy

C# code, syntax for, What you'll do next, Use the IDE to create your own method

code automatically generated by Visual Studio IDE, What Visual Studio does for you...

programs, Fill in the code for your method, Fill in the code for your method, Finish the method and run your program, Finish the method and run your program, Where programs come from, The IDE helps you code, Anatomy of a program, Your programs use variables to work with data, A few useful types, Use the debugger to see your variables change, Use the debugger to see your variables change, Loops perform an action over and over

anatomy of C# program, Anatomy of a program

breaking, restarting, and stopping in IDE, Fill in the code for your method, Finish the method and run your program

IDE helping you code, The IDE helps you code

loops in, Loops perform an action over and over

operators in, A few useful types

origins of C# programs, Where programs come from

running, Fill in the code for your method, Finish the method and run your program

using debugger to, Use the debugger to see your variables change

using debugger to, Use the debugger to see your variables change (see variables change)

variables in, Your programs use variables to work with data

ProgressBar, Set up the grid for your page, Add controls to your grid, Use properties to change how the controls look, Controls make the game work

adding to Windows Store app, Set up the grid for your page, Add controls to your grid

updating for Windows Store app game, Use properties to change how the controls look, Controls make the game work

projects, When you're doing this..., Where programs come from

creating Windows Store project, When you're doing this...

project files (.csproj), Where programs come from

properties, Set up the grid for your page, Add controls to your grid, Some methods have a return value, An instance uses fields to keep track of things, Properties make encapsulation easier, Use automatic properties to finish the class, Use automatic properties to finish the class, What if we want to change the feed multiplier?, What if we want to change the feed multiplier?, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Use a constructor to initialize private fields, An interface tells a class that it must implement certain methods and properties, You can find out if a class implements a certain interface with "is", Access modifiers change visibility, Keep your eyes open for polymorphism in the next exercise!

automatic, Use automatic properties to finish the class, Keep your eyes open for polymorphism in the next exercise!

using backing fields instead of, Keep your eyes open for polymorphism in the next exercise!

class, Some methods have a return value

get and set accessors as, Use a constructor to initialize private fields

in interfaces, You can find out if a class implements a certain interface with “is”

initializing public properties, What if we want to change the feed multiplier?

interface requiring class to implement, An interface tells a class that it must implement certain methods and properties

making encapsulation easier, Properties make encapsulation easier

public, capitalization, Use a constructor to initialize private fields

read-only, Use automatic properties to finish the class, What if we want to change the feed multiplier?

statements in, Use a constructor to initialize private fields

using to fix Dinner Party calculator (example), Use a constructor to initialize private fields

versus fields, Access modifiers change visibility

XAML controls, Set up the grid for your page, Add controls to your grid

Properties window, Add controls to your grid, Use properties to change how the controls look, Controls make the game work, Controls make the game work, Make the Start button work, Make the Start button work, Add code to make your controls interact with the player, Add code to make your controls interact with the player, Dragging humans onto enemies ends the game, Dragging humans onto enemies ends the game, Make your enemies look like aliens, Make your enemies look like aliens, Rebuild your app for Windows Desktop

event handlers, Make the Start button work, Add code to make your controls interact with the player

Search box using to find XAML properties, Controls make the game work

Search box, using to find XAML properties, Controls make the game work

switching between event handlers and properties in, Make the Start button work, Add code to make your controls interact with the player

switching between events and properties in, Dragging humans onto enemies ends the game, Make your enemies look like aliens

transforms, Dragging humans onto enemies ends the game, Make your enemies look like aliens

using to change controls in Windows Store apps, Add controls to your grid, Use properties to change how the controls look

using to set up Windows Desktop app controls, Rebuild your app for Windows Desktop

PropertyChanged event, INotifyPropertyChanged lets bound objects send updates, Modify MenuMaker to notify you when the GeneratedDate property changes, INotifyPropertyChanged lets bound objects send updates, Modify MenuMaker to notify you when the GeneratedDate property changes

raising, Modify MenuMaker to notify you when the GeneratedDate property changes, Modify MenuMaker to notify you when the GeneratedDate property changes

protected access modifier, There's more than just public and private

protected keyword, Access modifiers change visibility

public access modifier, Two classes can be in the same namespace, There's more than just public and private, ...so what did I just do?

classes, Two classes can be in the same namespace

public fields, A few ideas for encapsulating classes, What if we want to change the feed multiplier?

initializing, What if we want to change the feed multiplier?

public interfaces, Use the interface keyword to define an interface

public methods, Private fields and methods can only be accessed from inside the class, A few ideas for encapsulating classes, Use a constructor to initialize private fields

accessing private fields, Private fields and methods can only be accessed from inside the class

capitalization, Use a constructor to initialize private fields

public properties, What if we want to change the feed multiplier?, Use a constructor to initialize private fields

capitalization, Use a constructor to initialize private fields

initializing, What if we want to change the feed multiplier?

public void method, Get a little practice using interfaces

Publisher-Subscriber pattern, MessageDialog uses the callback pattern

publishing apps to Windows Store, Publish your app

Q

queries, LINQ makes queries easy, LINQ is simple, but your queries don't have to be, Start building Jimmy an app, LINQ is versatile, Use join to combine two collections into one sequence, Add semantic zoom to Jimmy's app

- anatomy of, LINQ is simple, but your queries don't have to be
- editing with LINQPad, Add semantic zoom to Jimmy's app
- LINQ, LINQ makes queries easy, LINQ is versatile, Use join to combine two collections into one sequence
  - using join to combine two collections into one query, Use join to combine two collections into one sequence
- query manager class, Start building Jimmy an app

query detail page, Use the new keyword to create anonymous types

queues, And yet MORE collection types..., A queue is FIFO — First In, First Out, A queue is FIFO — First In, First Out, A stack is LIFO — Last In, First Out, A stack is LIFO — Last In, First Out

- converting to lists, A stack is LIFO — Last In, First Out
- enqueuing and dequeuing, A queue is FIFO — First In, First Out
- FIFO (First In, First Out), A queue is FIFO — First In, First Out
- foreach loop, A stack is LIFO — Last In, First Out

R

\r (return character), Build a program that uses a dictionary, Things you can do with Directory:

Racetrack Simulator project., Here's what your GUI should look like (see labs, #1 A Day at the Races)

Random class, Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!, Enums let you represent numbers with names

- Next( ) method, Enums let you represent numbers with names

randomizing results, Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!

random numbers, generation of, Private fields and methods can only be accessed from inside the class

readonly keyword, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

read-only properties, Use automatic properties to finish the class, What if we want to change the feed multiplier?, Use a constructor to initialize private fields

Cost property, Dinner Party calculator, Use a constructor to initialize private fields

ReadTextAsync( ) method, Use the FileIO class to read and write files

receivers of events, connecting with senders, Connecting event senders with event listeners

Rectangle controls, Controls make the game work, Controls make the game work, Controls make the game work, Controls make the game work

adding to Canvas, Controls make the game work, Controls make the game work

turning into diamond by rotating, Controls make the game work, Controls make the game work

rectangles, The form brings it all together

using for Game boundaries, The form brings it all together

refactoring, #8. Refactoring

references., Refer to your objects with reference variables, You can't instantiate an interface, but you can reference an interface, Interface references work just like object references, Access modifiers change visibility, Values get copied; references get assigned, Pass by reference using the ref modifier

(see also reference variables)

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

object, Interface references work just like object references

object versus interface, Access modifiers change visibility

passing by reference using ref modifier, Pass by reference using the ref modifier

versus values, Values get copied; references get assigned

reference variables, Refer to your objects with reference variables, References are like labels for your object, If there aren't any more references, your object gets garbage-collected, Multiple references and their side effects, Two references means TWO ways to change an object's data, Arrays can contain a bunch of reference variables, too, Objects use references to talk to each other, Where no object has gone before, Controls are objects, just like any other object, Any place where you can use a base class, you can use one of its subclasses instead, Any place where you can use a base class, you can use one of its subclasses instead, Upcasting and downcasting work with interfaces, too, Polymorphism means that one object can take many different forms, Polymorphism means that one object can take many different forms, Connecting event senders with event listeners

arrays of, Arrays can contain a bunch of reference variables, too

assigning to instance of subclasses, Any place where you can use a base class, you can use one of its subclasses instead

for controls, Controls are objects, just like any other object

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

how they work, Where no object has gone before

interface type, pointing to object implementing interface, Polymorphism means that one object can take many different forms

multiple references to single object, References are like labels for your object, Multiple references and their side effects, Two references means TWO ways to change an object's data, Upcasting and downcasting work with interfaces, too

accessing different methods and properties, Upcasting and downcasting work with interfaces, too

side effects of, Multiple references and their side effects

unintentional changes, Two references means TWO ways to change an object's data

objects talking to other objects, Objects use references to talk to each other

setting equal to instance of different class, Polymorphism means that one object can take many different forms

substituting subclass reference in place of base class, Any place where you can use a base class, you can use one of its subclasses instead

ref keyword, Pass by reference using the ref modifier

reimbursement calculator for mileage, When you call a method, the arguments must be compatible with the types of the parameters

Remote Debugger, using to sideload your app, Use the Remote Debugger to sideload your app

remote debugging, starting, Start remote debugging

reserved words, When you call a method, the arguments must be compatible with the types of the parameters, Where no object has gone before, Controls are objects, just like any other object

return statement, Let's take a closer look at your code, Mike's Navigator class has methods to set and modify routes, Some methods have a return value

return type, Mike's Navigator class has methods to set and modify routes, Some methods have a return value

return value, Let's take a closer look at your code, Mike's Navigator class has methods to set and modify routes

risky code, _, What happens when a method you want to call is risky?

RoboBee class, The RoboBee 4000 can do a worker bee's job without using valuable honey

robust, Uh oh — the code's still got problems..., What happens when a method you want to call is risky?, If you have code that ALWAYS should run, use a finally block, Nullable types help you make your programs more robust, Uh-oh — the code's still got problems..., What happens when a method you want to call is risky?

rotations, Controls make the game work, Controls make the game work

RoutedEventArgs object, XAML controls use routed events, XAML controls use routed events

routed events, XAML controls use routed events, Create an app to explore routed events, User controls let you create your own controls, XAML controls use routed events, Create an app to explore routed events

  creating Windows Store app to explore, Create an app to explore routed events, Create an app to explore routed events

  user controls, User controls let you create your own controls

rows and columns, resizing in Windows Store app page, Rows and columns can resize to match the page size, Use the grid system to lay out app pages, Use the grid system to lay out app pages, Rows and columns can resize to match the page size

  Auto setting, Use the grid system to lay out app pages

  using * for row height or column width, Use the grid system to lay out app pages

RSS feed, LINQ to XML, Save and load XML files

S

SaveFileDialog control, Dialog boxes are objects, too, Use file dialogs to open and save files (all with just a few lines of code), Use file dialogs to open and save files (all with just a few lines of code)

  Title property, Use file dialogs to open and save files (all with just a few lines of code)

sbyte type, The variable's type determines what kind of data it can store

scope, Access modifiers change visibility

ScrollViewer controls, Page layout starts with controls, XAML controls can contain text...and more, Page layout starts with controls

  nesting only single control in, XAML controls can contain text...and more

  Windows Store Go Fish! app page, Page layout starts with controls, Page layout starts with controls

sealed access modifier, There's more than just public and private

sealed keyword, Extension methods add new behavior to EXISTING classes

select clause, Add the new queries to Jimmy's app, Jimmy saved a bunch of dough

selecting and deselecting controls for editing, Add controls to your grid, Use properties to change

how the controls look

SelectionChanged event handler, Use the new keyword to create anonymous types

select new clause, Use join to combine two collections into one sequence, Jimmy saved a bunch of dough

Selector class, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

semantic zoom control, Use semantic zoom to navigate your data, Use semantic zoom to navigate your data, Add semantic zoom to Jimmy's app

  adding to comic books management app, Add semantic zoom to Jimmy's app

  basic XAML pattern for, Use semantic zoom to navigate your data

senders of events, connecting with receivers, Connecting event senders with event listeners

separation of concerns, Use a constructor to initialize private fields, Now you're ready to finish the job for Kathleen!

sequences, LINQ is versatile, LINQ is versatile

  defined, LINQ is versatile

Serializable attribute, If you want your class to be serializable, mark it with the [Serializable] attribute

serialization, Add an overloaded Deck() constructor that reads a deck of cards in from a file, What happens to an object when it's serialized?, But what exactly IS an object's state? What needs to be saved?, When an object is serialized, all of the objects it refers to get serialized, too..., Serialization lets you read or write a whole object graph all at once, If you want your class to be serializable, mark it with the [Serializable] attribute, Let's serialize and deserialize a deck of cards, Let's serialize and deserialize a deck of cards, You can read and write serialized files manually, too, Find where the files differ, and use that information to alter them, Build a slightly less simple text editor, Finalizers can't depend on stability, Make an object serialize itself in its Dispose()

data contract, Build a slightly less simple text editor

finalizers and, Finalizers can't depend on stability

finding where serialized files differ and altering them, Find where the files differ, and use that information to alter them

making classes serializable, If you want your class to be serializable, mark it with the [Serializable] attribute

making object serialize in Dispose( ) method, Make an object serialize itself in its Dispose()

object states, But what exactly IS an object's state? What needs to be saved?

reading and writing serialized files manually, You can read and write serialized files manually, too

reading entire object, Serialization lets you read or write a whole object graph all at once

serializing and deserializing deck of cards, Let's serialize and deserialize a deck of cards

serializing objects out to file, Let's serialize and deserialize a deck of cards

what happens to objects, What happens to an object when it's serialized?, When an object is serialized, all of the objects it refers to get serialized, too...

SerializationException, Uh oh — the code's still got problems..., Uh oh — the code's still got problems..., Use the debugger to follow the try/catch flow, Uh-oh — the code's still got problems..., Uh-oh — the code's still got problems...

BinaryFormatter, Uh oh — the code's still got problems..., Uh-oh — the code's still got problems...

set accessor, Properties make encapsulation easier, Use a constructor to initialize private fields, Get a little practice using interfaces, You can find out if a class implements a certain interface with "is"

interface properties, You can find out if a class implements a certain interface with "is"

interfaces with get accessor without set accssor, Get a little practice using interfaces

Settings charm, Use delegates to use the Windows settings charm, Use the Settings charm to open a SettingsFlyout

using to open About popup, Use the Settings charm to open a SettingsFlyout

SettingsPane class, Use delegates to use the Windows settings charm

short-circuit operators, ...more basics...

short type, The variable's type determines what kind of data it can store, A variable is like a data to-go cup, 10 pounds of data in a 5-pound bag

ShowDialog( ) method, Use built-in objects to pop up standard dialog boxes, Dialog boxes are objects, too

signature (method), Use a constructor to initialize private fields

similar behaviors, When your classes use inheritance, you only need to write your code once

similar code, When your classes use inheritance, you only need to write your code once

simulator in Visual Studio, running Windows Store apps, Build Brian a new Excuse Manager app

skew transforms, Dragging humans onto enemies ends the game, Make your enemies look like aliens

Sloppy Joe's Random Menu Item project, Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!, Use data binding to build Sloppy Joe a better menu

building better menu with data binding, Use data binding to build Sloppy Joe a better menu

Solution Explorer, What Visual Studio does for you..., The IDE helps you code, The IDE helps you code, When you change things in the IDE, you're also changing your code

Form designer, When you change things in the IDE, you're also changing your code

showing everything in a project, The IDE helps you code

switching between open project files, The IDE helps you code

solutions (.sln files), created by IDE, Where programs come from

SortBy field, IComparer can do complex comparisons

Sort( ) method, Lists are easy, but SORTING can be tricky

Source property, getting or setting with two way binding, Two-way binding can get or set the source property

Source property, Image control, animating, Create a user control to animate a picture, Create a user control to animate a picture

spec (specification), The Spec: Build a Racetrack Simulator, Build a program that uses a dictionary

building a racetrack simulator, The Spec: Build a Racetrack Simulator

splash screen, adding to a program, Add a splash screen and a tile

Split App template, creating project with, The IDE's Split App template helps you build apps for navigating data, The IDE's Split App template helps you build apps for navigating data, The IDE's Split App template helps you build apps for navigating data, The IDE's Split App template helps you build apps for navigating data, The IDE's Split App template helps you build apps for navigating data

adding images files to Assets folder, The IDE's Split App template helps you build apps for navigating data

modifying code-behind in ItemsPage.xaml.cs, The IDE's Split App template helps you build apps for navigating data

modifying code-behind in SplitPage.xaml.cs, The IDE's Split App template helps you build apps for navigating data

modifying SplitPage.xaml to show comic book details, The IDE's Split App template helps you build apps for navigating data

sprites, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind

controls to a Canvas

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

stack, And yet MORE collection types..., A stack is LIFO — Last In, First Out, A stack is LIFO — Last In, First Out, A stack is LIFO — Last In, First Out, A stack is LIFO — Last In, First Out, The stack vs. the heap: more on memory, Nullable types help you make your programs more robust

converting to lists, A stack is LIFO — Last In, First Out

foreach loop, A stack is LIFO — Last In, First Out

LIFO (Last In, First Out), A stack is LIFO — Last In, First Out

popping items off, A stack is LIFO — Last In, First Out

versus heap, The stack vs. the heap: more on memory

StackPanel, Here's what you're going to build, Start with a blank application, Use properties to change how the controls look, Controls make the game work, Controls make the game work, Add code to make your controls interact with the player, Redesign the Go Fish! form as a Windows Store app page, Page layout starts with controls, Rows and columns can resize to match the page size, Use the grid system to lay out app pages, XAML controls can contain text...and more, XAML controls can contain text...and more, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu, Use static resources to declare your objects in XAML, Use a data template to display objects, Create the main page for the Excuse Manager, Rows and columns can resize to match the page size, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu, Use static resources to declare your objects in XAML, Use a data template to display objects

adding human to Windows Store app game, Controls make the game work

DataContext property, setting, Use data binding to build Sloppy Joe a better menu, Use static resources to declare your objects in XAML, Use data binding to build Sloppy Joe a better menu, Use static resources to declare your objects in XAML

Excuse Manager project, Create the main page for the Excuse Manager

Grid layout versus, XAML controls can contain text...and more

using Document Outline to modify, Controls make the game work

using to group TextBlock and ContentControl, Use properties to change how the controls look

StackPanel controls adding human to Windows Store app game, Controls make the game work

using Document Outline to modify, Use properties to change how the controls look

using to group TextBlock and ContentControl, Add controls to your grid

StackTrace property, Exception class, All exception objects inherit from Exception

stars and bees, program that animates, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

Start button making it start the program, Add timers to manage the gameplay

Start button, making it start the program, Make the Start button work

Start Debugging button, Use the debugger to see your variables change

start tags, What Visual Studio does for you...

state, User controls let you create your own controls, MVVM means thinking about the state of the app, Events alert the rest of the app to state changes, Finish the stopwatch app

  changes in, stopwatch app, Events alert the rest of the app to state changes

  code related to timing, Finish the stopwatch app

  thinking about, in MVVM, MVVM means thinking about the state of the app

statements, Let's take a closer look at your code, Loops perform an action over and over, Make each button do something, Use logical operators to check conditions, Use logical operators to check conditions

  defined, Let's take a closer look at your code

  ending with ; (semicolon), Make each button do something

  important points about, Use logical operators to check conditions

  in loops, Loops perform an action over and over

static keyword, A little advice for the code exercises, Let's create some instances!

  instance creation and, Let's create some instances!

static methods, A little advice for the code exercises, A little advice for the code exercises

  when to use, A little advice for the code exercises

static resources in XAML, Use static resources to declare your objects in XAML, Use a data template to display objects, Modify MenuMaker to notify you when the GeneratedDate property changes, Use static resources to declare your objects in XAML, Use a data template to display objects, Modify MenuMaker to notify you when the GeneratedDate property changes

Step Over (Debug), Use the debugger to see your variables change

stopwatch app, The ref needs a stopwatch, Events alert the rest of the app to state changes, Build the view for a simple stopwatch, Add the stopwatch ViewModel, Finish the stopwatch app, Converters automatically convert values for binding, Build an analog stopwatch using the same ViewModel, Build the view for a simple stopwatch, Finish the stopwatch app, Converters automatically convert values for binding, Build an analog stopwatch using the same ViewModel

building analog stopwatch using same ViewModel, Build an analog stopwatch using the same ViewModel, Build an analog stopwatch using the same ViewModel

converters converting values for binding, Converters automatically convert values for binding, Converters automatically convert values for binding

events in Model alerting app to state changes, Events alert the rest of the app to state changes

finishing touches, Finish the stopwatch app, Finish the stopwatch app

View, Build the view for a simple stopwatch, Build the view for a simple stopwatch

ViewModel, Add the stopwatch ViewModel

Storyboard object, Create a user control to animate a picture, Create a user control to animate a picture, Make your bees fly around a page, Create a user control to animate a picture, Create a user control to animate a picture, Make your bees fly around a page

Begin( ) method, Create a user control to animate a picture, Create a user control to animate a picture

garbage collection for, Make your bees fly around a page, Make your bees fly around a page

SetTarget( ) and SetTargetProperty( ) methods, Create a user control to animate a picture, Create a user control to animate a picture

Storyboard tags, Visual states make controls respond to changes, Use object animations to animate object values

Pressed storyboard, adding animation to, Use object animations to animate object values

Stream object, .NET uses streams to read and write data, Use Stream.Read() to read bytes from a stream

Read( ) method, Use Stream.Read() to read bytes from a stream

StreamReader, The Swindler launches another diabolical plan, Things you can do with Directory:, StreamReader and StreamWriter will do just fine (for now), Use Stream.Read() to read bytes from a stream

hex dump, StreamReader and StreamWriter will do just fine (for now)

versus FileStreams, Use Stream.Read() to read bytes from a stream

streams, .NET uses streams to read and write data, Different streams read and write different things, Different streams read and write different things, A FileStream reads and writes bytes to a file, Write text to a file in three simple steps, Data can go through more than one stream, Things you can do with Directory:, Avoid filesystem errors with using statements, Avoid filesystem errors with using statements, Let’s serialize and deserialize a deck of cards, Use file streams to build a hex dumper, Use Stream.Read() to read bytes from a stream

chaining, Data can go through more than one stream

closing, Things you can do with Directory:

different types, Different streams read and write different things

Dispose( ) method, Avoid filesystem errors with using statements

forgetting to close, A FileStream reads and writes bytes to a file

reading bytes from, using Stream.Read( ), Use Stream.Read() to read bytes from a stream

serializing objects to, Let's serialize and deserialize a deck of cards

things you can do with, Different streams read and write different things

using file streams to build hex dumper, Use file streams to build a hex dumper

using statements, Avoid filesystem errors with using statements

writing text to files, Write text to a file in three simple steps

StreamWriter, Write text to a file in three simple steps, Write text to a file in three simple steps, Write text to a file in three simple steps, The Swindler launches another diabolical plan, The Swindler launches another diabolical plan, Things you can do with Directory:, Things you can do with Directory:, StreamReader and StreamWriter will do just fine (for now), Use Stream.Read() to read bytes from a stream

{0} and {1}, passing variables to strings, Things you can do with Directory:

Close( ) method, Write text to a file in three simple steps

hex dump, StreamReader and StreamWriter will do just fine (for now)

using with StreamReader, The Swindler launches another diabolical plan

versus FileStreams, Use Stream.Read() to read bytes from a stream

Write( ) and WriteLine( ) methods, Write text to a file in three simple steps, The Swindler launches another diabolical plan

StringBuilder class, #3. Namespaces and assemblies

string concatenation operator (+), When you cast a value that's too big, C# will adjust it automatically, #3. Namespaces and assemblies

converting numbers or Booleans to strings, When you cast a value that's too big, C# will adjust it automatically

String.IsNullOrEmpty( ), How you'll build the beehive management system

string literals, Write text to a file in three simple steps, Things you can do with Directory:

String.PadLeft( ) method, ...more basics...

strings, A few useful types, C# does some casting automatically, You're going to build a program for Kathleen, Strings don't always work for storing categories of data, Things you can do with Directory:, Add an overloaded Deck() constructor that reads a deck of cards in from a file, .NET uses

Unicode to store characters and text, StreamReader and StreamWriter will do just fine (for now), StreamReader and StreamWriter will do just fine (for now), Extending a fundamental type: string

concatenating, automatic type conversions with + operator, C# does some casting automatically

concatenation operator (+), A few useful types

converting numbers to, StreamReader and StreamWriter will do just fine (for now)

converting to byte array, Things you can do with Directory:

extension methods, Extending a fundamental type: string

formatting, You're going to build a program for Kathleen

splitting, Add an overloaded Deck() constructor that reads a deck of cards in from a file

storage of data in memory as Unicode, .NET uses Unicode to store characters and text

storing categories of data, Strings don't always work for storing categories of data

Substring( ) method, StreamReader and StreamWriter will do just fine (for now)

string type, You have to assign values to variables before you use them, The variable's type determines what kind of data it can store, A variable is like a data to-go cup, C# does some casting automatically, When you call a method, the arguments must be compatible with the types of the parameters

converting other types to, C# does some casting automatically

structs, A struct looks like an object..., Structs are value types; objects are reference types, Structs are value types; objects are reference types, The stack vs. the heap: more on memory, Nullable types help you make your programs more robust, Nullable types help you make your programs more robust, "Captain" Amazing...not so much

boxed, The stack vs. the heap: more on memory, "Captain" Amazing...not so much

setting one equal to another, Structs are value types; objects are reference types, Nullable types help you make your programs more robust

versus classes, Nullable types help you make your programs more robust

versus objects, Structs are value types; objects are reference types

styles, Styles set properties on multiple controls

altering appearance of a type of control, Styles set properties on multiple controls

subclasses, When your classes use inheritance, you only need to write your code once, How would you design a zoo simulator?, How would you design a zoo simulator?, Create the class hierarchy, Use a colon to inherit from a base class, We know that inheritance adds the base class fields, properties, and methods to the subclass..., We know that inheritance adds the base class fields, properties, and methods to the subclass..., A subclass can override methods to change or replace methods it inherited, Any place where you can use a base class, you can use one of its subclasses instead, Any place where you can use a base class, you can use one of its subclasses instead, Any place where you can use a

base class, you can use one of its subclasses instead, A subclass can hide methods in the superclass, A subclass can access its base class using the base keyword, When a base class has a constructor, your subclass needs one, too, Upcasting works with both objects and interfaces, Polymorphism means that one object can take many different forms

avoiding duplicate code, using inheritance, How would you design a zoo simulator?

child and, How would you design a zoo simulator?

constructors, When a base class has a constructor, your subclass needs one, too

hiding superclass methods, A subclass can hide methods in the superclass

inheriting from base class, Use a colon to inherit from a base class

modifying, We know that inheritance adds the base class fields, properties, and methods to the subclass...

overriding inherited methods, A subclass can override methods to change or replace methods it inherited

passing instance of, Any place where you can use a base class, you can use one of its subclasses instead

upcasting, Upcasting works with both objects and interfaces, Polymorphism means that one object can take many different forms

using instead of base classes, Any place where you can use a base class, you can use one of its subclasses instead

subtraction operator., How would you design a zoo simulator? (see - (minus sign), under Symbols
superclass)

Suspending event, Windows Store apps, Windows Store apps use events for process lifetime management, Add process lifetime management to Jimmy's comics

modifying OnSuspending( ) event handler, Add process lifetime management to Jimmy's comics

SuspensionManager class, Add process lifetime management to Jimmy's comics

swipes and taps, handling in Invaders game, Respond to swipe and keyboard input, Handling user input

switch statements, Use a switch statement to choose the right option, Add an overloaded Deck() constructor that reads a deck of cards in from a file

building new constructors with, Add an overloaded Deck() constructor that reads a deck of cards in from a file

System.ComponentModel namespace, Modify MenuMaker to notify you when the GeneratedDate property changes, Modify MenuMaker to notify you when the GeneratedDate property changes

System.Diagnostics.Debug.WriteLine( ), Windows Forms use an object graph set up by the IDE

System.IO.File class, Use the FileIO class to read and write files

System namespace, Use logical operators to check conditions

System.Runtime.Serialization namespace, A data contract is an abstract definition of your object's data

System.Windows.Form class, Use the IDE to explore the object graph

System.Windows.Forms.Control class, Use the IDE to explore the object graph

System.Windows.Forms namespace, Your desktop app knows where to start, Some methods have a return value, Create a project for your guys, Use a constructor to initialize private fields

T

\t (tab character), The variable's type determines what kind of data it can store, Write text to a file in three simple steps, Things you can do with Directory:

TabControl, We need a BirthdayParty class, Build the Part y Planner version 2.0

TableLayoutPanel, Use file dialogs to open and save files (all with just a few lines of code), Everything in the dungeon is a PictureBox, Windows Forms use an object graph set up by the IDE

Controls property, Windows Forms use an object graph set up by the IDE

TabPages property, Build the Part y Planner version 2.0

tags, XAML, What Visual Studio does for you...

Take statement, Add the new queries to Jimmy's app

taps, page root event handlers for, Respond to swipe and keyboard input

target portal player will drag human into (game example), Controls make the game work, Controls make the game work

Task class (or Task<T>), Use a Task to call one async method from another

templates., Use properties to change how the controls look, Controls make the game work, Controls make the game work, Dragging humans onto enemies ends the game, Make your enemies look like aliens

(see also names of individual templates throughout)

creating enemy template for Windows Store app game, Use properties to change how the controls look, Controls make the game work

editing for enemy aliens (example), Dragging humans onto enemies ends the game, Make your enemies look like aliens

TextBlock controls, Add controls to your grid, Use properties to change how the controls look, Use properties to change how the controls look, Controls make the game work, Controls make the game work, Controls make the game work, if/else statements make decisions, Make each button do something, Make each button do something, Use logical operators to check conditions, Page layout starts with controls, Data binding connects your XAML pages to your classes, Data binding connects your XAML pages to your classes, Use data binding to build Sloppy Joe a better menu, Build an app

from the ground up, Page layout starts with controls, Use data binding to build Sloppy Joe a better menu

binding path, Data binding connects your XAML pages to your classes

changing text and style in Windows Store app, Add controls to your grid, Use properties to change how the controls look

data binding to properties in MenuMaker (example), Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

data context, Data binding connects your XAML pages to your classes

Game Over text for Windows Store app, Controls make the game work, Controls make the game work

Style property, if/else statements make decisions, Build an app from the ground up

updating by pressing buttons, Make each button do something

using Document Outline to modify, Use properties to change how the controls look, Controls make the game work

Windows Store Go Fish! app page, Page layout starts with controls, Page layout starts with controls

TextBox controls, So what did you just build?, Build the Part y Planner version 2.0, Build the Part y Planner version 2.0, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu, Build a slightly less simple text editor, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

Birthday Party project, Build the Part y Planner version 2.0, Build the Part y Planner version 2.0

adding TexChanged event handler, Build the Part y Planner version 2.0

Text property, using to modify text, Build a slightly less simple text editor

two-way data binding, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

TextChanged event handler, Build the Part y Planner version 2.0, Build a slightly less simple text editor, Build a slightly less simple text editor

text editors, Build a slightly less simple text editor

building less simple editor, Build a slightly less simple text editor

Text property, XAML controls, XAML controls can contain text...and more, XAML controls can contain text...and more

this keyword, Objects use references to talk to each other, Initialize your arrays of Greyhound and Guy objects, Use a constructor to initialize private fields, Use a constructor to initialize private fields, Access modifiers change visibility, Extension methods add new behavior to EXISTING classes, Use a standard name when you add a method to raise an event

distinguishing fields from parameters with same name, Use a constructor to initialize private fields, Use a constructor to initialize private fields

in extension method's first parameter, Extension methods add new behavior to EXISTING classes

labs, #1 A Day at the Races, Initialize your arrays of Greyhound and Guy objects

using to raise event, Use a standard name when you add a method to raise an event

this variable, Where no object has gone before

threading, #4. Use BackgroundWorker to make your WinForms responsive

throw, using to rethrow exceptions, Use more than one catch block to handle multiple types of exceptions, Bees need an OutOfHoney exception

tiles, Add a splash screen and a tile

TimeNumberFormatConverter class, Converters automatically convert values for binding, Converters automatically convert values for binding

timers, Here's what you've done so far, Add timers to manage the gameplay, Controls are objects, just like any other object

adding to manage gameplay, Here's what you've done so far, Add timers to manage the gameplay

LabelBouncer animation (example), Controls are objects, just like any other object

Title property, SaveFileDialog, Dialog boxes are objects, too, Use file dialogs to open and save files (all with just a few lines of code)

ToggleSwitch controls, Create an app to explore routed events

Toolbox window, What Visual Studio does for you..., Set up the grid for your page, Set up the grid for your page, Add controls to your grid, Add controls to your grid, Rebuild your app for Windows Desktop

ALL XAML Controls section, Set up the grid for your page, Add controls to your grid

Common XAML Controls section, Set up the grid for your page, Add controls to your grid

expanding, Rebuild your app for Windows Desktop

ToString( ) method, C# does some casting automatically, You're going to build a program for Kathleen, Enums let you represent numbers with names, Overriding a ToString() method lets an object describe itself, Update your foreach loops to let your Ducks and Cards print themselves

adding to Card object (example), Update your foreach loops to let your Ducks and Cards print themselves

overriding and letting object describe itself, Overriding a ToString() method lets an object describe itself

transforms, Controls make the game work, Controls make the game work, Dragging humans onto enemies ends the game, Make your enemies look like aliens, Build an analog stopwatch using the same ViewModel, Build an analog stopwatch using the same ViewModel

hands on analog stopwatch, Build an analog stopwatch using the same ViewModel, Build an analog stopwatch using the same ViewModel

performing in Properties window, Dragging humans onto enemies ends the game, Make your enemies look like aliens

rotating Rectangle 45 degrees, Controls make the game work, Controls make the game work

try blocks, Handle exceptions with try and catch, Handle exceptions with try and catch, What happens when a method you want to call is risky?, Bees need an OutOfHoney exception

try blocks, What happens when a method you want to call is risky?, Use the debugger to follow the try/catch flow, Bees need an OutOfHoney exception, Use the debugger to follow the try/catch flow

(see also exception handling)

following in debugger, Use the debugger to follow the try/catch flow, Use the debugger to follow the try/catch flow

getting with using statements, Bees need an OutOfHoney exception

try/catch/finally sequence for error handling, If you have code that ALWAYS should run, use a finally block

(see also exception handling)

try/finally block, Exception avoidance: implement IDisposable to do your own cleanup

(see also exception handling)

two-way data binding, Two-way binding can get or set the source property, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

TextBox control in MenuMaker project, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

type argument, Generics can store any type

Type class and GetType( ) method, #5. The Type class and GetType()

typeof keyword, Use the IDE to explore app page navigation

types, _, Your programs use variables to work with data, You have to assign values to variables before you use them, Mike's Navigator class has methods to set and modify routes, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, A variable is like a data to-go cup, Even when a number is the right size, you can't just assign it to any variable, C# does some casting automatically, When you call a method, the arguments must be compatible with the types of the parameters, Refer to your objects with reference variables, Multiple references and their side effects, A special case: arrays, Where no object has gone before, Generics can store any type, A delegate STANDS IN for an actual method

arguments, compatibility with types of parameters, When you call a method, the arguments must be compatible with the types of the parameters

arrays, A special case: arrays

automatic casting in C#, C# does some casting automatically

char, The variable's type determines what kind of data it can store

common types in C#, The variable's type determines what kind of data it can store

delegate, A delegate STANDS IN for an actual method

different types holding different-sized values, Where no object has gone before

for whole numbers, The variable's type determines what kind of data it can store

generic, Generics can store any type

int, string, and bool types, You have to assign values to variables before you use them

literals, The variable's type determines what kind of data it can store

multiple references and their side effects, Multiple references and their side effects

object, The variable's type determines what kind of data it can store

referring to objects with reference variables, Refer to your objects with reference variables

return type, Mike's Navigator class has methods to set and modify routes

storing really huge and really tiny numbers, The variable's type determines what kind of data it can store

value types, Even when a number is the right size, you can't just assign it to any variable

variable, Your programs use variables to work with data, A variable is like a data to-go cup

typing game, building, Build a typing game

U

UICommandInvokedHandler, Use delegates to use the Windows settings charm

UICommand object, Windows Store apps use await to be more responsive

UIElement base class, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

uint type, The variable's type determines what kind of data it can store

UI (user interface), C# and the Visual Studio IDE make lots of things easy, UI controls can be instantiated with C# code, too, UI controls can be instantiated with C# code, too

creating controls with C# code, UI controls can be instantiated with C# code, too, UI controls can be instantiated with C# code, too

creating using Visual Designer, C# and the Visual Studio IDE make lots of things easy

ulong type, The variable's type determines what kind of data it can store

Undo command undoing changes to controls, Add controls to your grid

Undo command (IDE), Use properties to change how the controls look, Use logical operators to check conditions

undoing changes to controls, Use properties to change how the controls look

unexpected input, What happens when a method you want to call is risky?, What happens when a method you want to call is risky?

unhandled exceptions, Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager, If you have code that ALWAYS should run, use a finally block

versus exceptions, If you have code that ALWAYS should run, use a finally block

Unicode, Build a program that uses a dictionary, Let's serialize and deserialize a deck of cards, .NET uses Unicode to store characters and text, Use Stream.Read() to read bytes from a stream, XAML controls can contain text...and more

converting text to, .NET uses Unicode to store characters and text

units, device-independent, Use the grid system to lay out app pages

upcasting, Upcasting works with both objects and interfaces, Upcasting and downcasting work with interfaces, too, Upcasting and downcasting work with interfaces, too, Polymorphism means that one object can take many different forms, You can upcast an entire list using IEnumerable

but not downcasting, Upcasting and downcasting work with interfaces, too

entire list, using IEnumerable<T>, You can upcast an entire list using IEnumerable

interfaces, Upcasting and downcasting work with interfaces, too

using subclass instead of base class, Polymorphism means that one object can take many different forms

Up Close, access modifiers, Access modifiers change visibility

user controls, User controls let you create your own controls, User controls let you create your own controls, User controls let you create your own controls, Build the view for a simple stopwatch, Finish the stopwatch app, Build an analog stopwatch using the same ViewModel, Create a user control to animate a picture, Build the view for a simple stopwatch, Finish the stopwatch app, Build an analog stopwatch using the same ViewModel, Create a user control to animate a picture

AnalogStopwatch, Build an analog stopwatch using the same ViewModel, Build an analog stopwatch using the same ViewModel

AnimatedImage, Create a user control to animate a picture, Create a user control to animate a picture

objects extending UserControl base class, User controls let you create your own controls

stopwatch app, Build the view for a simple stopwatch, Finish the stopwatch app, Build the view for a simple stopwatch, Finish the stopwatch app

event handlers for, Finish the stopwatch app, Finish the stopwatch app

ushort type, The variable's type determines what kind of data it can store

using statements, Use the IDE to create your own method, Finish the method and run your program, Anatomy of a program, Rebuild your app for Windows Desktop, Your desktop app knows where to start, You can change your program's entry point, Avoid filesystem errors with using statements, Bees need an OutOfHoney exception, Exception avoidance: implement IDisposable to do your own cleanup, Dispose() works with using; finalizers work with garbage collection

Dispose( ), Dispose() works with using; finalizers work with garbage collection

exception handling, Bees need an OutOfHoney exception

for desktop apps, Rebuild your app for Windows Desktop, Your desktop app knows where to start, You can change your program's entry point

using System.Windows.Forms, Your desktop app knows where to start

in C# programs, Anatomy of a program

V

value converters, Converters automatically convert values for binding, Converters automatically convert values for binding, Converters can work with many different types, Converters automatically convert values for binding, Converters automatically convert values for binding, Converters can work with many different types

automatically converting values for binding, Converters automatically convert values for binding, Converters automatically convert values for binding

working with many different types, Converters can work with many different types, Converters can work with many different types

value parameter, set accessors, Use a constructor to initialize private fields

values versus references, Values get copied; references get assigned

value types, You have to assign values to variables before you use them, You have to assign values to variables before you use them, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type

determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, The variable's type determines what kind of data it can store, Even when a number is the right size, you can't just assign it to any variable, Even when a number is the right size, you can't just assign it to any variable, When you call a method, the arguments must be compatible with the types of the parameters, Where no object has gone before, Where no object has gone before, Values get copied; references get assigned, Structs are value types; objects are reference types, Pass by reference using the ref modifier

bool, You have to assign values to variables before you use them (see bool type)

byte, The variable's type determines what kind of data it can store (see byte type)

casting, Even when a number is the right size, you can't just assign it to any variable

changing, Where no object has gone before

char, The variable's type determines what kind of data it can store (see char type)

decimal, The variable's type determines what kind of data it can store (see decimal type)

double, The variable's type determines what kind of data it can store (see double type)

int, You have to assign values to variables before you use them (see int type)

long, The variable's type determines what kind of data it can store (see long type)

more information on, Even when a number is the right size, you can't just assign it to any variable

sbyte, The variable's type determines what kind of data it can store

short, The variable's type determines what kind of data it can store (see short type)

structs as, Structs are value types; objects are reference types

TryParse( ) method using out parameters, Pass by reference using the ref modifier

uint, The variable's type determines what kind of data it can store

ulong, The variable's type determines what kind of data it can store

ushort, The variable's type determines what kind of data it can store

variables matching types of parameters, When you call a method, the arguments must be compatible with the types of the parameters

versus objects, Values get copied; references get assigned

variables, Your programs use variables to work with data, Your programs use variables to work with data, Your programs use variables to work with data, You have to assign values to variables before you use them, Use the debugger to see your variables change, Use the debugger to see your variables change, Make each button do something, Build a form to interact with the guys, Build a form to interact with the guys, The variable's type determines what kind of data it can store, A variable is like a data to-go cup, Even when a number is the right size, you can't just assign it to any variable, When you call a method, the arguments must be compatible with the types of the parameters, Combining =

with an operator, Objects use variables, too, Refer to your objects with reference variables, Rename a variable

adding to form, Build a form to interact with the guys, Build a form to interact with the guys

assigning values to, You have to assign values to variables before you use them, Even when a number is the right size, you can't just assign it to any variable

data type and, Even when a number is the right size, you can't just assign it to any variable

data types, The variable's type determines what kind of data it can store

declarations with name and type, Make each button do something

declaring, Your programs use variables to work with data

matching types of parameters, When you call a method, the arguments must be compatible with the types of the parameters

naming, Combining = with an operator

objects as, Objects use variables, too

reference, Refer to your objects with reference variables (see reference variables)

renaming, Rename a variable

using debugger to, Use the debugger to see your variables change

using debugger to, Use the debugger to see your variables change (see changes in)

values of, Your programs use variables to work with data

var keyword, LINQ makes queries easy, Jimmy saved a bunch of dough

VerticalAlignment property, controls, Add controls to your grid, Use properties to change how the controls look

vertical bars, You can help Brian out by building a program to manage his excuses

View, MVVM lets you design for binding and data, Build the view for a simple stopwatch, Finish the stopwatch app, Finish the stopwatch app, Build the view for a simple stopwatch, Finish the stopwatch app

(see also MVVM pattern)

building for simple stopwatch, Build the view for a simple stopwatch, Build the view for a simple stopwatch

rules for MVVM apps, Finish the stopwatch app

stopwatch app, buttons calling methods in ViewModel, Finish the stopwatch app, Finish the stopwatch app

ViewModel, MVVM lets you design for binding and data, User controls let you create your own controls, User controls let you create your own controls, Add the stopwatch ViewModel, Finish the stopwatch app, User controls let you create your own controls

(see also MVVM pattern)

BasketballRoster project, User controls let you create your own controls, User controls let you create your own controls

rules for MVVM apps, Finish the stopwatch app

stopwatch app, Add the stopwatch ViewModel

using Model statement at top of classes, User controls let you create your own controls

virtual keyword, A subclass can override methods to change or replace methods it inherited, Any place where you can use a base class, you can use one of its subclasses instead, Use the override and virtual keywords to inherit behavior

using to inherit behavior, Use the override and virtual keywords to inherit behavior

virtual machines, Where no object has gone before

virtual methods, Any place where you can use a base class, you can use one of its subclasses instead

Visibility enum, Converters can work with many different types, Converters can work with many different types

Visible property, forms or controls, Wait, wait! What did that say?

Visual Designer, C# and the Visual Studio IDE make lots of things easy, What Visual Studio does for you...

editing user interface, What Visual Studio does for you...

visual states, making controls respond to changes, Visual states make controls respond to changes

Visual Studio 2008 Express, What version of Windows are you using?

setting up, What version of Windows are you using?

Visual Studio IDE, Why you should learn C#, C# and the Visual Studio IDE make lots of things easy, What you do in Visual Studio..., What you do in Visual Studio..., What Visual Studio does for you..., What Visual Studio does for you..., Fill in the code for your method, Finish the method and run your program, Use the Remote Debugger to sideload your app, The IDE helps you code, Let's take a closer look at your code, Use logical operators to check conditions, Use logical operators to check conditions, Rebuild your app for Windows Desktop, When you change things in the IDE, you're also changing your code, Windows Store apps use XAML to create UI objects, Take your Guy Serializer for a test drive, Build Brian a new Excuse Manager app, Rename a variable, Take your Guy Serializer for a test drive

code automatically generated by, handling of, Use logical operators to check conditions

creating new project, What you do in Visual Studio...

different editions, look of, What you do in Visual Studio...

editions and versions of, What Visual Studio does for you...

exploring different parts of, What Visual Studio does for you...

Extract Method feature, Refactor menu, Rename a variable

helping you code, The IDE helps you code, Let's take a closer look at your code

making changes in, changes to code, When you change things in the IDE, you're also changing your code

Remote Debugger, Use the Remote Debugger to sideload your app

Reset Window Layout, from Window menu, Fill in the code for your method, Finish the method and run your program

running Windows Store apps in simulator, Build Brian a new Excuse Manager app

Undo command, and automatically generated code, Use logical operators to check conditions

using with C#, capabilities of, C# and the Visual Studio IDE make lots of things easy

Visual Studio 2012 for Windows Desktop, Rebuild your app for Windows Desktop

Watch windows in Visual Studio 2012 for Windows 8, Windows Store apps use XAML to create UI objects

XAML designer, giving message to rebuild code, Take your Guy Serializer for a test drive, Take your Guy Serializer for a test drive

void keyword, preceding methods, Let's take a closer look at your code

void method, Get a little practice using interfaces, Get a little practice using interfaces

interfaces, Get a little practice using interfaces

public, Get a little practice using interfaces

void return type, Mike's Navigator class has methods to set and modify routes, Some methods have a return value, You can use class and method names to make your code intuitive, Build a form to interact with the guys

W

Watch window, Use the debugger to see your variables change

where clause, LINQ is simple, but your queries don't have to be, Add the new queries to Jimmy's app

while loops, Loops perform an action over and over, Use logical operators to check conditions, Wait, wait! What did that say?, Wait, wait! What did that say?, ...more basics...

continue and break keywords in, ...more basics...

infinite loop, Wait, wait! What did that say?

whitespace, extra, in C# code, Make each button do something

Windows 8, Here's what you're going to build

Windows 8 Camp Training Kit, #1. There's so much more to Windows Store

Windows App Certification Kit, Publish your app

Windows calculator, The variable's type determines what kind of data it can store

Windows Desktop, Windows Desktop apps are easy to build, Rebuild your app for Windows Desktop, Your desktop app knows where to start, You can change your program's entry point, You can change your program's entry point, When you change things in the IDE, you're also changing your code, Wait, wait! What did that say?

building an app, Windows Desktop apps are easy to build, Rebuild your app for Windows Desktop, Your desktop app knows where to start, You can change your program's entry point, You can change your program's entry point, When you change things in the IDE, you're also changing your code, Wait, wait! What did that say?

animations, Wait, wait! What did that say?

changes made in IDE and code changes, When you change things in the IDE, you're also changing your code

changing program's entry point, You can change your program's entry point

entry point, Main( ) method, Rebuild your app for Windows Desktop

MessageBox.Show( ) method, You can change your program's entry point

nuts and bolts of desktop apps, Your desktop app knows where to start

Windows Forms Application project, creating, Rebuild your app for Windows Desktop

Windows Presentation Foundation., Here's what you're going to build (see WPF)

Windows Runtime, namespaces for tools in, The .NET Framework gives you the right tools for the job

Windows Settings charm, using, Use delegates to use the Windows settings charm

Windows.Storage.IStorageFolder, Use async methods to find and open files

Windows.Storage namespace, Use the FileIO class to read and write files, KnownFolders helps you access high-profile folders

KnownFolders class, KnownFolders helps you access high-profile folders

Windows Store apps, _, Here's what you're going to build, Publish your app, When you're doing this..., The .NET Framework gives you the right tools for the job, Windows Store apps use XAML to create UI objects, Redesign the Go Fish! form as a Windows Store app page, Page layout starts with

controls, Rows and columns can resize to match the page size, Use the grid system to lay out app pages, Use the grid system to lay out app pages, Use the grid system to lay out app pages, Data binding connects your XAML pages to your classes, Use data binding to build Sloppy Joe a better menu, Use static resources to declare your objects in XAML, Use a data template to display objects, Use a data template to display objects, INotifyPropertyChanged lets bound objects send updates, Modify MenuMaker to notify you when the GeneratedDate property changes, Brian runs into file trouble, Windows Store apps use await to be more responsive, Build a slightly less simple text editor, Use async methods to find and open files, Build Brian a new Excuse Manager app, Build Brian a new Excuse Manager app, Jimmy could use some help, Use the IDE to explore app page navigation, Start building Jimmy an app, Add the new queries to Jimmy's app, Combine Jimmy's values into groups, Add semantic zoom to Jimmy's app, The IDE's Split App template helps you build apps for navigating data, #1. There's so much more to Windows Store, WPF applications use XAML to create UI objects, Page layout starts with controls, Rows and columns can resize to match the page size, Rows and columns can resize to match the page size, Use data binding to build Sloppy Joe a better menu, Use static resources to declare your objects in XAML, INotifyPropertyChanged lets bound objects send updates, C# programs can use await to be more responsive

building with WPF for operating systems before Windows 8, Here's what you're going to build

creating new project for, When you're doing this...

data binding, connecting XAML pages to classes, Data binding connects your XAML pages to your classes

exploring app page navigation using the IDE, Use the IDE to explore app page navigation

INotifyPropertyChanged, letting bound objects send updates, INotifyPropertyChanged lets bound objects send updates, INotifyPropertyChanged lets bound objects send updates

learning more about programming, #1. There's so much more to Windows Store

managing Jimmy's comics collection, Jimmy could use some help, Start building Jimmy an app, Add the new queries to Jimmy's app, Combine Jimmy's values into groups, Add semantic zoom to Jimmy's app, The IDE's Split App template helps you build apps for navigating data

combining values into groups, Combine Jimmy's values into groups

semantic zoom, Add semantic zoom to Jimmy's app

Split App for navigating data, The IDE's Split App template helps you build apps for navigating data

.NET for, tools for building apps, The .NET Framework gives you the right tools for the job

protecting your filesystem, Use async methods to find and open files

publishing apps to Windows Store, Publish your app

rebuilding Excuse Manager as, Build Brian a new Excuse Manager app

redesigning GoFish! form as app page, Redesign the Go Fish! form as a Windows Store app page, Page layout starts with controls, Rows and columns can resize to match the page size, Use the grid system to lay out app pages, Modify MenuMaker to notify you when the GeneratedDate property

changes, Page layout starts with controls, Rows and columns can resize to match the page size

finishing conversion, Modify MenuMaker to notify you when the GeneratedDate property changes

page layout starting with controls, Page layout starts with controls, Page layout starts with controls

rows and columns resizing to match screen size, Rows and columns can resize to match the page size, Rows and columns can resize to match the page size

using grid system to lay out pages, Use the grid system to lay out app pages

redesigning Windows Desktop forms as, Use the grid system to lay out app pages, Use the grid system to lay out app pages, Rows and columns can resize to match the page size

running in Visual Studio simulator, Build Brian a new Excuse Manager app

superior IO tools, Brian runs into file trouble

text editor, Build a slightly less simple text editor

using awit to be more responsive, Windows Store apps use await to be more responsive, C# programs can use await to be more responsive

using data binding to build better menu, Use data binding to build Sloppy Joe a better menu, Use data binding to build Sloppy Joe a better menu

using data template to display objects, Use a data template to display objects, Use a data template to display objects

using static resources to declare objects in XAML, Use static resources to declare your objects in XAML, Use static resources to declare your objects in XAML

using XAML to create UI objects, Windows Store apps use XAML to create UI objects, WPF applications use XAML to create UI objects

Windows UI controls., Here’s what you’re going to build (see controls)

Windows.UI namespace, UI controls can be instantiated with C# code, too, UI controls can be instantiated with C# code, too

Windows.UI.Xaml.Conrols namespace, The .NET Framework gives you the right tools for the job

Windows.UI.Xaml namespace, Use ItemsPanelTemplate to bind controls to a Canvas, Use ItemsPanelTemplate to bind controls to a Canvas

WinForms apps, Brian’s running Windows 8, Brian’s running Windows 8, Windows Forms use an object graph set up by the IDE, XAML controls can contain text...and more, Use the FileIO class to read and write files, #4. Use BackgroundWorker to make your WinForms responsive

GDI+ grpahics, Brian's running Windows 8

reasons for learning, XAML controls can contain text...and more

using BackgroundWorker to make apps responsive, #4. Use BackgroundWorker to make your WinForms responsive

using object graph set up by IDE, Windows Forms use an object graph set up by the IDE

using System.IO.File to read/write files, Use the FileIO class to read and write files

versus Windows Store apps, Brian's running Windows 8

WPF (Windows Presentation Foundation), Here's what you're going to build, Start with a blank application

X

XAML, C# and the Visual Studio IDE make lots of things easy, What you do in Visual Studio..., What Visual Studio does for you..., Here's what you're going to build, Set up the grid for your page, Set up the grid for your page, Add controls to your grid, Add controls to your grid, Add controls to your grid, Use properties to change how the controls look, Controls make the game work, Controls make the game work, Finish the method and run your program, Here's what you've done so far, Dragging humans onto enemies ends the game, Make your enemies look like aliens, ...the IDE does this, Brian's running Windows 8, Windows Store apps use XAML to create UI objects, Use the grid system to lay out app pages, Use the grid system to lay out app pages, Use the grid system to lay out app pages, Data binding connects your XAML pages to your classes, XAML controls can contain text...and more, XAML controls can contain text...and more, Use static resources to declare your objects in XAML, WPF applications use XAML to create UI objects, Rows and columns can resize to match the page size, Use static resources to declare your objects in XAML

application code, Here's what you're going to build

changes to code from changes made in IDE, ...the IDE does this

combining with C#, creating wisual programs, C# and the Visual Studio IDE make lots of things easy

controls, containing text and other controls, XAML controls can contain text...and more

data binding in, Data binding connects your XAML pages to your classes

defined, What Visual Studio does for you...

editing templates, Dragging humans onto enemies ends the game, Make your enemies look like aliens

file created by Solution Explorer on creating new project in Visual Studio, What you do in Visual Studio...

flexibility with tag order, Use the grid system to lay out app pages

generated for controls dragged from Toolbox onto page, Set up the grid for your page, Add controls to your grid

page design with, WinForms versus, XAML controls can contain text...and more

properties, Set up the grid for your page, Add controls to your grid, Add controls to your grid, Use properties to change how the controls look, Controls make the game work, Controls make the game work, Finish the method and run your program, Here's what you've done so far

using to change controls, Add controls to your grid, Use properties to change how the controls look

redesigning Windows Desktop forms, Use the grid system to lay out app pages, Use the grid system to lay out app pages, Rows and columns can resize to match the page size

using static resources to declare objects in, Use static resources to declare your objects in XAML, Use static resources to declare your objects in XAML

using to create UI for Windows Store apps, Windows Store apps use XAML to create UI objects, WPF applications use XAML to create UI objects

XML, Wait, wait! What did that say?, Use static resources to declare your objects in XAML, A data contract is an abstract definition of your object's data, The whole object graph is serialized to XML, User controls let you create your own controls, #2. The Basics, #10. LINQ to XML, Use static resources to declare your objects in XAML, User controls let you create your own controls

comments, Wait, wait! What did that say?, #2. The Basics

LINQ to XML, #10. LINQ to XML

namespace, Use static resources to declare your objects in XAML, A data contract is an abstract definition of your object's data, User controls let you create your own controls, Use static resources to declare your objects in XAML, User controls let you create your own controls

xmlns, Use static resources to declare your objects in XAML, User controls let you create your own controls, Use static resources to declare your objects in XAML, User controls let you create your own controls

output of data contract serializer, The whole object graph is serialized to XML

x:Name and x:Key properties, static resources, Use a data template to display objects, Use a data template to display objects

x:Name property, if/else statements make decisions, Build an app from the ground up

XOR operator (^), #6. Equality, IEquatable, and Equals()

XOR operator (~), ...more basics...

Y

yield return, using to create enumerable objects, #7. Using yield return to create enumerable objects

Z

zoom, semantic zoom control, Use semantic zoom to navigate your data

Zoo Simulator project, How would you design a zoo simulator?, Create the class hierarchy, Create the class hierarchy, Create the class hierarchy

class hierarchy, Create the class hierarchy

extending base class, Create the class hierarchy

inheriting from base class, Create the class hierarchy

About the Authors

Bestselling O'Reilly authors Jennifer Greene and Andrew Stellman have been building software and writing about software engineering together since they first met in 1998. Their first book, Applied Software Project Management, was published by O'Reilly in 2005. Other Stellman and Greene books for O'Reilly include Beautiful Teams (2009), and their first book in the Head First series, Head First PMP (2007). They founded Stellman & Greene Consulting in 2003 to build a really neat software project for scientists studying herbicide exposure in Vietnam vets. In addition to building software and writing books, they've provided training and consulted for companies and spoken at conferences and meetings of software engineers, architects and project managers.

Jenny studied philosophy in college but, like everyone else in the field, couldn't find a job doing it. Luckily, she's a great software engineer, so she started out working at an online service, and that's the first time she really got a good sense of what good software development looked like. She moved to New York in 1998 to work on software quality at a financial software company. She's managed a teams of developers, testers and PMs on software projects in media and finance since then. She's traveled all over the world to work with different software teams and build all kinds of cool projects.

Andrew, despite being raised a New Yorker, has lived in Minneapolis, Geneva, and Pittsburgh... twice. The first time was when he graduated from Carnegie Mellon's School of Computer Science, and then again when he and Jenny were starting their consulting business and writing their first book for O'Reilly. He and Jenny first worked together at a company on Wall Street that built financial software, where he was managing a team of programmers. Over the years he's been a Vice President at a major investment bank, architected large-scale real-time back end systems, managed large international software teams, and consulted for companies, schools, and organizations, including Microsoft, the National Bureau of Economic Research, and MIT. He's had the privilege working with some pretty amazing programmers during that time, and likes to think that he's learned a few things from them.

Special Upgrade Offer

If you purchased this ebook from a retailer other than O’Reilly, you can upgrade it for $4.99 at oreilly.com by clicking here.

Head First C#

Jennifer Greene

Andrew Stellman

Editor
Courtney Nash

Copyright © 2013 Andrew Stellman, Jennifer Greene

Head First C#

Third Edition

by Andrew Stellman and Jennifer Greene

All rights reserved.

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

Series Creators:	Kathy Sierra, Bert Bates
Cover Designers:	Louise Barr, Karen Montgomery
Production Editor:	Melanie Yarbrough
Proofreader:	Rachel Monaghan
Indexer:	Ellen Troutman-Zaig
Page Viewers:	Quentin the whippet and Tequila the pomeranian

Printing History:

November 2007: First Edition.

May 2010: Second Edition.

August 2013: Third Edition.

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

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

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

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

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

[LSI]	[2014-09-12]

O’Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472

2014-09-23T09:12:34-07:00