Head First C# is a complete learning experience for programming with C#, XAML, the .NET Framework, and Visual Studio. Built for your brain, this book keeps you engaged from the first chapter, where you’ll build a fully functional video game. After that, you’ll learn about classes and object-oriented programming, draw graphics and animation, query your data with LINQ, and serialize it to files. And you’ll do it all by building games, solving puzzles, and doing hands-on projects. By the time you’re done you’ll be a solid C# programmer, and you’ll have a great time along the way!

What will you learn from this book?

Head First C# is a complete learning experience for programming with C#, XAML, the .NET Framework, and Visual Studio. Built for your brain, this book keeps you engaged from the first chapter, where you’ll build a fully functional video game. After that, you’ll learn about classes and object-oriented programming, draw graphics and animation, query your data with LINQ, and serialize it to files. And you’ll do it all by building games, solving puzzles, and doing hands-on projects. By the time you’re done you’ll be a solid C# programmer, and you’ll have a great time along the way!

Why does this book look so different?

We think your time is too valuable to spend struggling with new concepts. Using the latest research in cognitive science and learning theory to craft a multi-sensory learning experience, Head First C# uses a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.

Why does this book look so different?

We think your time is too valuable to spend struggling with new concepts. Using the latest research in cognitive science and learning theory to craft a multi-sensory learning experience, Head First C# uses a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.

"If you want to learn C# in depth and have fun doing it, this is the book for you."
—Andy Peake, fledgling C# programmer

"Head First C# will guide beginners of all sorts to a long and productive relationship with C# and the .NET Framework."
—Chris Burrows, Developer on Microsoft’s C# Compiler team

"Head First C# got me up to speed in no time for my first large scale C# development project at work—I highly recommend it."
—Shelana Odumowo, Technical Account Manager, Google

We think your time is too valuable to spend struggling with new concepts. Using the latest research in cognitive science and learning theory to craft a multi-sensory learning experience, Head First C# uses a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.

Why does this book look so different?

We think your time is too valuable to spend struggling with new concepts. Using the latest research in cognitive science and learning theory to craft a multi-sensory learning experience, Head First C# uses a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.
Head First C# is a complete learning experience for programming with C#, XAML, the .NET Framework, and Visual Studio. Built for your brain, this book keeps you engaged from the first chapter, where you'll build a fully functional video game. After that, you'll learn about classes and object-oriented programming, draw graphics and animation, query your data with LINQ, and serialize it to files. And you'll do it all by building games, solving puzzles, and doing hands-on projects. By the time you're done you'll be a solid C# programmer, and you'll have a great time along the way!

Why does this book look so different?

We think your time is too valuable to spend struggling with new concepts. Using the latest research in cognitive science and learning theory to craft a multi-sensory learning experience, Head First C# uses a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.

What will you learn from this book?

Head First C# is a complete learning experience for programming with C#, XAML, the .NET Framework, and Visual Studio. Built for your brain, this book keeps you engaged from the first chapter, where you'll build a fully functional video game. After that, you'll learn about classes and object-oriented programming, draw graphics and animation, query your data with LINQ, and serialize it to files. And you'll do it all by building games, solving puzzles, and doing hands-on projects. By the time you're done you'll be a solid C# programmer, and you'll have a great time along the way!

Why does this book look so different?

We think your time is too valuable to spend struggling with new concepts. Using the latest research in cognitive science and learning theory to craft a multi-sensory learning experience, Head First C# uses a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.

What will you learn from this book?

Head First C# is a complete learning experience for programming with C#, XAML, the .NET Framework, and Visual Studio. Built for your brain, this book keeps you engaged from the first chapter, where you'll build a fully functional video game. After that, you'll learn about classes and object-oriented programming, draw graphics and animation, query your data with LINQ, and serialize it to files. And you'll do it all by building games, solving puzzles, and doing hands-on projects. By the time you're done you'll be a solid C# programmer, and you'll have a great time along the way!

Why does this book look so different?

We think your time is too valuable to spend struggling with new concepts. Using the latest research in cognitive science and learning theory to craft a multi-sensory learning experience, Head First C# uses a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.

What will you learn from this book?

Head First C# is a complete learning experience for programming with C#, XAML, the .NET Framework, and Visual Studio. Built for your brain, this book keeps you engaged from the first chapter, where you'll build a fully functional video game. After that, you'll learn about classes and object-oriented programming, draw graphics and animation, query your data with LINQ, and serialize it to files. And you'll do it all by building games, solving puzzles, and doing hands-on projects. By the time you're done you'll be a solid C# programmer, and you'll have a great time along the way!

Why does this book look so different?

We think your time is too valuable to spend struggling with new concepts. Using the latest research in cognitive science and learning theory to craft a multi-sensory learning experience, Head First C# uses a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.
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 Sempheore 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 recommend 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*
Praise for other Head First books

“I received the book yesterday and started to read it…and I couldn’t stop. This is definitely trè ‘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


— 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
WON‘T IT BE DREAMY IF THERE WAS A C# BOOK THAT WAS MORE FUN THAN MEMORIZING A PHONE BOOK? IT’S PROBABLY NOTHING BUT A FANTASY....

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

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

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.

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.

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 (2007), 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
# Table of Contents (Summary)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intro</td>
<td>xxxi</td>
</tr>
<tr>
<td>1 Start building with C#: Building something cool, fast!</td>
<td>1</td>
</tr>
<tr>
<td>2 It's All Just Code: Under the hood</td>
<td>53</td>
</tr>
<tr>
<td>3 Objects: Get Oriented: Making code make sense</td>
<td>101</td>
</tr>
<tr>
<td>4 Types and References: It's 10:00. Do you know where your data is?</td>
<td>141</td>
</tr>
<tr>
<td>C# Lab 1: A Day at the races</td>
<td>187</td>
</tr>
<tr>
<td>5 Encapsulation: Keep your privates…private</td>
<td>197</td>
</tr>
<tr>
<td>6 Inheritance: Your object's family tree</td>
<td>237</td>
</tr>
<tr>
<td>7 Interfaces and abstract classes: Making classes keep their promises</td>
<td>293</td>
</tr>
<tr>
<td>8 Enums and collections: Storing lots of data</td>
<td>351</td>
</tr>
<tr>
<td>9 Reading and Writing Files: Save the last byte for me!</td>
<td>409</td>
</tr>
<tr>
<td>C# Lab 2: The Quest</td>
<td>465</td>
</tr>
<tr>
<td>10 Designing Windows Store Apps with XAML: Taking your apps to the next level</td>
<td>487</td>
</tr>
<tr>
<td>11 XAML, File, I/O, and Data Contract Serialization: Writing files right</td>
<td>535</td>
</tr>
<tr>
<td>12 Exception Handling: Putting out fires gets old</td>
<td>569</td>
</tr>
<tr>
<td>13 Captain Amazing: The Death of the Object</td>
<td>611</td>
</tr>
<tr>
<td>14 Querying Data and Building Apps with LINQ: Get control of your data</td>
<td>649</td>
</tr>
<tr>
<td>15 Events and Delegates: What your code does when you’re not looking</td>
<td>701</td>
</tr>
<tr>
<td>16 Architecting Apps with the MVVM Pattern: Great apps on the inside and outside</td>
<td>745</td>
</tr>
<tr>
<td>C# Lab 3: Invaders</td>
<td>807</td>
</tr>
<tr>
<td>i Leftovers: The top 10 things we wanted to include in this book</td>
<td>845</td>
</tr>
<tr>
<td>ii Windows presentation foundation: WPF Learner’s Guide to Head First C#</td>
<td>861</td>
</tr>
</tbody>
</table>

# Table of Contents (the real thing)

## Intro

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

- Who is this book for?.xxxii
- We know what you’re thinking.xxxiii
- Metacognition: thinking about thinking.xxxv
- Here’s what YOU can do to bend your brain into submission.xxxvii
- What you need for this book.xxxviii
- Read me.xxxxi
- The technical review team.xl
- Acknowledgments.xli
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 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  
- 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
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…                        54
…the IDE does this                              55
Where programs come from                       56
The IDE helps you code                          58
Anatomy of a program                           60
Two classes can be in the same namespace       65
Your programs use variables to work with data 66
C# uses familiar math symbols                  68
Use the debugger to see your variables change  69
Loops perform an action over and over          71
if/else statements make decisions              72
Build an app from the ground up                73
Make each button do something                  75
Set up conditions and see if they’re true      76
Windows Desktop apps are easy to build         87
Rebuild your app for Windows Desktop           88
Your desktop app knows where to start          92
You can change your program’s entry point     94
When you change things in the IDE, you’re also changing your code 96
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
How Mike’s car navigation system thinks about his problems
Mike’s Navigator class has methods to set and modify routes
Use what you’ve learned to build a program that uses a class
Mike gets an idea
Mike can use objects to solve his problem
You use a class to build an object
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!
An instance uses fields to keep track of things
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
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
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
- 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
- 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
- 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
- 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
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.
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
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 keeps your data pristine
Properties make encapsulation easier
Build an application to test the Farmer class
Use automatic properties to finish the class
What if we want to change the feed multiplier?
Use a constructor to initialize private fields
inheritance

Your object’s family tree

Sometimes you \textit{DO} want to be just like your parents.

Ever run across an object that \textit{almost} does exactly what you want \textit{your} object to do? Found yourself wishing that if you could just \textit{change a few things}, that object would be perfect? Well, that’s just one reason that \textit{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 \textbf{subclass} an object to get its behavior, but keep the \textbf{flexibility} to make changes to that behavior. You’ll \textbf{avoid duplicate code, model the real world} more closely, and end up with code that’s \textbf{easier to maintain}.

<table>
<thead>
<tr>
<th>Kathleen does birthday parties, too</th>
<th>238</th>
</tr>
</thead>
<tbody>
<tr>
<td>We need a BirthdayParty class</td>
<td>239</td>
</tr>
<tr>
<td>Build the Party Planner version 2.0</td>
<td>240</td>
</tr>
<tr>
<td>One more thing…can you add a $100 fee for parties over 12?</td>
<td>247</td>
</tr>
<tr>
<td>When your classes use inheritance, you only need to write your code once</td>
<td>248</td>
</tr>
<tr>
<td>Build up your class model by starting general and getting more specific</td>
<td>249</td>
</tr>
<tr>
<td>How would you design a zoo simulator?</td>
<td>250</td>
</tr>
<tr>
<td>Use inheritance to avoid duplicate code in subclasses</td>
<td>251</td>
</tr>
<tr>
<td>Different animals make different noises</td>
<td>252</td>
</tr>
<tr>
<td>Think about how to group the animals</td>
<td>253</td>
</tr>
<tr>
<td>Create the class hierarchy</td>
<td>254</td>
</tr>
<tr>
<td>Every subclass extends its base class</td>
<td>255</td>
</tr>
<tr>
<td>Use a colon to inherit from a base class</td>
<td>256</td>
</tr>
<tr>
<td>We know that inheritance adds the base class fields, properties, and methods to the subclass…</td>
<td>259</td>
</tr>
<tr>
<td>A subclass can override methods to change or replace methods it inherited</td>
<td>260</td>
</tr>
<tr>
<td>Any place where you can use a base class, you can use one of its subclasses instead</td>
<td>261</td>
</tr>
<tr>
<td>A subclass can hide methods in the superclass</td>
<td>268</td>
</tr>
<tr>
<td>Use the override and virtual keywords to inherit behavior</td>
<td>270</td>
</tr>
<tr>
<td>A subclass can access its base class using the base keyword</td>
<td>272</td>
</tr>
<tr>
<td>When a base class has a constructor, your subclass needs one, too</td>
<td>273</td>
</tr>
<tr>
<td>Now you’re ready to finish the job for Kathleen!</td>
<td>274</td>
</tr>
<tr>
<td>Build a beehive management system</td>
<td>279</td>
</tr>
<tr>
<td>How you’ll build the beehive management system</td>
<td>280</td>
</tr>
</tbody>
</table>
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

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

The RoboBee 4000 can do a worker bee’s job without using valuable honey

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

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

An abstract method doesn’t have a body

The Deadly Diamond of Death!

Polymorphism means that one object can take many different forms
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

 Enums let you work with a set of valid values

 Enums let you represent numbers with names

 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

 Lists are easy, but **SORTING** can be tricky

 `IComparable<Duck>` helps your list sort its ducks

 Use `IComparer` to tell your List how to sort

 Create an instance of your comparer object

 `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

 When you write a foreach loop, you’re using `IEnumerable<T>`

 You can upcast an entire list using `IEnumerable`

 You can build your own overloaded methods

 Use a dictionary to store keys and values

 The dictionary functionality rundown

 Build a program that uses a dictionary

 And yet **MORE** collection types…

 A queue is FIFO—First In, First Out

 A stack is LIFO—Last In, First Out
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.
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.

The spec: build an adventure game

The fun’s just beginning!
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 8 488
- Windows Forms use an object graph set up by the IDE 494
- Use the IDE to explore the object graph 497
- Windows Store apps use XAML to create UI objects 498
- Redesign the Go Fish! form as a Windows Store app page 500
- Page layout starts with controls 502
- Rows and columns can resize to match the page size 504
- Use the grid system to lay out app pages 506
- Data binding connects your XAML pages to your classes 512
- XAML controls can contain text...and more 514
- Use data binding to build Sloppy Joe a better menu 516
- Use static resources to declare your objects in XAML 522
- Use a data template to display objects 524
- INotifyPropertyChanged lets bound objects send updates 526
- Modify MenuMaker to notify you when the GeneratedDate property changes 527
Writing files right

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 536
Windows Store apps use await to be more responsive 538
Use the FileIO class to read and write files 540
Build a slightly less simple text editor 542
A data contract is an abstract definition of your object's data 547
Use async methods to find and open files 548
KnownFolders helps you access high-profile folders 550
The whole object graph is serialized to XML 551
Stream some Guy objects to your app's local folder 552
Take your Guy Serializer for a test drive 556
Use a Task to call one async method from another 557
Build Brian a new Excuse Manager app 558
Separate the page, excuse, and Excuse Manager 559
Create the main page for the Excuse Manager 560
Add the app bar to the main page 561
Build the ExcuseManager class 562
Add the code-behind for the page 564
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

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

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

A few simple ideas for exception handling

```csharp
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your last chance to DO something…your object’s finalizer</td>
<td>618</td>
</tr>
<tr>
<td>When EXACTLY does a finalizer run?</td>
<td>619</td>
</tr>
<tr>
<td>Dispose() works with using; finalizers work with garbage collection</td>
<td>620</td>
</tr>
<tr>
<td>Finalizers can’t depend on stability</td>
<td>622</td>
</tr>
<tr>
<td>Make an object serialize itself in its Dispose()</td>
<td>623</td>
</tr>
<tr>
<td>A struct looks like an object…</td>
<td>627</td>
</tr>
<tr>
<td>…but isn’t an object</td>
<td>627</td>
</tr>
<tr>
<td>Values get copied; references get assigned</td>
<td>628</td>
</tr>
<tr>
<td>Structs are value types; objects are reference types</td>
<td>629</td>
</tr>
<tr>
<td>The stack vs. the heap: more on memory</td>
<td>631</td>
</tr>
<tr>
<td>Use out parameters to make a method return more than one value</td>
<td>634</td>
</tr>
<tr>
<td>Pass by reference using the ref modifier</td>
<td>635</td>
</tr>
<tr>
<td>Use optional parameters to set default values</td>
<td>636</td>
</tr>
<tr>
<td>Use nullable types when you need nonexistent values</td>
<td>637</td>
</tr>
<tr>
<td>Nullable types help you make your programs more robust</td>
<td>638</td>
</tr>
<tr>
<td>“Captain” Amazing…not so much</td>
<td>641</td>
</tr>
<tr>
<td>Extension methods add new behavior to EXISTING classes</td>
<td>642</td>
</tr>
<tr>
<td>Extending a fundamental type: string</td>
<td>644</td>
</tr>
</tbody>
</table>
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...  
- ...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  
- 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
events and delegates

15

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? 702
But how does an object KNOW to respond? 702
When an EVENT occurs…objects listen 703
One object raises its event, others listen for it… 704
Then, the other objects handle the event 705
Connecting the dots 706
The IDE generates event handlers for you automatically 710
Generic EventHandlers let you define your own event types 716
Windows Forms use many different events 717
One event, multiple handlers 718
Windows Store apps use events for process lifetime management 720
Add process lifetime management to Jimmy’s comics 721
XAML controls use routed events 724
Create an app to explore routed events 725
Connecting event senders with event listeners 730
A delegate STANDS IN for an actual method 731
Delegates in action 732
An object can subscribe to an event… 735
Use a callback to control who’s listening 736
A callback is just a way to use delegates 738
You can use callbacks with MessageDialog commands 740
Use delegates to use the Windows settings charm 742
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 746
But can they agree on how to build it? 747
Do you design for binding or for working with data? 748
MVVM lets you design for binding and data 749
Use the MVVM pattern to start building the basketball roster app 750
User controls let you create your own controls 753
The ref needs a stopwatch 761
MVVM means thinking about the state of the app 762
Start building the stopwatch app’s Model 763
Events alert the rest of the app to state changes 764
Build the view for a simple stopwatch 765
Add the stopwatch ViewModel 766
Converters automatically convert values for binding 770
Converters can work with many different types 772
Visual states make controls respond to changes 778
Use DoubleAnimation to animate double values 779
Use object animations to animate object values 780
Build an analog stopwatch using the same ViewModel 781
UI controls can be instantiated with C# code, too 786
C# can build “real” animations, too 788
Create a user control to animate a picture 789
Make your bees fly around a page 790
Use ItemsPanelTemplate to bind controls to a Canvas 793
Congratulations! (But you’re not done yet...) 806
C# Lab 3

Invaders

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.

The grandfather of video games 808
And yet there’s more to do… 829
appendix i: 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 846
#2. The Basics 848
#3. Namespaces and assemblies 854
#4. Use BackgroundWorker to make your WinForms responsive 858
#5. The Type class and GetType() 861
#6. Equality, IComparable, and Equals() 862
#7. Using yield return to create enumerable objects 865
#8. Refactoring 868
#9. Anonymous types, anonymous methods, and lambda expressions 870
#10. LINQ to XML 872

Did you know that C# and the .NET Framework can… 875
WPF Learner’s Guide to Head First C#


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
Build WPF projects in Visual Studio
How to use this appendix
Start with a blank application
Use properties to change how the controls look
Add a method that does something
Finish the method and run your program
Add timers to manage the gameplay
Add code to make your controls interact with the player
Build an app from the ground up
Redesign the Go Fish! form as a WPF application
Use data binding to build Sloppy Joe a better menu
Use a data template to display objects
INotifyPropertyChanged lets bound objects send updates
What happens when a method you want to call is risky?
Build a WPF comic query application
Create an app to explore routed events
Build the view for a simple stopwatch
Build an analog stopwatch using the same ViewModel
Create a user control to animate a picture
Use ItemsPanelTemplate to bind controls to a Canvas
Congratulations! (But you’re not done yet...)
how to use this book

Intro

I can’t believe they put that in a C# programming book!

In this section, we answer the burning question: “So why DID they put that in a C# programming book?”
Who is this book for?

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

1. Do you want to **learn** C#?
2. Do you like to tinker—do you learn by doing, rather than just reading?
3. Do you prefer **stimulating dinner party conversation** to **dry, dull, academic lectures**?

this book is for you.

Who should probably back away from this book?

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

1. Does the idea of doing projects and building programs make you bored and a little twitchy?
2. Are you a really advanced C++ programmer looking for a dry reference book?
3. 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?

this book is not for you.

---

**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.

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.

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

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

3. **Read the “There are No Dumb Questions”**
   That means all of them. They’re not optional sidebars—they’re part of the core content! Don’t skip them.

4. **Make this the last thing you read before bed. Or at least the last challenging thing.**
   Part of the learning (especially the transfer to long-term memory) happens after you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

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

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

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

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

9. **Write a lot of 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.

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.

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 II, 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, Chapters 3 through 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.

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

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:

The Visual Studio home page also has many useful download links: http://www.microsoft.com/visualstudio
You’ll move on to create desktop applications

Chapters 1 and 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.

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.

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

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.
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.

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());
        }
    }
}
```

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.

The result is a better-looking app that takes less time to write.

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#, the .NET Framework, and the Visual Studio IDE have prebuilt structures that handle the tedious code that’s part of most programming tasks.

Let’s get started!

How good?

- Good
- Better
- Best

Value

- Excitement
- Boredom
- Brain

On
Off
On
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:

1. **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.

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

3. **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.

4. **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.**
let's get started

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).

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.

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

 MainPage.xaml

Visual Studio creates all three of these files automatically. It creates several other files as well! You can see them in the Solution Explorer window.

The C# code that controls the main page’s behavior lives here.

 MainPage.Xaml.cs

To see this file, you need to expand App.xaml, just like you need to look under MainPage.xaml to see MainPage.xaml.cs

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.

This file contains the C# code that’s run when the app is launched or resumed.

 App.xaml.cs

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.

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

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-to-use interface in the IDE.

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

A: XAML (the X is pronounced like Z, and it rhymes with "camel") is a markup language that you'll use to build your user interfaces for your 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:

```xml
<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: 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: 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.
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!

if only humans weren’t so delicious
Only you 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?

More and more evil aliens will fill up the screen. If you drag your human into one, “Game over, man!”

Drag the human into the target before the timer at the bottom of the page runs out.

Don’t drag your human too quickly or you’ll lose him.

Our greatest human scientific minds have invented protective interdimensional diamond-shaped portals to protect the human race.

It’s up to YOU to SAVE THE HUMANS by guiding them safely to their target portals.
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:

You’ll be building an app that has a main page with a bunch of visual controls on it.

**XAML Main Page and Containers**

**Windows UI Controls**

The app uses controls to provide gameplay for the player.

The Target timer checks the ProgressBar’s properties to see if the player ran out of time.

Each human that the player has to save is drawn using a StackPanel, which contains an ellipse and a rectangle.

You’ll lay out the main page using a grid. The gameplay will take place in the center cell of the grid—we’ll use a Canvas for that.

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.

You’ll write C# code that manipulates the controls and makes the game work.

C# Code

Deployment Package

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.

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.

Start building with C#
fill in the blanks

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.

1. 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 designer shows you a preview of the page that you’re working on. It looks like a blank page with a default black background.

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

![XAML Grid Code](image)

**These are the opening and closing tags for a grid that contains controls.** When you add rows, columns, and controls to the grid, the code for them will go between these opening and closing tags.

*You can see the XAML code for the blank grid that the IDE generated for you. Keep your eyes on it—we’ll add some columns and rows in a minute.*

**You are here!**

---

**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!
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:

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

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:

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!

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.

If you really get stuck, you can download all of the code for this project from the book’s website: http://www.headfirstlabs.com/hfcsharp — all of the code in this chapter was copied and pasted from the downloadable source!
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.

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.

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!

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:

```
<Grid GridBackground="{(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="39,55,39,0" Command="{Binding NavigationHelper.GoBackCommand}"/>
    </Grid>
</Grid>
```

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

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:

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

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

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#.
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.

Here’s the width of the left column you created in step 5—the width matches the width that you saw in the designer. That’s because the IDE generated this XAML code for you.

When you added the new Basic Page, it already had a top row with a height of 140 pixels. It contains the back button and page name.

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.

1. **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:

   ![Image](image1)

2. **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*.

![Image](image2)

**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!
### 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.

When you enter 1* into the box, the IDE sets the column to its default width. It might adjust the other columns. If it does, just reset them back to 160 pixels.

---

### 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.

```xml
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="160"/>
    <ColumnDefinition/>
    <ColumnDefinition Width="160"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="140"/>
    <RowDefinition/>
    <RowDefinition Height="160"/>
  </Grid.RowDefinitions>
</Grid>```

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

---

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

When you enter 1* into the box, the IDE sets the column to its default width. It might adjust the other columns. If it does, just reset them back to 160 pixels.
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.

1. 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.

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

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

2. 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.

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

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.

We added line breaks to make the XAML easier to read. You can add line breaks too. Give it a try!
Next, expand the \(\text{All XAML Controls}\) section of the toolbox. Drag a \(\text{ProgressBar}\) into the bottom-center cell, a \(\text{ContentControl}\) into the bottom-right cell (make sure it’s \text{below} the TextBlock you already put in that cell), and a \(\text{Canvas}\) 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):

![Diagram of XAML control placement]

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 \text{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, \text{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.

1. **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:

   ```xml
   <Button Content="Start!" HorizontalAlignment="Left" VerticalAlignment="Top"/>
   ```

2. **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.

   Use the Name box to change the name of the control to startButton.

   ```xml
   <Button x:Name="startButton" Content="Start!"
           Grid.Row="2"
           HorizontalAlignment="Center"
           VerticalAlignment="Center"/>
   ```

   These little squares tell you if the property has been set. A filled square means it’s been set; an empty square means it’s been left with a default value.

   When you edit the text in the button, the IDE updates the Content property in the XAML.

   Use the buttons to set the HorizontalAlignment and VerticalAlignment properties to “Center” and center the button in the cell.

   When you dragged the button onto the page, the IDE used the Margin property to place it in an exact position in the cell. Click on the square and choose Reset from the menu to reset the margins to 0.

   The properties may be in a different order. That’s OK!
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:

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

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:

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

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

### Save the Humans

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 SubheaderTextStyle to make its text bigger.

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.
you want your game to work, right?

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:

1. 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:

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

2. 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`.

   After you’ve named the Canvas control, you can close the Document Outline window. Then use the `View` and `Name` buttons in the Properties window to set its vertical and horizontal alignments to Stretch, reset the margins, and click both `Width` and `Height` 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 `Color` 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.
5 **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.

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** $\rightarrow$ **Edit Current**.

4 **Edit the enemy template.**

Add a red circle to the template:

- Double-click on $\text{Ellipse}$ 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 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 $\text{Solid Color}$ 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:

```xml
<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.

5 **Use the Document Outline to modify the StackPanel and TextBlock controls.**

Go back to the Document Outline (if you see $\downarrow$ at the top of the Document Outline window, just click $\uparrow$ 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.
6 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:

```xml
<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:

![Diagram of controls]

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

7 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.
Start building with C#.

8 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.

9 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 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.

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

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

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.

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.

...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.
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:

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

Use the IDE to create your own method

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

```csharp
private void startButton_Click(object sender, RoutedEventArgs e)
{
    // The IDE created this method. It will run when a user clicks the "Start!" button in the running application.
    AddEnemy();
}
```

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!

Q: What’s a method?
A: A method is just a named block of code. We’ll talk a lot more about methods in Chapter 2.

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

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

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

2. 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.

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

3. Finish adding the first line of code. You’ll get another IntelliSense window after you type `new`.

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

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!

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.
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):

```csharp
/// <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();
}
```

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

```csharp
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.

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`.

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

The IDE may generate the method stub with “int” types. Change them to “double”. You’ll learn about types in Chapter 4.
ok, that's pretty cool

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!

1 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:

```csharp
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;
```

2 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.

```csharp
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();
}
```

3 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.

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.

And you’ll learn about animation in Chapter 16.
You’ll learn about object initializers like this in Chapter 4.
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.

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

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

You’ll learn about using the error window and debugging your code in Chapter 2.
4 Start your program.
Find the button at the top of the IDE. This starts your program running.

5 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.

6 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.

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

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

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.

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.

You are here!

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

<table>
<thead>
<tr>
<th>XAML property</th>
<th>Where to find it in the Properties window in the IDE</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>At the top</td>
<td>At the top</td>
</tr>
<tr>
<td>Height</td>
<td>Brush, Appearance, Common, Layout, Transform</td>
<td>Determines how tall the control should be</td>
</tr>
<tr>
<td>Rotation</td>
<td></td>
<td>Sets the angle that the control is turned</td>
</tr>
<tr>
<td>Fill</td>
<td></td>
<td>You use this in your C# code to manipulate a specific control</td>
</tr>
<tr>
<td>x:Name</td>
<td></td>
<td>The color of the control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use this when you want text or graphics in your control</td>
</tr>
</tbody>
</table>

Remember how you set the Name of the Canvas control to "playArea"? That set its "x:Name" property in the XAML, which will come in handy in a minute when you write C# code to work with the Canvas.
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.

1. **ADD MORE LINES TO THE TOP OF YOUR C# CODE.**
   Go up to the top of the file where you added that Random line. Add three more lines:
   ```csharp
   /// <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;
   }
   
   2. **ADD A METHOD FOR ONE OF YOUR TimERS.**
   Find this code that the IDE generated:
   ```csharp
   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:
   ```csharp
   enemyTimer.Tick += enemyTimer_Tick;  // (Press TAB to insert)
   
   Press the Tab key. The IDE will pop up another box:
   ```csharp
   enemyTimer.Tick += enemyTimer_Tick;
   
   Press Tab one more time. Here’s the code the IDE generated for you:
   ```csharp
   ...
   enemyTimer.Tick += enemyTimer_Tick;
   
   void enemyTimer_Tick(object sender, object e)
   {
     throw new NotImplementedException();
   }
   
   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.

   The IDE generated a method for you called an event handler. You’ll learn about event handlers in Chapter 15.
**3 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:

```csharp
public MainPage()
{
    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?*

**4 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:

```csharp
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.

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:

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

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.

1. **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:
   ```csharp
   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.

   It’s normal to add parentheses () when writing about a method.

2. **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:
   ```csharp
   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.

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

   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.

   If you get stuck, you can download working versions of `MainPage.xaml` and `MainPage.Xaml.cs` or copy and paste XAML or C# code for each individual method:

   http://www.headfirstlabs.com/hfcsharp
Run the program to see your progress

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

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.

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

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.

1. 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.

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

```xml
<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:

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

2. Fill in the C# code:

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

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.
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:

```csharp
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;
    }
}
```

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

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:

```csharp
private void playArea_PointerMoved(object sender, PointerRoutedEventArgs e)
{
    if (humanCaptured)
    {
        Point pointerPosition = e.GetCurrentPoint(null).Position;
        Point relativePosition = grid.TransformToVisual(playArea).TransformPoint(pointerPosition);
            || (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);
    }
}
```

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

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

```csharp
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);
}
```

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:

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

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

```csharp
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

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

1. 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" />`

2. 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:

3. Use the button in the Transforms section of the Properties window to add a Skew transform:

4. 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:

   ![Avoid These](image)
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.

כתובת: http://www.headfirstlabs.com/hfcsharp

<Template 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>
    ... (more XAML code)
  </Grid>
</Template>

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!
**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:

1. **Open a Windows Store developer account.**

2. 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.

3. Test your app using the Windows App Certification Kit to identify and fix any problems.

4. **Submit your app to the Store!** Once it’s accepted, millions of people around the world can find and download it.

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.

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.**
- **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:

![Visual Studio Remote Debugging Monitor](image)

- 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](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.

1. **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 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:

![Remote Machine drop-down](image)

Don’t forget to change this back to Simulator when you’re ready to move on to the next chapter! You’ll be writing a bunch of programs, and you’ll need this button to run them.

2. **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:

![Remote Debugger Connections window](image)

If you need to change the machine in the future, you can do it in the project settings. Right-click on the project name in the Solution Explorer and choose Properties, then choose the tab. If you clear the field and restart the remote debugger, the Remote Debugger Connections window will pop up again.
3. **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!).

4. **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.

5. **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!

---

**Invasion force, full retreat! That's an order! These Earthlings are no pushovers. We'll need to regroup and replan our attack.**

**Congratulations! You've held off the alien invasion...for now. But we have a feeling that this isn't the last we've heard from them.**
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.

One of these days I’ll figure out what’s going on under there…
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…

1. **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.

   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.

2. **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.

3. **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.

1. **The IDE creates the files and folders for the project.**

   - Save The Humans.csproj
   - MainPage.xaml
   - MainPage.xaml.cs
   - SplashScreen.scale-100.png
   - Properties

2. **The IDE adds code to **MainPage.xaml** that adds a button, and then adds a method to **MainPage.xaml.cs** that gets run anytime the button is clicked.**

   ```csharp
   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.
   ```

3. **The IDE opens the **MainPage.xaml** file and updates a line of XAML code.**

   ```xml
   <Button x:Name="startButton"
           Content="Start!"
           HorizontalAlignment="Center"
           VerticalAlignment="Center" 
           Click="startButton_Click"/>
   
   The IDE went into this file...
   ```

   ...and updated this XAML code.

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

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.

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.Controls—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.


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.

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:

![Solution Explorer](image)

**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.

![Code Editor](image)
**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:

```
random.
```

- Equals
- GetHashCode
- GetType
- Next
  - int Random.Next(int minvalue, int maxvalue) (+ 2 overload(s))
  - Returns a random number within a specified range.
  - Exceptions: System.ArgumentOutOfRangeException
- NextBytes
- NextDouble
- ToString

The IDE knows that `random` has methods `Next`, `NextBytes`, `NextDouble`, and four others. If you type `N`, it selects `Next`. Type “(” or space, Tab, or Enter to tell the IDE to fill it in for you. That can be a real timesaver if you’re typing a lot of really long method names.

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

- `Random.Next()`

```
random.Next()
```

- `2 of 3` int Random.Next(int maxValue)
  - Returns a nonnegative random number less than the specified maximum.
  - `maxValue`: The exclusive upper bound of the random number to be generated. `maxValue` must be greater than or equal to zero.

This means that there are 3 different ways that you can call the `Random.Next()` method.

When you use the debugger to run your program inside the IDE, the first thing it does is build your program. If it compiles, then your program runs. If not, it won’t run, and will show you errors in the Error List.

**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:

![Error List](image)

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

```
int j = random.Next(10); ; expected
```

The IDE will show a squiggly underscore to show you that there’s an error. Hover over it to see the same error message that appears in the Error List.
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.

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

A class has one or more methods. Your methods always have to live inside a class. And methods are made up of statements—like the ones you’ve already seen.

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.

1. **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.


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

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:

```csharp
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).
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.

```csharp
namespace Save_the_Humans
{
    public sealed partial class MainPage : Page
    {
        This is a class called MainPage. It contains all of the code to make the page work. The IDE created it when you told it to create a new blank C# Windows Store project.

        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.

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

        This line calls a method named StartGame(), which the IDE helped you create when you asked it to add a method stub.

        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.

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

**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:** 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:** 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.

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.
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!)

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

Set properties for a TextBlock control

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

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

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

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

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

Change the background color of a Grid control named myGrid

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

A method that executes whenever a program displays its main page
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);`
  - Set properties for a TextBlock control

- `// This loop gets executed three times`
  - Nothing—it’s a comment that the programmer added to explain the code to anyone who’s reading it

- `public sealed partial class MainPage : Page`
  - 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.

- `helloLabel.Text = "hi there"; helloLabel.FontSize = 24;`
  - A special kind of comment that the IDE uses to explain what an entire block of code does

- `/// <summary>
/// Bring up the picture of Rover when the button is clicked
/// </summary>`
  - Change the background color of a Grid control named `myGrid`

- `partial class Form1`
  - A method that executes whenever a program displays its main page

- `this(MaximizeBox = false;`
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.

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

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.

**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.

```csharp
int maxWeight;
string message;
bool boxChecked;
```

These names are for **YOU**. Like methods and classes, use names that make sense and describe the variable’s usage.

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

These are the variable types.

These are the names of these variables.

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:

```csharp
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:

```csharp
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:

```csharp
    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:

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

Each declaration has a type, exactly like before.

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.
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:

```csharp
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;
```

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

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

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.

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.

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

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

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.

1. **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.

2. **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.

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!
**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.

**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.

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

**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:

```plaintext
Watch 1
Name     Value     Type
number   0         int
```

**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:

```plaintext
Watch 1
Name     Value     Type
number   15        int
```

As soon as the `number` variable gets a new value (15), its watch is updated.

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

---

**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.

**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.**

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!).

```
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.

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

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.

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

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:

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

```java
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.

`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.

```
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.";
}
```

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

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.
Build an app from the ground up

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.

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`.

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.

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.

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

When you see these sneakers, it means that it’s time for you to come up with code on your own.
Chapter 2

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!

Here are the row and column definitions: three rows and two columns.

This button is in the second column and second row, so these properties are set to 1.

When you double-clicked on each button, the IDE generated a method with the name of the button followed by _Click.

This button is in the second column and second row, so these properties are set to 1.

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?

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!

**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:

```csharp
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 + "
x is " + x
    + "\n\nd is " + d;
}
```

Here’s the code for the button:

```csharp
int j     =      1234  ;
```  

is the same as:

```csharp
int j = 1234;
```

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.

```
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.
```

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

**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:

```csharp
name is Quentin
x is 51
d is 1.5707963267949
```

Here’s the code for the button:

```csharp
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;
}
```

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

**A few helpful tips**

- Don’t forget that all your statements need to end in a semicolon:
  ```csharp
  name = "Joe";
  ```
- You can add comments to your code by starting them with two slashes:
  ```csharp
  // 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):
  ```csharp
  int weight;
  // weight is an integer
  ```
- The code for a class or a method goes between curly braces:
  ```csharp
  public void Go() {
    // your code here
  }
  ```
- Most of the time, extra whitespace is fine:
  ```csharp
  int j = 1234;
  ```
  ```csharp
  int j     =      1234  ;
  ```
  is the same as:
  ```csharp
  int j = 1234;
  ```
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).

2 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.

```csharp
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";
    }
}
```

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

```csharp
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";
}
```

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!

4 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.

```csharp
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;
}
```

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!
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.

```csharp
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;
```
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!

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

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

**Loop #3**
```java
int j = 2;
for (int i = 1; i < 100; i = i * 2) {
    j = j - 1;
    while (j < 25) {
        j = j + 5;
    }
}
```

**Loop #4**
```java
while (true) { int i = 1;}
```

**Loop #5**
```java
int p = 2;
for (int q = 2; q < 32; q = q * 2) {
    while (p < q) {
        p = p * 2;
    }
    q = p - q;
}
```

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

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

```csharp
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;
```

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 #2**
This loop runs forever

**Loop #3**
This loop executes 7 times

**Loop #4**
Another infinite loop

**Loop #5**
This loop executes 8 times

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.
Q: Is every statement always in a class?
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: Are there any namespaces I’m not allowed to use? Are there any I have to use?
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: I still don’t get why I need this partial class stuff.
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: 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: 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: So exactly how careful do I have to be with the code that’s automatically generated by the IDE?
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.

```csharp
int x = 3;
while (x > 0)
{
    if (x > 2) {
        result = result + "a";
    }
    x = x - 1;
    result = result + "-";
    if (x == 2) {
        result = result + "b c";
    }
    if (x == 1) {
        result = result + "d";
        x = x - 1;
    }
    if (x <= 1) {
        result = result + "d";
        x = x - 1;
    }
}
output.Text = result;
```

Output:

```
a-b c-d
```

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

Answers on page 86.
Time to get some practice using if/else statements. Can you build this program?

**Exercise**

**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).

---

**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:

```
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.

Output

a noise annoys an oyster

Here's another TextBlock, and we also gave it the name "output".

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.

Note: each snippet from the pool can only be used once!

```java
int x = 0;
string poem = "";
while ( __________ ) {
    _______________________
    if ( x < 1 ) {
        _______________________
    }
    _______________________
    if ( __________ ) {
        _______________________
    }
    _______________________
    if ( x == 1 ) {
        _______________________
    }
    _______________________
    if ( ___________ ) {
        _______________________
    }
    _______________________
} _______________________
output.Text = poem;
```
Time to get some practice using if/else statements. Can you build this program?

Here's the XAML code for the grid:
```
<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>
```

And here's the C# code for the button's event handler method:
```csharp
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**

Introducing a different kind of app

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

```csharp
string result = "";
int x = 3;
while (x > 0)
{
    if (x > 2) {
        result = result + "a";
    }
    x = x - 1;
    result = result + "-";
    if (x == 2) {
        result = result + "b c";
    }
    if (x == 1) {
        result = result + "d";
        x = x - 1;
    }
}
output.Text = result;
```

The first time through the loop, x is equal to 3, so this conditional test will be true.

This statement makes x equal to 2 the first time through the loop, and 1 the second time through.

---

**Pool Puzzle**

Solution

```csharp
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.

---

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.
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.

When you create a new project in Visual Studio 2013 Express for Windows Desktop, you get these options. Choose Windows Forms Application.

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.

Normally you should choose a better name than “Chapter 2 - Program 4,” but we’re specifically using a name with spaces and a hyphen for this project so you can see what it does to the namespace.

Keep dragging these handles until your form is the right size.

Here’s what your form should look like after you resize it.
**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.

**Add a Button, CheckBox, and Label.**
Open up the toolbox and drag a Button, CheckBox, and Label control onto your form.

---

**Toolbox**
You can expand the toolbox by choosing “Toolbox” from the View menu, or by clicking on the Toolbox tab on the side of the IDE. You can keep it from disappearing by clicking the pushpin icon (¶) on the Toolbox window. You can also drag the window title so that it floats over the IDE. These spacer lines help you position your controls as you drag them around.

On the next page you’ll use the Properties window to change the text on each control, and to set the CheckBox control’s state to checked. See if you can figure out how to do that before you flip the page!

**Hint:** you’ll need to use the AutoSize property to get the Label control to look right.

---

**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.
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.

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.

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?

```csharp
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");
        }
    }
}
```

Here’s a hint. You haven’t seen MessageBox yet, but it’s something that a lot of desktop apps use. Like most classes and methods, it has a sensible name.
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.

Here’s some code the IDE built for you automatically in the last chapter. You’ll find it in Program.cs.

```csharp
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());
        }
    }
}
```

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.

I do declare!
The first part of every class or method is called a declaration.
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.

**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.

**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.

**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.

**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.

**Each desktop app has a special kind of method called the entry point.**
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.

1. 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?

2. 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:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Chapter_2___Program_4
{
    class AnotherClass
    {
    }
}
```

3. 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!

4. Add this method to the AnotherClass class by typing it in between the curly brackets:

```csharp
class AnotherClass
{
    public static void Main()
    {
        MessageBox.Show("Pow!");
    }
}
```

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.

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.

Figure out how to fix your program so it pops up the app again.

Hint: you only have to change two lines in two files to do it.

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");
        }
    }
}
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.

1. **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.

2. **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.

   Select “Local resource” and click the Import... button to pop up a dialog to find the image file to import.

   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.

3. **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:

   ```
   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;
   ```
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:

```csharp
/// <summary>
/// Required method for Designer support - do not modify the contents of this method with the code editor.
/// </summary>
```  

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:**

1. **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?

2. **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?

3. **Change the code that sets the LOCATION property to (0,0) and the size property to make the button really big.**
   Did it work?

4. **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?

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.

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

A: Your program has a whole lot of statements in it, but they’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().
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.

**Exercise**

1. **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.

2. **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:

   ```csharp
   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.

   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!

3. **Make it slower.**
   Slow down the flashing by adding this line after the `Application.DoEvents()` line:

   ```csharp
   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.

   **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.
**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.

**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!)

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).

**MAKE IT STOP.**
Make the loop you added in step #5 stop when the program is closed. Change your outer loop to this:

```
while (Visible)
```

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.

When you’re checking a Boolean value like `Visible` in an `if` statement or a loop, sometimes it’s tempting to test for (`Visible == true`). You can leave off the `== true`—it’s enough to include the Boolean.

**Can you figure out what’s causing that delay? Can you fix it so the program ends immediately when you close the window?**
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);
        }
    }
}

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!
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!

1. Mike figures out the route he’ll take to get to the interview.

2. Good thing he had his radio on. There’s a huge traffic jam that’ll make him late!

3. Mike comes up with a new route to get to his interview on time.
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.

```csharp
SetDestination("Fifth Ave & Penn Ave");
string route;
route = GetRoute();
```

The navigation system sets a destination and comes up with a route.

Here's a diagram of a class in Mike's program. It shows the name on top, and the methods on the bottom.

<table>
<thead>
<tr>
<th>Navigator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetCurrentLocation()</td>
</tr>
<tr>
<td>SetDestination()</td>
</tr>
<tr>
<td>ModifyRouteToAvoid()</td>
</tr>
<tr>
<td>ModifyRouteToInclude()</td>
</tr>
<tr>
<td>GetRoute()</td>
</tr>
<tr>
<td>GetTimeToDestination()</td>
</tr>
<tr>
<td>TotalDistance()</td>
</tr>
</tbody>
</table>

"Take 31st Street Bridge to Liberty Avenue to Bloomfield"

Now it can come up with a new route to the destination.

```csharp
ModifyRouteToAvoid("Liberty Ave");
```

The navigation system gets new information about a street it needs to avoid.

"Take Route 28 to the Highland Park Bridge to Washington Blvd"

GetRoute() gives a new route that doesn't include the street Mike wants to avoid.

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.

```csharp
class Navigator {
    public void SetCurrentLocation(string locationName) { ... }
    public void SetDestination(string destinationName) { ... }
    public void ModifyRouteToAvoid(string streetName) { ... }
    public string 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.

```csharp
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:

```csharp
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.

1. 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.

2. Add using System.Windows.Forms; to the top of the class file. Then add code to the class:

   ```csharp
   class Talker {
       public static int BlahBlahBlahBlah(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.

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.

Flip the page to keep going!
introducing objects

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!

3 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:

```csharp
private void button1_Click(object sender, EventArgs e)
{
    int len = Talker.BlahBlahBlah(textBox1.Text, (int)numericUpDown1.Value);
    MessageBox.Show("The message length is " + len);
}
```

4 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.

<table>
<thead>
<tr>
<th>Navigator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetCurrentLocation()</td>
</tr>
<tr>
<td>SetDestination()</td>
</tr>
<tr>
<td>ModifyRouteToAvoid()</td>
</tr>
<tr>
<td>ModifyRouteToInclude()</td>
</tr>
<tr>
<td>GetRoute()</td>
</tr>
<tr>
<td>GetTimeToDestination()</td>
</tr>
<tr>
<td>TotalDistance()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Navigator2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetCurrentLocation()</td>
</tr>
<tr>
<td>SetDestination()</td>
</tr>
<tr>
<td>ModifyRouteToAvoid()</td>
</tr>
<tr>
<td>ModifyRouteToInclude()</td>
</tr>
<tr>
<td>GetRoute()</td>
</tr>
<tr>
<td>GetTimeToDestination()</td>
</tr>
<tr>
<td>TotalDistance()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Navigator3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetCurrentLocation()</td>
</tr>
<tr>
<td>SetDestination()</td>
</tr>
<tr>
<td>ModifyRouteToAvoid()</td>
</tr>
<tr>
<td>ModifyRouteToInclude()</td>
</tr>
<tr>
<td>GetRoute()</td>
</tr>
<tr>
<td>GetTimeToDestination()</td>
</tr>
<tr>
<td>TotalDistance()</td>
</tr>
</tbody>
</table>

Whoa, that can’t be right! What if I want to change a method? Then I need to go back and fix it in three places.

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.

<table>
<thead>
<tr>
<th>Navigator</th>
<th>new Navigator()</th>
<th>new Navigator()</th>
<th>new Navigator()</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetCurrentLocation()</td>
<td>navigator1</td>
<td>navigator2</td>
<td>navigator3</td>
</tr>
<tr>
<td>SetDestination()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ModifyRouteToAvoid()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ModifyRouteToInclude()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetRoute()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetTimeToDestination()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TotalDistance()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All you need to create an object is the **new** keyword and the name of a class.

```csharp
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.

When you define a class, you define its methods, just like a blueprint defines the layout of the house.

You can use one blueprint to make any number of houses, and you can use one class to make any number of objects.

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.

<table>
<thead>
<tr>
<th>House</th>
</tr>
</thead>
<tbody>
<tr>
<td>GiveShelter()</td>
</tr>
<tr>
<td>GrowLawn()</td>
</tr>
<tr>
<td>MailDelivered()</td>
</tr>
<tr>
<td>ClogDrainPipes()</td>
</tr>
<tr>
<td>AccruePropertyTaxes()</td>
</tr>
<tr>
<td>NeedRepairs()</td>
</tr>
</tbody>
</table>

Object: 115 Maple Drive

Object: 26A Elm Lane

Object: 38 Pine Street
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!

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.

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.

1. 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.

2. He created a **Navigator** object and set its destination.

   ```csharp
   string destination = textBox1.Text;
   Navigator navigator1 = new Navigator();
   navigator1.SetDestination(destination);
   route = navigator1.GetRoute();
   ```

3. 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.

4. The third **Navigator** object is called **navigator3**. Mike set its destination, and then called its ModifyRouteToInclude() method.

5. 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!
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();

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).

★ If your solution just won’t build, try downloading it from the Head First Labs website: http://www.headfirstlabs.com/hfsharp

When you run into a problem with a coding exercise, don’t be afraid to peek at the solution. You can also download the solution from the Head First Labs website.
string destination = textBox1.Text;
string route2StreetToAvoid = textBox2.Text;
string route3StreetToInclude = textBox3.Text;

Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
int distance1 = navigator1.TotalDistance();

1. Create the navigator2 object, set its destination, call its `ModifyRouteToAvoid()` method, and use its `TotalDistance()` method to set an integer variable called `distance2`.

   ```csharp
   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`.

   ```csharp
   int 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.

```csharp
int shortestDistance = Math.Min(distance1, Math.Min(distance2, distance3));
```
string destination =textBox1.Text;
string route2StreetToAvoid =textBox2.Text;
string route3StreetToInclude =textBox3.Text;
Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
int distance1 = navigator1.TotalDistance();

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.

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:

```csharp
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.

---

**Q:** When I think of something that’s “static,” I think of something that doesn’t change. Does that mean nonstatic methods can change, but static methods don’t? Do they behave differently?

**A:** No, both static and nonstatic methods act exactly the same. The only difference is that static methods don’t require an instance, while nonstatic methods do. A lot of people have trouble remembering that, because the word “static” isn’t really all that intuitive.

**Q:** So I can’t use my class until I create an instance of an object?

**A:** You can use its static methods. But if you have methods that aren’t static, then you need an instance before you can use them.

**Q:** Then why would I want a method that needs an instance? Why wouldn’t I make all my methods static?

**A:** Because if you have an object that’s keeping track of certain data—like 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:** So how does an instance keep track of data?

**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:

```csharp
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.

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.

```csharp
class Clown {
    public string Name;
    public int Height;

    public void TalkAboutYourself() {
        MessageBox.Show("My name is " + Name + " and I’m " + Height + " inches tall.");
    }
}
```

Write down the contents of each message box that will be displayed after the statement next to it is executed.

Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;

```csharp
oneClown.TalkAboutYourself();
```

"My name is _______ and I’m ______ inches tall."

Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;

```csharp
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.

> 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:

1. Clown oneClown = new Clown();
   oneClown.Name = "Boffo";
   oneClown.Height = 14;
   oneClown.TalkAboutYourself();

2. Clown anotherClown = new Clown();
   anotherClown.Name = "Biff";
   anotherClown.Height = 16;
   anotherClown.TalkAboutYourself();

3. Clown clown3 = new Clown();
   clown3.Name = anotherClown.Name;
   clown3.Height = oneClown.Height - 3;
   clown3.TalkAboutYourself();

4. anotherClown.Height *= 2;
   anotherClown.TalkAboutYourself();
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.

Here’s a nice, compact chunk of code. It’s from a control program that runs a machine that makes candy bars.

```java
int t = m.chkTemp();
if (t > 160) {
    T tb = new T();
    tb.clsTrpV(2);
    ics.Fill();
    ics.Vent();
    m.airsyschk();
}
```

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.

The `chkTemp()` method returns an integer…but what does it do?

The `clsTrpV()` method has one parameter, but we don’t know what it’s supposed to be.

### 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.
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.

```java
public boolean IsNougatTooHot() {
    int temp = Maker.CheckNougatTemperature();
    if (temp > 160) {
        return true;
    } else {
        return false;
    }
}
```

This method’s return type is `Boolean`, which means it returns a true or false value.

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):

```java
public void DoCICSVentProcedure() {
    Turbine turbineController = new Turbine();
    turbineController.CloseTripValve(2);
    IsolationCoolingSystem.Fill();
    IsolationCoolingSystem.Vent();
    Maker.CheckAirSystem();
}
```

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:

```java
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?
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.

**Turbine**

We filled in the class name for this one. What method goes here?

One of the classes had a method called Fill(). Fill in its class name and its other method.

**Fill()**

There was one other class in the code on the previous page. Fill in its name and method.
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.

The class is called Dishwasher, so all the methods should be about washing dishes. But one method—ParkTheCar()—has nothing to do with dishes, so it should be taken out and put in another class.

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.
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.

<table>
<thead>
<tr>
<th>Class23</th>
<th>DeliveryGuy</th>
<th>DeliveryGirl</th>
</tr>
</thead>
<tbody>
<tr>
<td>CandyBarWeight()</td>
<td>AddAPizza()</td>
<td>AddAPizza()</td>
</tr>
<tr>
<td>PrintWrapper()</td>
<td>PizzaDelivered()</td>
<td>PizzaDelivered()</td>
</tr>
<tr>
<td>GenerateReport()</td>
<td>TotalCash()</td>
<td>TotalCash()</td>
</tr>
<tr>
<td>Go()</td>
<td>ReturnTime()</td>
<td>ReturnTime()</td>
</tr>
</tbody>
</table>

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

These two classes are part of a system that a pizza parlor uses to track the pizzas that are out for delivery.

The CashRegister class is part of a program that's used by an automated convenience store checkout system.
Here’s how we corrected the classes. We show just one possible way to fix the problems—but there are plenty of other ways you could design these classes depending on how they’ll be used.

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

The class name doesn’t describe what the class does. A programmer who sees a line of code that calls Class23.Go() will have no idea what that line does. We’d also rename the method to something that’s more descriptive—we chose MakeTheCandy(), but it could be anything.

<table>
<thead>
<tr>
<th>CandyMaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>CandyBarWeight()</td>
</tr>
<tr>
<td>PrintWrapper()</td>
</tr>
<tr>
<td>GenerateReport()</td>
</tr>
<tr>
<td>MakeTheCandy()</td>
</tr>
</tbody>
</table>

These two classes are part of a system that a pizza parlor uses to track the pizzas that are out for delivery.

It looks like the DeliveryGuy class and the DeliveryGirl class both do the same thing—they track a delivery person who’s out delivering pizzas to customers. A better design would replace them with a single class that adds a field for gender.

We added the Gender field because we assumed there was a reason to track delivery guys and girls separately, and that’s why there were two classes for them.

<table>
<thead>
<tr>
<th>DeliveryPerson</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gender</td>
</tr>
<tr>
<td>AddAPizza()</td>
</tr>
<tr>
<td>PizzaDelivered()</td>
</tr>
<tr>
<td>TotalCash()</td>
</tr>
<tr>
<td>ReturnTime()</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>CashRegister</th>
</tr>
</thead>
<tbody>
<tr>
<td>MakeSale()</td>
</tr>
<tr>
<td>NoSale()</td>
</tr>
<tr>
<td>Refund()</td>
</tr>
<tr>
<td>TotalCashInRegister()</td>
</tr>
<tr>
<td>GetTransactionList()</td>
</tr>
<tr>
<td>AddCash()</td>
</tr>
<tr>
<td>RemoveCash()</td>
</tr>
</tbody>
</table>
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 (false) {
            result = result + e1.Hello() + "\n";
            if (false) {
                e2.Count = e2.Count + 1;
            }
            if (false) {
                e2.Count = e2.Count + e1.Count;
            }
            x = x + 1;
        }
        MessageBox.Show(result + "Count: " + e2.Count);
    }
}
class ____________ {
    public int _________ = 0;
    public string ___________ {
        return "helloooo...";
    }
}

Output

10
helloooo...
helloooo...
helloooo...
Count: 10

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!

x < 4
x < 5
x > 0
x > 1
e1 = e1 + 1;
e1 = Count + 1;
e1.Count = Count + 1;
e1.Count = e1.Count + 1;

Echo
Tester
Echo()
Count()
Hello()
e2 = e1;
e2 = e2;
Echo e2 = e1;
Echo e2 = new Echo();

Answers on page 138.
Build a class to work with some guys

Joe and Bob lend each other money all the time. Let’s create a class to keep track of them. We’ll start with an overview of what we’ll build.

1. **We’ll create a Guy class and add two instances of it to a form.**
   The form will have two fields, one called bob (to keep track of the second object), and the other called joe (to keep track of the first object).

2. **We’ll set each Guy object’s cash and name fields.**
   The two objects represent different guys, each with his own name and a different amount of cash in his pocket.

3. **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.

---

**Guy**

<table>
<thead>
<tr>
<th>Name</th>
<th>Cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>100</td>
</tr>
<tr>
<td>Bob</td>
<td>50</td>
</tr>
</tbody>
</table>

The form calls the object’s ReceiveCash() method. It’s called ReceiveCash() because he’s receiving the cash.

The method returns the number of bucks that the guy added to his Cash field.
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:

```csharp
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;
        }
    }

    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 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.

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. 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.

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.

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.

1. **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”.

   ![Image of a form with labels and buttons](image)

   This button will call the Joe object’s `ReceiveCash()` method, passing it 10 as the amount, and subtracting from the form’s bank field the cash that Joe receives.

   Name the top label `joesCashLabel`, the label underneath it `bobsCashLabel`, and the bottom label `bankCashLabel`. You can leave their Text properties alone; we’ll add a method to the form to set them.

   This button will call the Bob object’s `GiveCash()` method, passing it 5 as the amount, and adding the cash that Bob gives to the form’s bank field.

2. **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.

   ```csharp
   namespace Your_Project_Name {
       public partial class Form1 : Form {
           public Form1() {
               InitializeComponent();
               
               Guy joe;
               Guy bob;
               int bank = 100;
               
           }
       }
   }
   ```

   The amount of cash in the form’s bank field goes up and down depending on how much money the form gave to and received from the guys.

   ![Image of code snippet](image)

   Since we’re using `Guy` objects to keep track of Joe and Bob, you declare their fields in the form using the `Guy` class.

   ```csharp
   Joe says, “where’s my money?”
   ```
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:

```csharp
public void UpdateForm() {
    joesCashLabel.Text = joe.Name + " has $" + joe.Cash;
    bobsCashLabel.Text = bob.Name + " has $" + bob.Cash;
    bankCashLabel.Text = "The bank has $" + bank;
}
```

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.

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:

```csharp
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.

```csharp
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.

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.

```csharp
public Form1() {
    InitializeComponent();
    // Initialize joe and bob here!
}
```
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.

```csharp
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.

Make sure you call `UpdateForm()` so the labels look right when the form first pops up.

There are no Dumb Questions

Q: Why doesn’t the solution start with “`Guy bob = new Guy()`”? Why did you leave off the first “Guy”?

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: OK, so then why not get rid of the “`Guy bob;`” line at the top of the form?

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: 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: 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:

```csharp
public partial class Form1 : Form {
    Guy bob;
}
```

and then you have this code later on, inside a method:

```csharp
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.

1. Here’s the original code that you wrote to initialize Joe’s Guy object.
   ```csharp
   joe = new Guy();
   joe.Name = "Joe";
   joe.Cash = 50;
   ```

2. Delete the second two lines and the semicolon after “Guy();,” and add a right curly bracket.
   ```csharp
   joe = new Guy() {
   ```

3. 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.
   ```csharp
   joe = new Guy() {
       Cash = 50,
   };
   ```

4. Press Tab to tell it to add the Cash field. Then set it equal to 50.
   ```csharp
   joe = new Guy() { Cash = 50
   ```

5. Type in a comma. As soon as you do, the other field shows up.
   ```csharp
   joe = new Guy() { Cash = 50,
       Name = "Joe"
   };
   ```

6. Finish the object initializer. Now you’ve saved yourself two lines of code!
   ```csharp
   joe = new Guy() { Cash = 50, Name = "Joe" }
   ```

Object initializers save you time and make your code more compact and easier to read...and the IDE helps you write them.

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.
Add buttons to the “Fun with Joe and Bob” program to make the guys give each other cash.

**Exercise**

1. **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.

   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.

2. **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`.

   ![Image of button names](image)

3. **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.

   ![Image of button functionality](image)

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.
Add buttons to the “Fun with Joe and Bob” program to make the guys give each other cash.

```csharp
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.

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

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

Down

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();
        _____________
        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...";
    }
}

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
Across
1. This form control lets the user choose a number from a range you set. [numericupdown]
3. It’s a great idea to create a class ________ on paper before you start writing code [diagram]
4. What an object uses to keep track of what it knows [field]
5. These define what an object does [methods]
6. An object’s methods define its ________ [behavior]
7. Don’t use this keyword in your class declaration if you want to be able to create instances of it [static]
8. An object is an ______________ of a class [instance]
9. An object’s fields define its _______ [state]
10. Where objects live [heap]
11. What you use to build an object [class]
12. This statement tells a method to immediately exit, and specifies the value that should be passed back to the statement that called the method. [return]
13. What you use to pass information into a method [parameters]
14. The statement you use to create an object [new]
15. A special kind of field that’s used by the form controls [property]

Down
1. This form control lets the user choose a number from a range you set. [numericupdown]
2. If a method’s return type is _____, it doesn’t return anything. [void]
4. What an object uses to keep track of what it knows [field]
5. These define what an object does [methods]
6. An object’s methods define its ________ [behavior]
7. An object’s fields define its _______ [state]
9. A good method __________ makes it clear what the method does. [name]
10. Where objects live [heap]
11. What you use to build an object [class]
12. This statement tells a method to immediately exit, and specifies the value that should be passed back to the statement that called the method. [return]
Thanks for reading the first three chapters of our book. We hope it gave you a nice preview... because we know you're going to have a great time learning C#.
The fun's just beginning!

Get C# programming into your brain... fast!

*Head First C#* is a complete learning experience for programming with C#, XAML, the .NET Framework, and Visual Studio. **Built for your brain**, this book keeps you engaged from the first chapter. You'll learn about classes and object-oriented programming, draw graphics and animation, query your data with LINQ, and serialize it to files. And you'll do it all by building **games**, solving **puzzles**, and doing **hands-on projects**. By the time you're done you'll be a solid C# programmer, and you'll have a great time along the way!
You'll learn all about objects and references, and how they help make your data make sense in the real world.

“If you want to learn C# in depth and have fun doing it, this is THE book for you.”
—Andy Parker, fledgling C# programmer
You'll put object oriented programming theory into practice and get it into your brain fast by building games, doing projects, and solving puzzles.
Effective programming means getting a handle on your data. You'll learn to model your data, manage it in memory, write it to files, and get into the bits and bytes.

"Head First C# 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
Harness the power of XAML to build sleek, modern apps. You'll learn how to create a modern user interface with graphics, animation, and more.

In the UI and XAML world, animations can refer to any property that changes over a specific time period. But in the real world, it means drawing that never changes. So, let's build a simple program to do some "real" animation.

Create a project and add the pictures

Let's get started with the project. Create a new Windows Store project called Animations. Then add each one of the assets shown. You'll need to create Tile, Title, and TouchAnimation.

Download the images for this chapter from the Head First Labs site:

www.headfirstlabs.com/books/htsharp/

Keep an open mind about animation. With careful planning, even simple animations can be very effective.

Use semantic zoom to navigate your data

It's great to give users an overview of a collection, but you've got a way to make it even better. The semantic zoom, a scalable vector that you can zoom in on two different views of a zoomable view, shows a zoomed out view that shows more detail, and a zoomed-in view that shows more detail.

Semantic zoom allows you to display two different views of the same sequence of data: a zoomed-out view that shows many items, and a zoomed-in view that shows more detail.

Join purchases...
Advanced concepts like MVVM and design patterns are made simple with examples, projects, and straightforward explanations.
You'll build full-featured, exciting video games! We'll keep your brain engaged, and give you the practice that you need to become a solid C# programmer.