

A Brain-Friendly Guide

Head First

C#

2nd
Edition
Covers C# & .Net 4.0,
and Visual Studio 2010



Boss your
data around
with LINQ

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

Build a fully
functional
retro classic
arcade game



Discover the
secrets of
abstraction and
inheritance



Learn how
extension
methods helped
Sue bend the
rules in Objectville



See how Jim used
generic collections to
wrangle his data

O'REILLY®

Andrew Stellman
& Jennifer Greene

Head First C#

Programming Languages/Microsoft C#/.NET

What will you learn from this book?

Head First C# is a complete learning experience for programming with C#, the .NET Framework, and the Visual Studio IDE. Built for your brain, this book covers C# & .NET 4.0 and Visual Studio 2010, and teaches everything from inheritance to serialization. You'll query your data with LINQ, draw graphics and animation, and learn all about classes and object-oriented programming, all through building games, doing hands-on projects, and solving puzzles. You'll become a solid C# programmer, and you'll have a great time along the way!

Understand the difference between classes and objects.

Exercise your C# skills by building an invaders game...
...and creating a role-playing game with deadly enemies.

Learn how to get the IDE to do your grunt work for you.

Create a beehive simulation program using double-buffered animation.

Master the principles of object-oriented programming.

Inheritance
Encapsulation
Abstraction
Polymorphism

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 Parker,
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#* is a highly enjoyable tutorial, full of memorable examples and entertaining exercises.”

—Joseph Albahari,
C# Design Architect at Egton Medical Information Systems, the UK's largest primary healthcare software supplier, co-author of C# 4.0 in a Nutshell

US \$49.99

CAN \$62.99

ISBN: 978-1-449-38034-2



9

Safari 
Books Online

Free online edition
for 45 days with
purchase of this book.
Details on last page.

O'REILLY[®]
oreilly.com
headfirstlabs.com

Advance Praise for *Head First C#*

“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 walk the reader through some of the more challenging concepts of C# in a simple, easy-to-follow way. At the end of some of the larger projects/labs, the reader can look back at their programs and stand in awe of what they’ve accomplished.”

— **David Sterling, 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

“Kathy and Bert’s *Head First Java* transforms the printed page into the closest thing to a GUI you’ve ever seen. In a wry, hip manner, the authors make learning Java an engaging ‘what’re they gonna do next?’ experience.”

—**Warren Keuffel, *Software Development Magazine***

“Beyond the engaging style that drags you forward from know-nothing into exalted Java warrior status, *Head First Java* covers a huge amount of practical matters that other texts leave as the dreaded “exercise for the reader...” It’s clever, wry, hip and practical—there aren’t a lot of textbooks that can make that claim and live up to it while also teaching you about object serialization and network launch protocols. ”

—**Dr. Dan Russell, Director of User Sciences and Experience Research
IBM Almaden Research Center (and teaches Artificial Intelligence at
Stanford University)**

“It’s fast, irreverent, fun, and engaging. Be careful—you might actually learn something!”

—**Ken Arnold, former Senior Engineer at Sun Microsystems
Co-author (with James Gosling, creator of Java),
*The Java Programming Language***

“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 très ‘cool.’ It is fun, but they cover a lot of ground and they are right to the point. I’m really impressed.”

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

“One of the funniest and smartest books on software design I’ve ever read.”

— **Aaron LaBerge, VP Technology, ESPN.com**

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

— **Mike Davidson, CEO, Newsvine, Inc.**

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

— **Ken Goldstein, Executive Vice President, Disney Online**

“I ♥ Head First HTML with CSS & XHTML—it teaches you everything you need to learn in a ‘fun coated’ format.”

— **Sally Applin, UI Designer and Artist**

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

“While other books on design patterns are saying ‘Bueller... Bueller... Bueller...’ this book is on the float belting out ‘Shake it up, baby!’”

— **Eric Wuehler**

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

— **Satish Kumar**

Other related books from O'Reilly

Programming C# 4.0

C# 4.0 in a Nutshell

C# Essentials

C# Language Pocket Reference

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

Head First 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

Head First PHP & MySQL

Head First Algebra

Head First Data Analysis

Head First Excel

Head First C#

Second Edition

Wouldn't it be dreamy
if there was a *C#* book that
was more fun than endlessly
debugging code? It's probably
nothing but a fantasy....



Andrew Stellman
Jennifer Greene

O'REILLY®

Beijing • Cambridge • Köln • Sebastopol • Taipei • Tokyo

Head First C#

Second Edition

by Andrew Stellman and Jennifer Greene

Copyright © 2010 Andrew Stellman and Jennifer Greene. All rights reserved.

Printed in the United States of America.

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

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

Series Creators:

Kathy Sierra, Bert Bates

Cover Designers:

Louise Barr, Karen Montgomery

Production Editor:

Rachel Monaghan

Proofreader:

Emily Quill

Indexer:

Lucie Haskins

Page Viewers:

Quentin the whippet and Tequila the pomeranian

Printing History:

November 2007: First Edition.

May 2010: Second Edition.



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

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

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

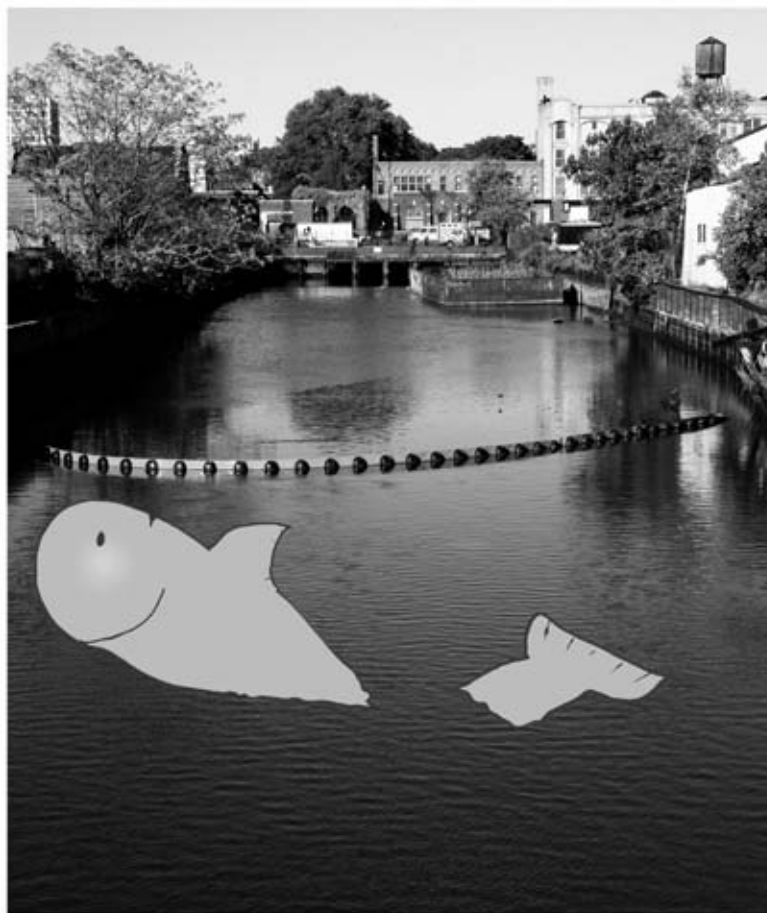
While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

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

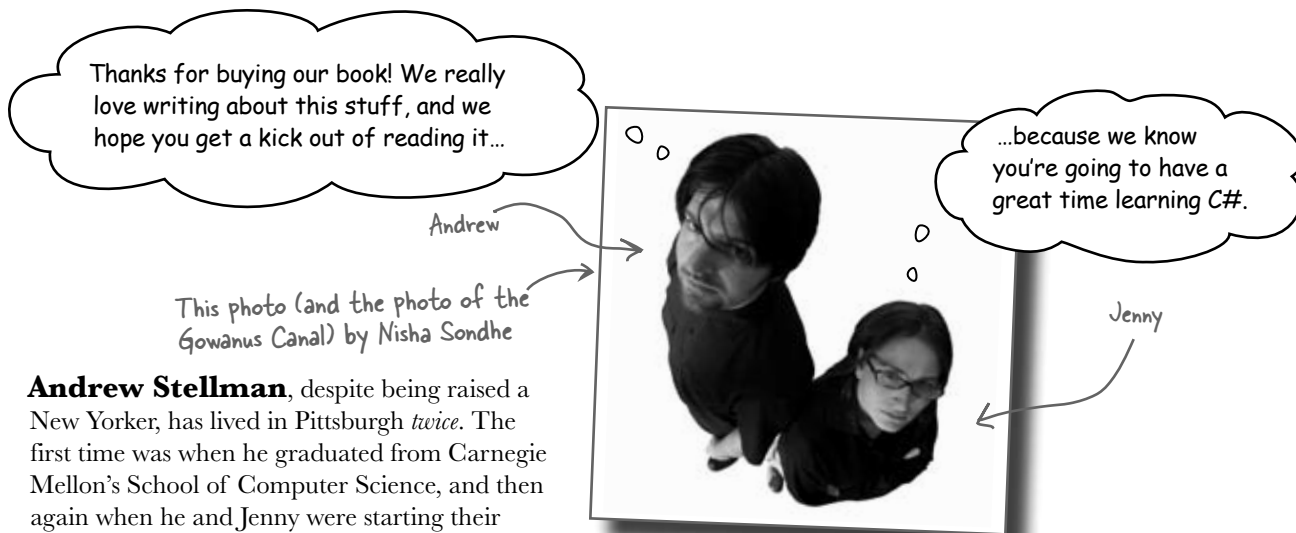
ISBN: 978-1-449-38034-2

[SB]

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.*



Andrew Stellman, despite being raised a New Yorker, has lived in 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.

When he moved back to his hometown, his first job after college was as a programmer at EMI-Capitol Records—which actually made sense, since he went to LaGuardia High School of Music and Art and the Performing Arts to study cello and jazz bass guitar. He and Jenny first worked together at that same financial software company, where he was managing a team of programmers. He's had the privilege of working with some pretty amazing programmers over the years, 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 music (but video games even more), experimenting with circuits that make odd noises, studying taiji and aikido, having a girlfriend named Lisa, 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. They published their first book in the Head First series, *Head First PMP*, in 2007.

They founded Stellman & Greene Consulting in 2003 to build a really neat software project for scientists studying herbicide exposure in Vietnam vets. When they're not building software or writing books, they do a lot of speaking at conferences and meetings of software engineers, architects and project managers.

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

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 managed a team of testers at a really cool startup that did artificial intelligence and natural language processing.

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 (especially *LittleBigPlanet!*), and owning a whippet.

Table of Contents (Summary)

	Intro	xxix
1	Get productive with C#: <i>Visual Applications, in 10 minutes or less</i>	1
2	It's All Just Code: <i>Under the hood</i>	41
3	Objects: Get Oriented: <i>Making code make sense</i>	85
4	Types and References: <i>It's 10:00. Do you know where your data is?</i>	125
	C# Lab 1: <i>A Day at the races</i>	169
5	Encapsulation: <i>Keep your privates... private</i>	179
6	Inheritance: <i>Your object's family tree</i>	215
7	Interfaces and abstract classes: <i>Making classes keep their promises</i>	269
8	Enums and collections: <i>Storing lots of data</i>	327
	C# Lab 2: <i>The Quest</i>	385
9	Reading and Writing Files: <i>Save the byte array, save the world</i>	407
10	Exception Handling: <i>Putting out fires gets old</i>	463
11	Events and Delegates: <i>What your code does when you're not looking</i>	507
12	Review and Preview: <i>Knowledge, power, and building cool stuff</i>	541
13	Controls and Graphics: <i>Make it pretty</i>	589
14	Captain Amazing: <i>The Death of the Object</i>	647
15	LINQ: <i>Get control of your data</i>	685
	C# Lab 3: <i>Invaders</i>	713
i	Leftovers: <i>The top 11 things we wanted to include in this book</i>	735

Table of Contents (the real thing)

Intro

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

Who is this book for?	xxx
We know what you're thinking	xxxi
Metacognition	xxxiii
Bend your brain into submission	xxxv
What you need for this book	xxxvi
Read me	xxxvii
The technical review team	xxxviii
Acknowledgments	xxxix

get productive with C#

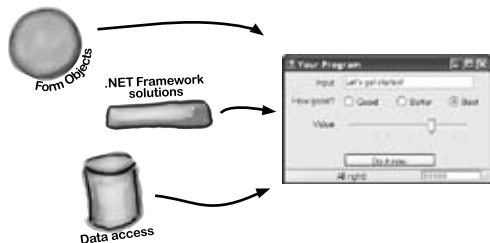
1

Visual Applications, in 10 minutes or less

Want to build great programs really fast?

With C#, you've got a **powerful 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 **focus on getting your work done**, rather than remembering which method parameter 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#	2
C# and the Visual Studio IDE make lots of things easy	3
Help the CEO go paperless	4
Get to know your users' needs before you start building your program	5
What you do in Visual Studio...	8
What Visual Studio does for you...	8
Develop the user interface	12
Visual Studio, behind the scenes	14
Add to the auto-generated code	15
We need a database to store our information	18
The IDE created a database	19
SQL is its own language	19
Creating the table for the Contact List	20
Finish building the table	25
Insert your card data into the database	26
Connect your form to your database objects with a data source	28
Add database-driven controls to your form	30
How to turn YOUR application into EVERYONE'S application	35
Give your users the application	36
You're NOT done: test your installation	37
You've built a complete data-driven application	38



it's all just code

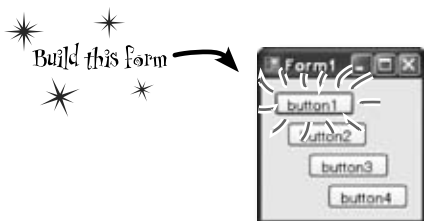
Under the hood

2

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.

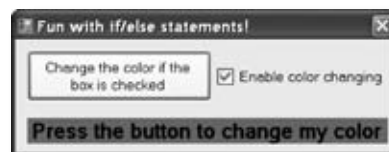
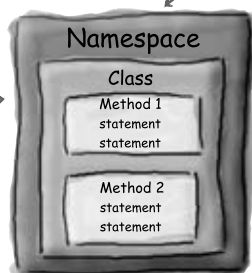
When you're doing this...	42
...the IDE does this	43
Where programs come from	44
The IDE helps you code	46
When you change things in the IDE, you're also changing your code	48
Anatomy of a program	50
Your program knows where to start	52
Two classes can be in the same namespace	59
Your programs use variables to work with data	60
C# uses familiar math symbols	62
Use the debugger to see your variables change	63
Loops perform an action over and over	65
Time to start coding	66
if/else statements make decisions	67
Set up conditions and see if they're true	68



Every time you make a new program, you define a namespace for it so that its code is separate from the .NET Framework classes.

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.



objects: get oriented!

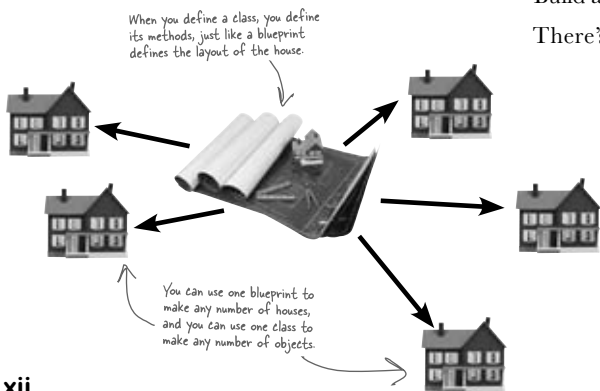
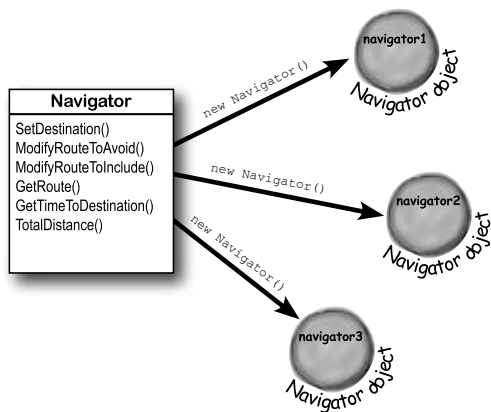
Making Code Make Sense

3

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	86
How Mike's car navigation system thinks about his problems	87
Mike's Navigator class has methods to set and modify routes	88
Use what you've learned to build a program that uses a class	89
Mike can use objects to solve his problem	92
You use a class to build an object	93
When you create a new object from a class, it's called an instance of that class	94
A better solution...brought to you by objects!	95
An instance uses fields to keep track of things	100
Let's create some instances!	101
What's on your program's mind	103
You can use class and method names to make your code intuitive	104
Give your classes a natural structure	106
Class diagrams help you organize your classes so they make sense	108
Build a class to work with some guys	112
Create a project for your guys	113
Build a form to interact with the guys	114
There's an easier way to initialize objects	117



types and references

It's 10:00. Do you know where your data is?

4

Data type, database, Lieutenant Commander Data...

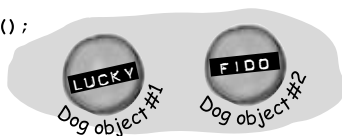
it's all important stuff. Without data, your programs are useless. You need **information** from your users, and you use that to look up or produce new information to give back to them. In fact, almost everything you do in programming involves **working with data** in one way or another. In this chapter, 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** (*psst...objects are data, too*).

The variable's type determines what kind of data it can store	126
A variable is like a data to-go cup	128
10 pounds of data in a 5 pound bag	129
Even when a number is the right size, you can't just assign it to any variable	130
When you cast a value that's too big, C# will adjust it automatically	131
C# does some casting automatically	132
When you call a method, the arguments must be compatible with the types of the parameters	133
Combining = with an operator	138
Objects use variables, too	139
Refer to your objects with reference variables	140
References are like labels for your object	141
If there aren't any more references, your object gets garbage-collected	142
Multiple references and their side effects	144
Two references means TWO ways to change an object's data	149
A special case: arrays	150
Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!	152
Objects use references to talk to each other	154
Where no object has gone before	155
Build a typing game	160

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



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



```
lucky = null;
```



C# Lab 1

A Day at the Races

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.

The spec: build a racetrack simulator	170
The Finished Product	178



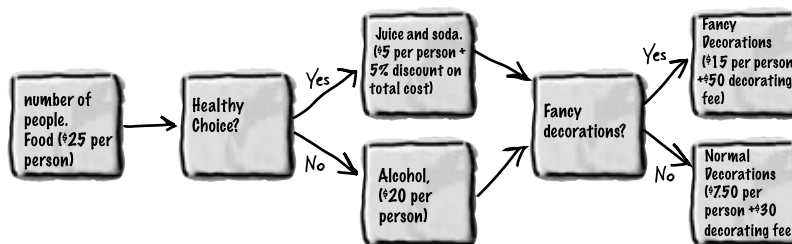
encapsulation

5

Keep your privates... private**Ever wished for a little more privacy?**

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

Kathleen is an event planner	180
What does the estimator do?	181
Kathleen's Test Drive	186
Each option should be calculated individually	188
It's easy to accidentally misuse your objects	190
Encapsulation means keeping some of the data in a class private	191
Use encapsulation to control access to your class's methods and fields	192
But is the realName field REALLY protected?	193
Private fields and methods can only be accessed from inside the class	194
Encapsulation keeps your data pristine	202
Properties make encapsulation easier	203
Build an application to test the Farmer class	204
Use automatic properties to finish the class	205
What if we want to change the feed multiplier?	206
Use a constructor to initialize private fields	207



inheritance

6

Your object's family tree

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

Ever run across an object that *almost* does exactly what you want *your* object to do? Found yourself wishing that if you could just *change a few things*, that object would be perfect? Well, that's just one reason that **inheritance** is one of the most powerful concepts and techniques in the C# language. Before you're through with this chapter, you'll learn how to **subclass** an object to get its behavior, but keep the **flexibility** to make changes to that behavior. You'll **avoid duplicate code**, **model the real world** more closely, and end up with code that's **easier to maintain**.

Kathleen does birthday parties, too	216
We need a BirthdayParty class	217
Build the Party Planner version 2.0	218
When your classes use inheritance, you only need to write your code once	226
Kathleen needs to figure out the cost of her parties, no matter what kind of parties they are.	226
Build up your class model by starting general and getting more specific	227
How would you design a zoo simulator?	228
Use inheritance to avoid duplicate code in subclasses	229
Think about how to group the animals	231
Create the class hierarchy	232
Every subclass extends its base class	233
A subclass can override methods to change or replace methods it inherited	238
Any place where you can use a base class, you can use one of its subclasses instead	239
A subclass can hide methods in the superclass	246
Use the override and virtual keywords to inherit behavior	248
Now you're ready to finish the job for Kathleen!	252
Build a beehive management system	257
First you'll build the basic system	258
Use inheritance to extend the bee management system	263



interfaces and abstract classes

Making classes keep their promises

7

Actions speak louder than words.

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

* **Inheritance**

Let's get back to bee-sics 270

We can use inheritance to create classes for different types of bees 271

An interface tells a class that it must implement certain methods and properties 272

Use the interface keyword to define an interface 273

Classes that implement interfaces have to include ALL of the interface's methods 275

You can't instantiate an interface, but you can reference an interface 278

Interface references work just like object references 279

You can find out if a class implements a certain interface with "is" 280

Interfaces can inherit from other interfaces 281

Upcasting works with both objects and interfaces 285

Downcasting lets you turn your appliance back into a coffee maker 286

Upcasting and downcasting work with interfaces, too 287

There's more than just public and private 291

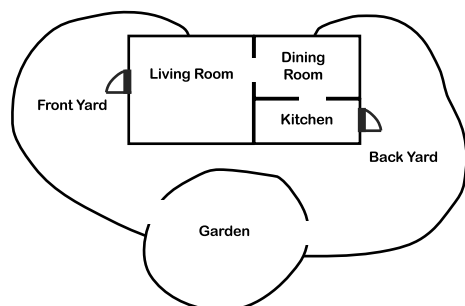
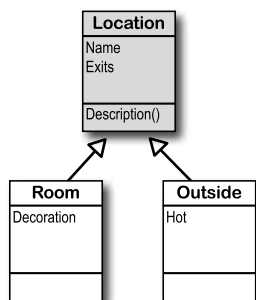
Access modifiers change visibility 292

Some classes should never be instantiated 295

An abstract class is like a cross between a class and an interface 296

An abstract method doesn't have a body 299

Polymorphism means that one object can take many different forms 307

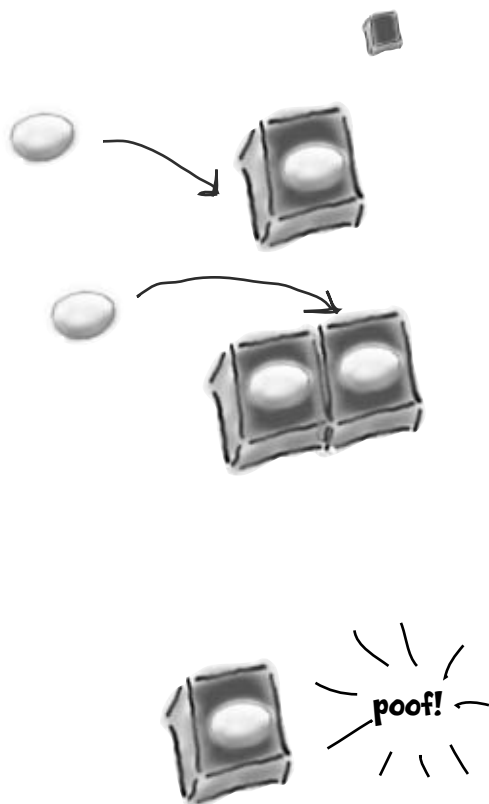
* **Abstraction*** **Encapsulation*** **Polymorphism**

enums and collections

8

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.



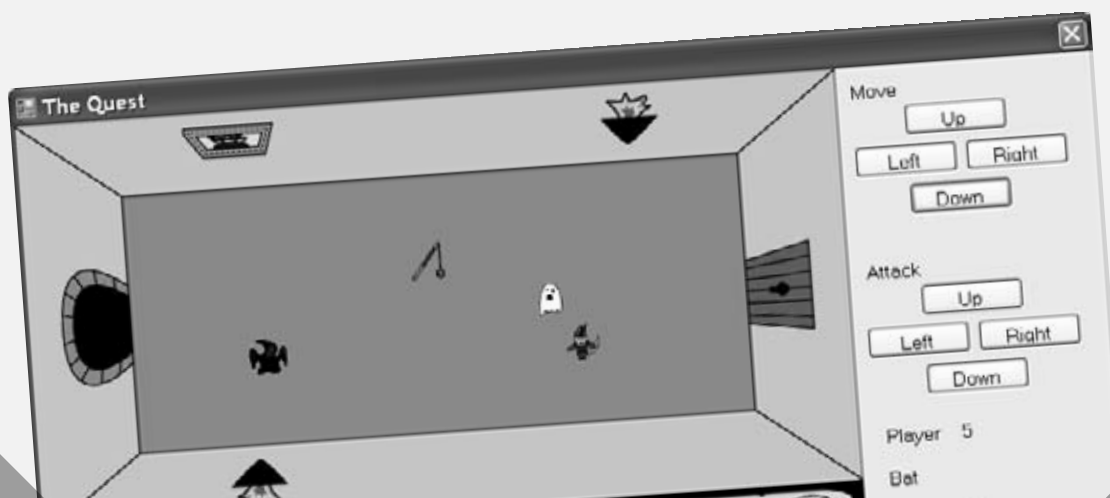
Strings don't always work for storing categories of data	328
Enums let you work with a set of valid values	329
Enums let you represent numbers with names	330
We could use an array to create a deck of cards...	333
Lists are more flexible than arrays	336
Generics can store any type	340
Collection initializers work just like object initializers	344
Let's create a List of Ducks	345
Lists are easy, but SORTING can be tricky	346
IComparable <Duck> helps your list sort its ducks	347
Use IComparer to tell your List how to sort	348
Create an instance of your comparer object	349
IComparer can do complex comparisons	350
Overriding a ToString() method lets an object describe itself	353
Update your foreach loops to let your Ducks and Cards print themselves	354
You can upcast an entire list using IEnumerable	356
You can build your own overloaded methods	357
The Dictionary Functionality Rundown	364
Build a program that uses a Dictionary	365
And yet MORE collection types...	377
A queue is FIFO—First In, First Out	378
A stack is LIFO—Last In, First Out	379

C# Lab 2

The Quest

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	386
The fun's just beginning!	406



reading and writing files

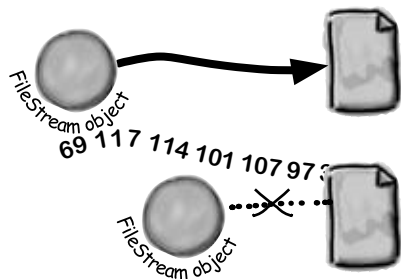
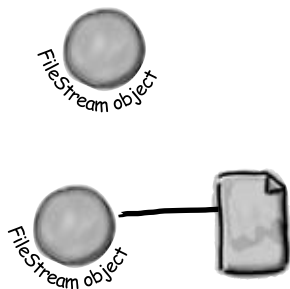
Save the byte array, save the world

9

Sometimes it pays to be a little persistent.

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

.NET uses streams to read and write data	408
Different streams read and write different things	409
A FileStream reads and writes bytes to a file	410
How to write text to a file in 3 simple steps	411
Reading and writing using two objects	415
Data can go through more than one stream	416
Use built-in objects to pop up standard dialog boxes	419
Dialog boxes are just another .NET control	420
Dialog boxes are objects, too	421
IDisposable makes sure your objects are disposed of properly	427
Avoid file system errors with using statements	428
Writing files usually involves making a lot of decisions	434
Use a switch statement to choose the right option	435
Serialization lets you read or write a whole object all at once	442
.NET uses Unicode to store characters and text	447
C# can use byte arrays to move data around	448
You can read and write serialized files manually, too	451
Working with binary files can be tricky	453
Use file streams to build a hex dumper	454
StreamReader and StreamWriter will do just fine (for now)	455
Use Stream.Read() to read bytes from a stream	456



exception handling

10

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: **master programmer**. 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	464
When your program throws an exception, .NET generates an Exception object.	468
All exception objects inherit from Exception	472
The debugger helps you track down and prevent exceptions in your code	473
Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager	474
Handle exceptions with try and catch	479
What happens when a method you want to call is risky?	480
Use the debugger to follow the try/catch flow	482
If you have code that ALWAYS should run, use a finally block	484
One class throws an exception, another class catches the exception	491
Bees need an OutOfHoney exception	492
An easy way to avoid a lot of problems: using gives you try and finally for free	495
Exception avoidance: implement IDisposable to do your own cleanup	496
The worst catch block EVER: catch-all plus comments	498
Temporary solutions are OK (temporarily)	499
A few simple ideas for exception handling	500
Brian finally gets his vacation...	505



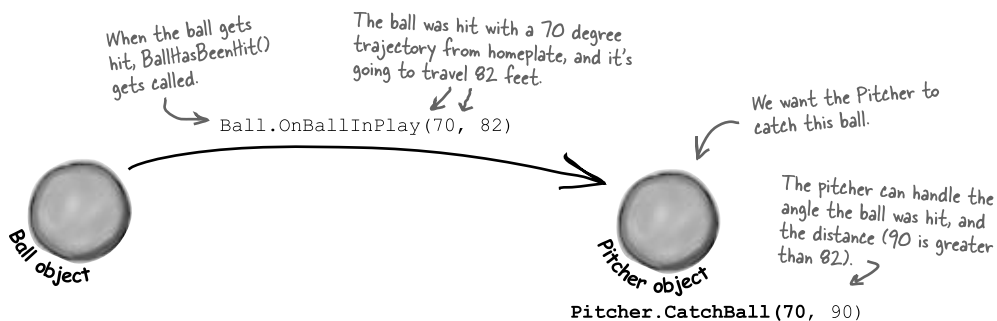
11

events and delegates

What your code does when you're not looking**Your objects are starting to think for themselves.**

You can't always control what your objects are doing. Sometimes things...happen. And when they do, you want your objects to be smart enough to **respond to anything** that pops up. And that's what events are all about. One object *publishes* an event, other objects *subscribe*, and everyone works together to keep things moving. Which is great, until you want your object to take control over who can listen. That's when **callbacks** will come in handy.

Ever wish your objects could think for themselves?	508
But how does an object KNOW to respond?	508
When an EVENT occurs...objects listen	509
Then, the other objects handle the event	511
Connecting the dots	512
The IDE creates event handlers for you automatically	516
Generic EventHandlers let you define your own event types	522
The forms you've been building all use events	523
One event, multiple handlers	524
Connecting event senders with event receivers	526
A delegate STANDS IN for an actual method	527
Delegates in action	528
An object can subscribe to an event...	531
Use a callback to control who's listening	532
A callback is just a way to use delegates	534



review and preview

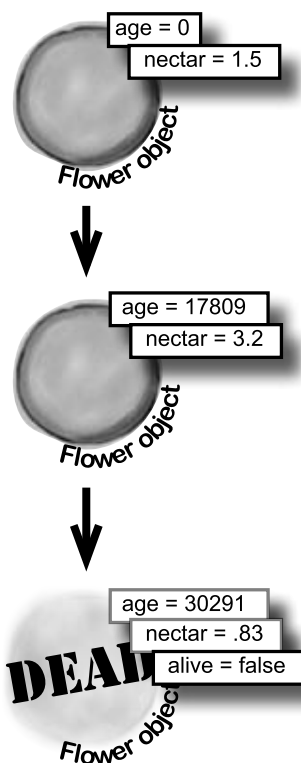
12

Knowledge, power, and building cool stuff

Learning's no good until you BUILD something.

Until you've actually written working code, it's hard to be sure if you really *get* some of the tougher concepts in C#. In this chapter, we're going to use what we've learned to do just that. We'll also get a preview of some of the new ideas coming up soon. And we'll do all that by building phase I of a **really complex application** to make sure you've got a good handle on what you've already learned from earlier chapters. So buckle up...it's time to **build some software!**

Life and death of a flower



You've come a long way, baby	542
We've also become beekeepers	543
The beehive simulator architecture	544
Building the beehive simulator	545
Life and death of a flower	549
Now we need a Bee class	550
P. A. H. B. (Programmers Against Homeless Bees)	554
The hive runs on honey	554
Filling out the Hive class	558
The hive's Go() method	559
We're ready for the World	560
We're building a turn-based system	561
Here's the code for World	562
Giving the bees behavior	568
The main form tells the world to Go()	570
We can use World to get statistics	571
Timers fire events over and over again	572
Let's work with groups of bees	580
A collection collects...DATA	581
LINQ makes working with data in collections and databases easy	583
One final challenge: Open and Save	585

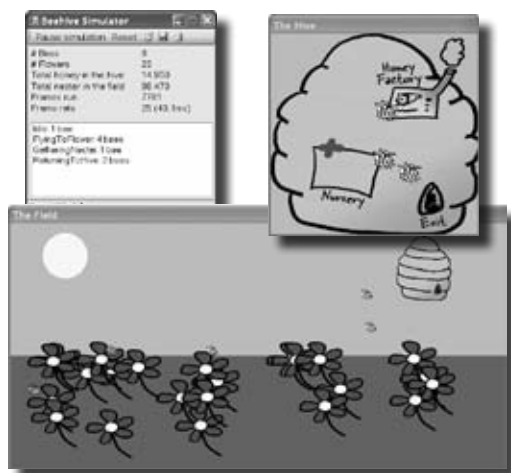
13

controls and graphics

Make it pretty**Sometimes you have to take graphics into your own hands.**

We've spent a lot of time relying on controls to handle everything visual in our applications. But sometimes that's not enough—like when you want to **animate a picture**. And once you get into animation, you'll end up **creating your own controls** for your .NET programs, maybe adding a little **double buffering**, and even **drawing directly onto your forms**. It all begins with the **Graphics** object, **bitmaps**, and a determination to not accept the graphics status quo.

You've been using controls all along to interact with your programs	590
Form controls are just objects	591
Use controls to animate the beehive simulator	592
Add a renderer to your architecture	594
Controls are well suited for visual display elements	596
Build your first animated control	599
Create a button to add the BeeControl to your form	602
Your controls need to dispose their controls, too!	603
A UserControl is an easy way to build a control	604
Your simulator's renderer will use your BeeControl to draw animated bees on your forms	606
Add the hive and field forms to the project	608
Build the renderer	609
You resized your Bitmaps using a Graphics object	618
Your image resources are stored in Bitmap objects	619
Use System.Drawing to TAKE CONTROL of graphics yourself	620
A 30-second tour of GDI+ graphics	621
Use graphics to draw a picture on a form	622
Graphics can fix our transparency problem...	627
Use the Paint event to make your graphics stick	628
A closer look at how forms and controls repaint themselves	631
Double buffering makes animation look a lot smoother	634
Use a Graphics object and an event handler for printing	640

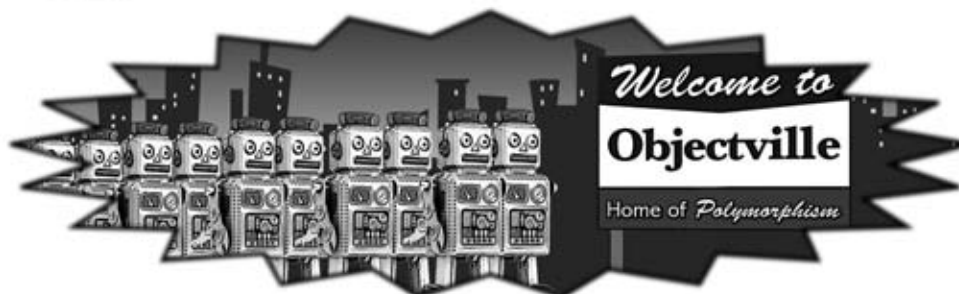


14

CAPTAIN AMAZING

THE DEATH OF THE OBJECT

Your last chance to DO something...your object's finalizer	654
When EXACTLY does a finalizer run?	655
Dispose() works with using, finalizers work with garbage collection	656
Finalizers can't depend on stability	658
Make an object serialize itself in its Dispose()	659
A struct looks like an object...	663
...but isn't an object	663
Values get copied; references get assigned	664
The stack vs. the heap: more on memory	667
Use out parameters to make a method return more than one value	670
Pass by reference using the ref modifier	671
Use optional parameters to set default values	672
Use nullable types when you need nonexistent values	673
Nullable types help you make your programs more robust	674
Captain Amazing...not so much	677
Extension methods add new behavior to EXISTING classes	678
Extending a fundamental type: string	680



LINQ

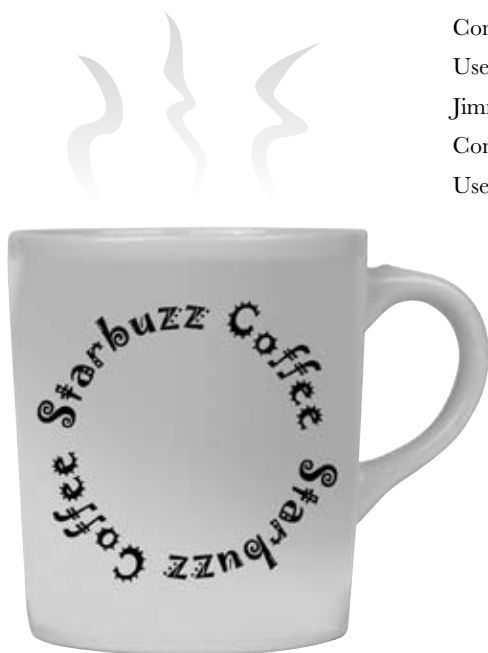
15

Get control of your data

It's a data-driven world...you better know how to live in it.

Gone are the days when you could program for days, even weeks, without dealing with **loads of data**. But today, **everything is about data**. In fact, you'll often have to work with data from **more than one place**...and in more than one format. Databases, XML, collections from other programs...it's all part of the job of a good C# programmer. 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**.

An easy project...	686
...but the data's all over the place	687
LINQ can pull data from multiple sources	688
.NET collections are already set up for LINQ	689
LINQ makes queries easy	690
LINQ is simple, but your queries don't have to be	691
LINQ is versatile	694
LINQ can combine your results into groups	699
Combine Jimmy's values into groups	700
Use join to combine two collections into one query	703
Jimmy saved a bunch of dough	704
Connect LINQ to a SQL database	706
Use a join query to connect Starbuzz and Objectville	710



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	714
And yet there's more to do...	733



leftovers




The top 11 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. The Basics	736
#2. Namespaces and assemblies	742
#3. Use BackgroundWorker to make your UI responsive	746
#4. The Type class and GetType()	749
#5. Equality, IEquatable, and Equals()	750
#6. Using yield return to create enumerable objects	753
#7. Refactoring	756
#8. Anonymous types, anonymous methods, and lambda expressions	758
#9. Serializing data using DataContractSerializer	760
#10. LINQ to XML	762
#11. Windows Presentation Foundation	764
Did you know that C# and the .NET Framework can...	766

 backgroundWorker 1

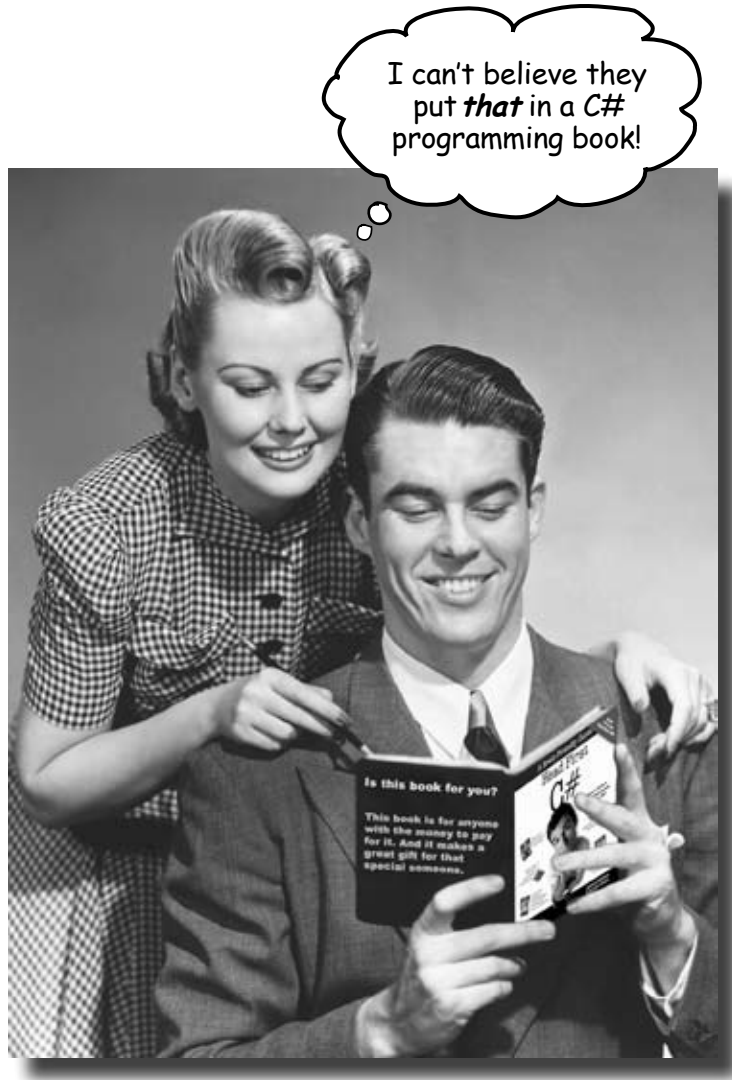
 fileSystemWatcher 1

 performanceCounter 1



how to use this book

Intro



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

Who is this book for?

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

- ① Do you want to **learn C#**?
- 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 writing a lot of code make you bored and a little twitchy?
- ② Are you a kick-butt C++ or Java programmer looking for a 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.



[Note from marketing: this book is for anyone with a credit card.]

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*.

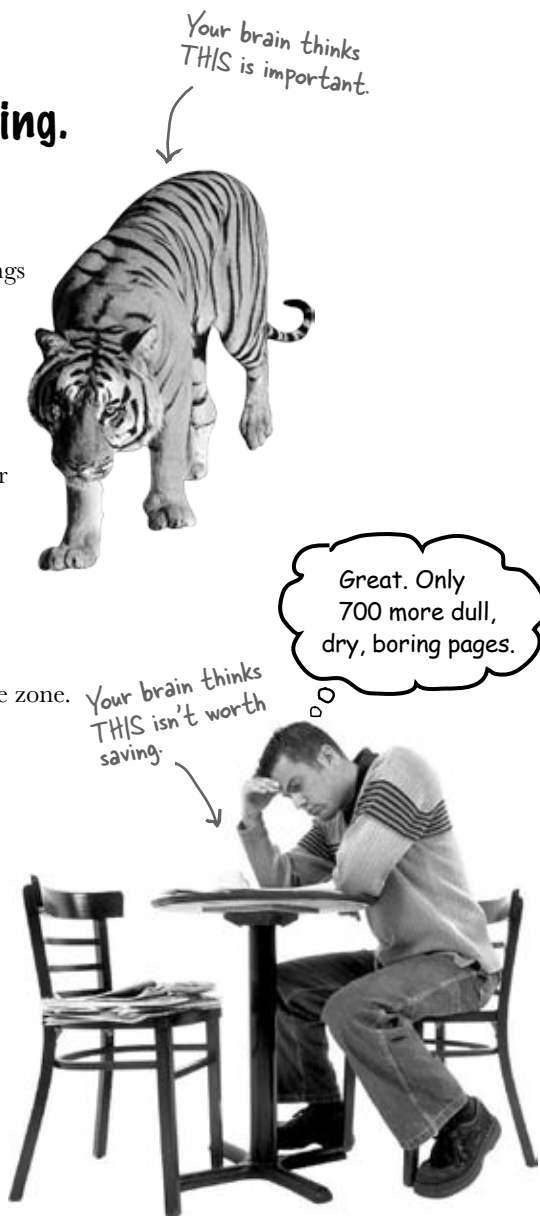
And that’s how your brain knows...

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.

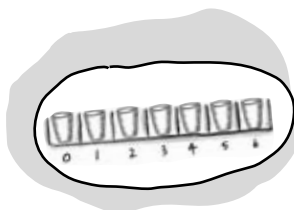
And there’s no simple way to tell your brain, “Hey brain, thank you very much, but no matter how dull this book is, and how little I’m registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around.”



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.

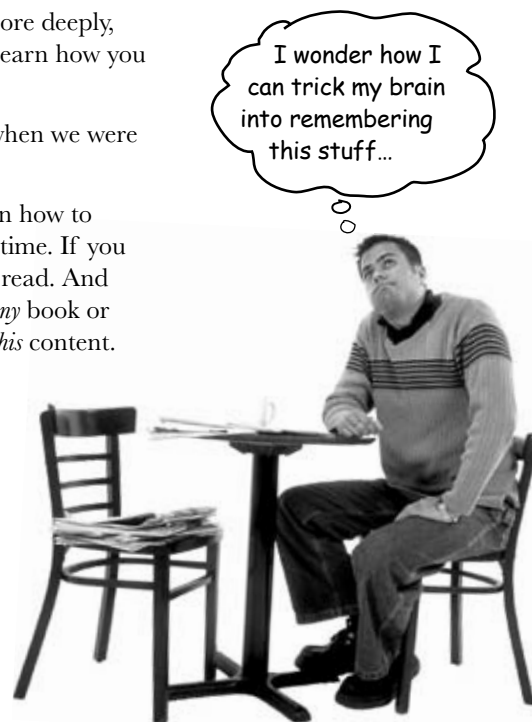
So just how **DO** you get your brain to treat C# like it was a hungry tiger?

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 make 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 more than 80 **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 exercises challenging-yet-do-able, because that's what most people prefer.

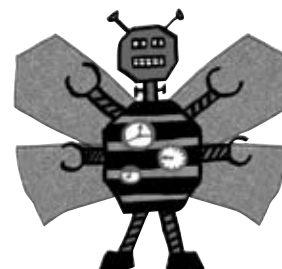
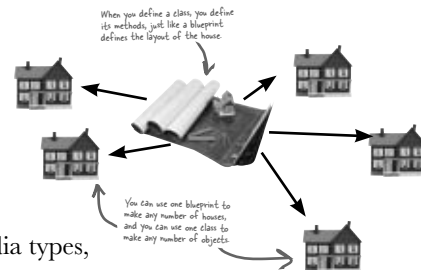
We used **multiple learning styles**, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented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



BULLET POINTS

Fireside Chats





Cut this out and stick it on your refrigerator.

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

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

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

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

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

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

③ **Read the "There are No Dumb Questions"**

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.

What you need for this book:

We wrote this book using Visual C# 2010 Express Edition, which uses C# 4.0 and .NET Framework 4.0. All of the screenshots that you see throughout the book were taken from that edition, so we recommend that you use it. If you're using Visual Studio 2010 Professional, Premium, Ultimate or Test Professional editions, you'll see some small differences, which we've pointed out wherever possible. You can download the Express Edition for free from Microsoft's website—it installs cleanly alongside other editions, as well as previous versions of Visual Studio.

SETTING UP VISUAL STUDIO 2010 EXPRESS EDITION

- It's easy enough to download and install Visual C# 2010 Express Edition. Here's the link to the Visual Studio 2010 Express Edition download page:

<http://www.microsoft.com/express/downloads/>

You don't need to check any of the options in the installer to get the code in this book to run, but feel free to if you want.



If you absolutely must use an older version of Visual Studio, C# or the .NET Framework, then please keep in mind that you'll come across topics in this book that won't be compatible with your version. The C# team at Microsoft has added some pretty cool features to the language. Keep in mind that if you're not using the latest version, there will be some code in this book that won't work.

- Download the installation package for Visual C# 2010 Express Edition. Make sure you do a complete installation. That should install everything that you need: the IDE (which you'll learn about), .NET Framework 4.0, and other tools.
- Once you've got it installed, you'll have a new Start menu option: **Microsoft Visual C# 2010 Express Edition**. Click on it to bring up the IDE, and you're all set.

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 exercises 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!

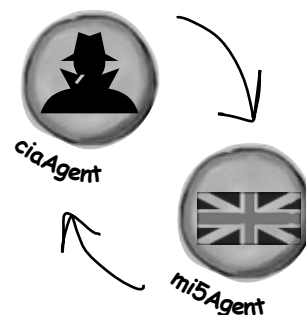
The one big assumption that we made when we wrote this book is that you want to learn how to program in C#. So we know you want to get your hands dirty right away, and dig right into the code. We gave you a lot of opportunities to sharpen your skills by putting exercises in every chapter. We've labeled 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" exercises 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 exercises you will find hints to point you in the right direction.

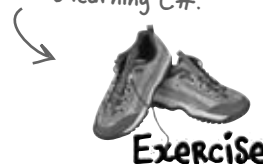
We use a lot of diagrams to make tough concepts easier to understand.



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



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



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



The technical review team

Lisa Kellner



Chris Burrows



We're especially grateful for Chris's insight and almost ridiculously helpful feedback.

Not pictured (but just as awesome as the reviewers from the first edition): Joe Albahari, Jay Hilyard, Aayam Singh, Theodore, Peter Ritchie, Bill Meitelski, Andy Parker, Wayne Bradney, Dave Murdoch, Bridgette Julie Landers. And special thanks to Jon Skeet for his thorough review and suggestions for the first edition!

Nick Paladino



David really helped us out, especially with some very neat IDE tricks.

David Sterling



Technical Reviewers:

When we wrote this book, it had a bunch of mistakes, issues, problems, typos, and terrible arithmetic errors. OK, it wasn't quite that bad. But we're still really grateful for the work that our technical reviewers did for the 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 **Chris Burrows** and **David Sterling** for their enormous amount of technical guidance. We also want to thank **Lisa Kellner**—this is our sixth 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 **Nick Paladino**. Thanks!

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.

David Sterling has worked on the Visual C# Compiler team for nearly 3 years.

Nicholas Paldino has been a Microsoft MVP for .NET/C# since the discipline's inception in the MVP program and has over 13 years of experience in the programming industry, specifically targeting Microsoft technologies.

Acknowledgments

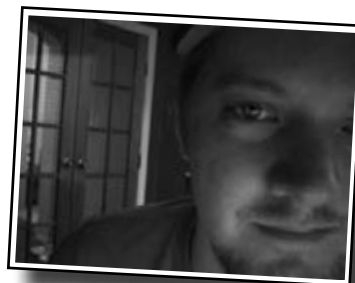
Our editor:

We want to thank our editors, **Brett McLaughlin** and **Courtney Nash**, for editing this book. Brett helped with a lot of the narrative, and the comic idea in Chapter 14 was completely his, and we think it turned out really well. Thanks!



←
Courtney Nash

Brett McLaughlin →



The O'Reilly team:



↖
Lou Barr

Lou Barr is an amazing graphic designer who went above and beyond on this one, putting in unbelievable hours and coming up with some pretty amazing visuals. If you see anything in this book that looks fantastic, you can thank her (and her mad InDesign skillz) for it. She did all of the monster and alien graphics for the labs, and the entire comic book. Thanks so much, Lou! You are our hero, and you're awesome to work with.



↖
Sanders Kleinfeld

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 **Rachel Monaghan**, indexer **Lucie Haskins**, **Emily Quill** for her sharp proofread, **Ron Bilodeau** for volunteering his time and preflighting expertise, and **Sanders Kleinfeld** 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** and **Mike Hendrickson**. And if you're reading this book right now, then you can thank the greatest publicity team in the industry: **Marsee Henon**, **Sara Peyton**, **Mary Rotman**, **Jessica Boyd**, **Kathryn Barrett**, 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://my.safaribooksonline.com/?portal=oreilly>.

1 get productive with c#

Visual Applications, in 10 minutes or less

Don't worry, Mother. With Visual Studio and C#, you'll be able to program so fast that you'll never burn the pot roast again.



Want to build great programs really fast?

With C#, you've got a **powerful 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 **focus on getting your work done**, rather than remembering which method parameter was for the *name* of a button, and which one was for its *label*. Sound appealing? Turn the page, and let's get programming.

makes it easy

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.

↙ The IDE—or Visual Studio Integrated Development Environment—is an important part of working in C#. It's a program that helps you edit your code, manage your files, and publish your projects.

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 form, your program needs a whole bunch of repetitive code.

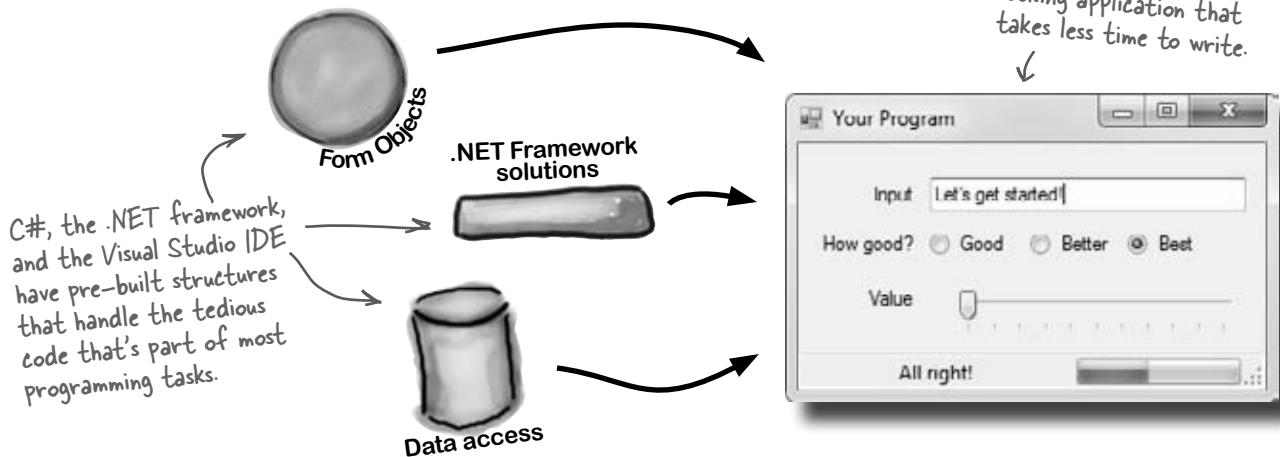
```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
namespace A_New_Program
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
       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

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

↖ It takes all this code just to draw a button on a form. Adding a few more visual elements to the form could take 10 times as much code.

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

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



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

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

- 1 **Build an application, FAST.** Creating programs in C# is a snap. The language is powerful 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 Form Designer in the Visual Studio IDE is one of the easiest design tools to use out there. It does so much for you that you'll find that making stunning user interfaces 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 **Create and interact with databases.** The IDE includes an easy-to-use interface for building databases, and integrates seamlessly with SQL Server Compact Edition and many other popular database systems.
- 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 just lets you focus on your program, your work (or fun!), and your customers. But the IDE handles all the grunt work, such as:
 - ★ Keeping track of all your projects
 - ★ Making it easy to edit your project's code
 - ★ Keeping track of your project's graphics, audio, icons, and other resources
 - ★ Managing and interacting with databases

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

↖ You're going to see exactly what we mean next.

Help the CEO go paperless

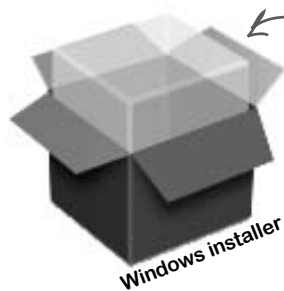
The Objectville Paper Company just hired a new CEO. He loves hiking, coffee, and nature...and he's decided that to help save forests, he wants to become a paperless executive, starting with his contacts. He's heading to Aspen to go skiing for the weekend, and expects a new address book program by the time he gets back. Otherwise...well...it won't be just the old CEO who's looking for a job.



Get to know your users' needs before you start building your program

Before we can start writing the address book application—or *any* application—we need to take a minute and think about **who's going to be using it**, and **what they need** from the application.

- ① The CEO needs to be able to run his address book program at work and on his laptop, too. He'll need an installer to make sure that all of the right files get onto each machine.



The CEO wants to be able to run his program on his desktop and laptop, so an installer is a must.

- ② The Objectville Paper Company sales team wants to access his address book, too. They can use his data to build mailing lists and get client leads for more paper sales.

The CEO figures a database would be the best way for everyone in the company to see his data, and then he can just keep up with one copy of all his contacts.

We already know that Visual C# makes working with databases easy. Having contacts in a database lets the CEO and the sales team all access the information, even though there's only one copy of the data.

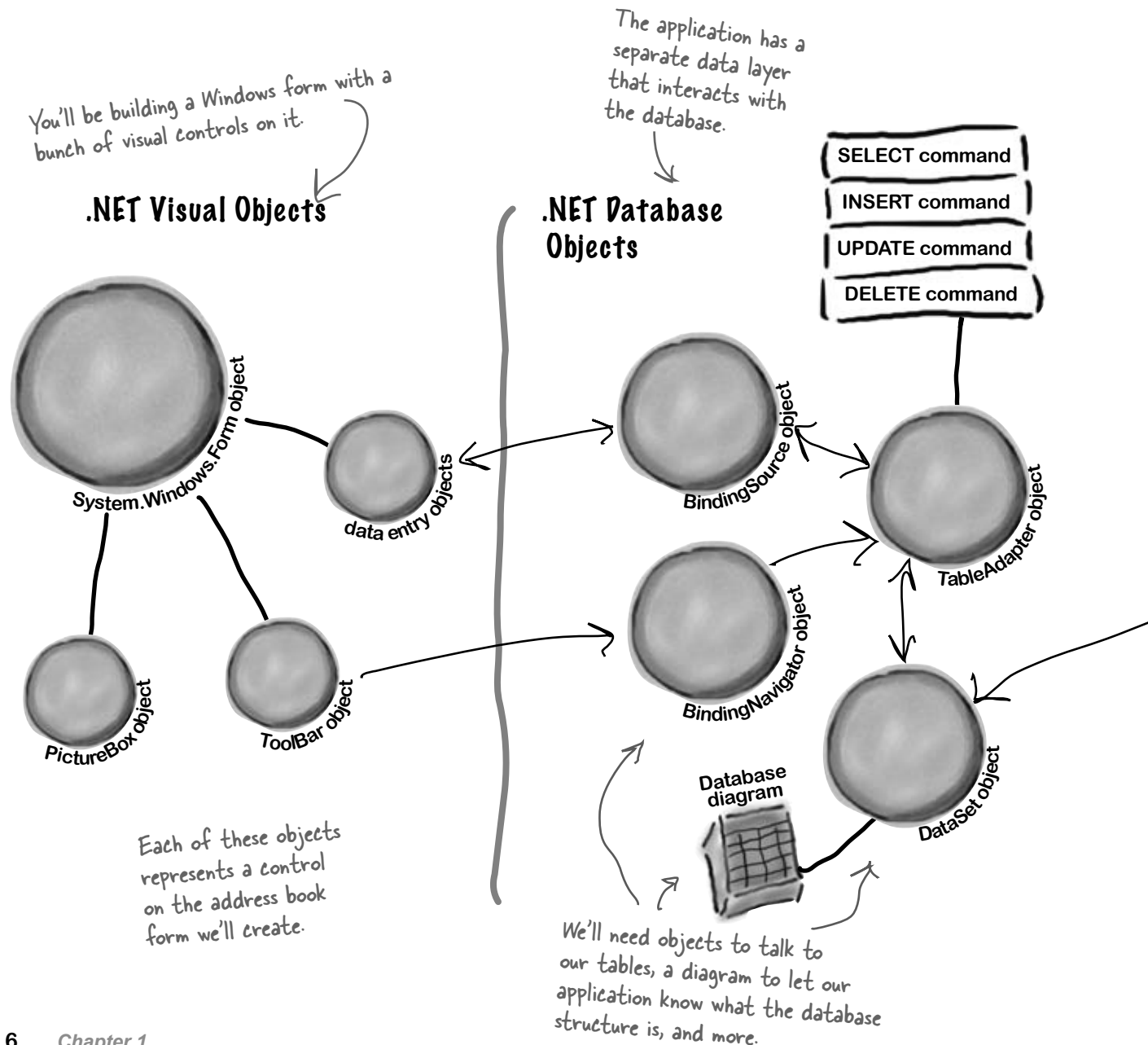


Think about your users and their needs before you start building the code, and they'll be happy with the final product once you're done!

Here's what you're going to build

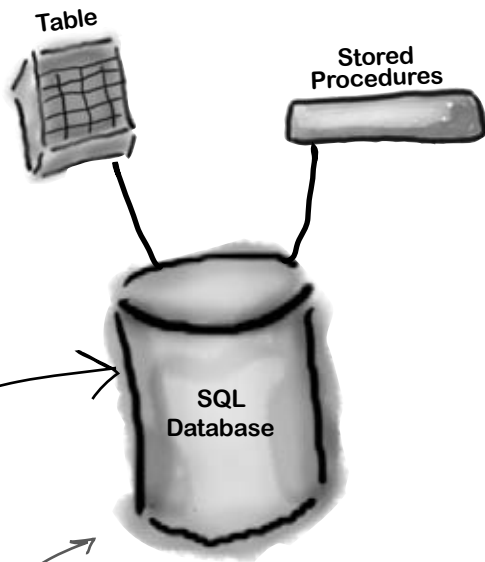
You're going to need an application with a graphical user interface, objects to talk to a database, the database itself, and an installer. It sounds like a lot of work, but you'll build all of this over the next few pages.

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



The data is all stored in a table in a SQL Server Compact database.

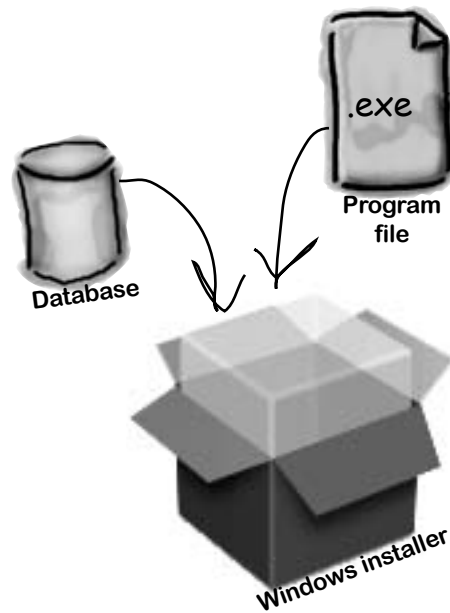
Data Storage



Here's the database itself, which Visual Studio will help us create and maintain.

Once the program's built, it'll be packaged up into a Windows installer.

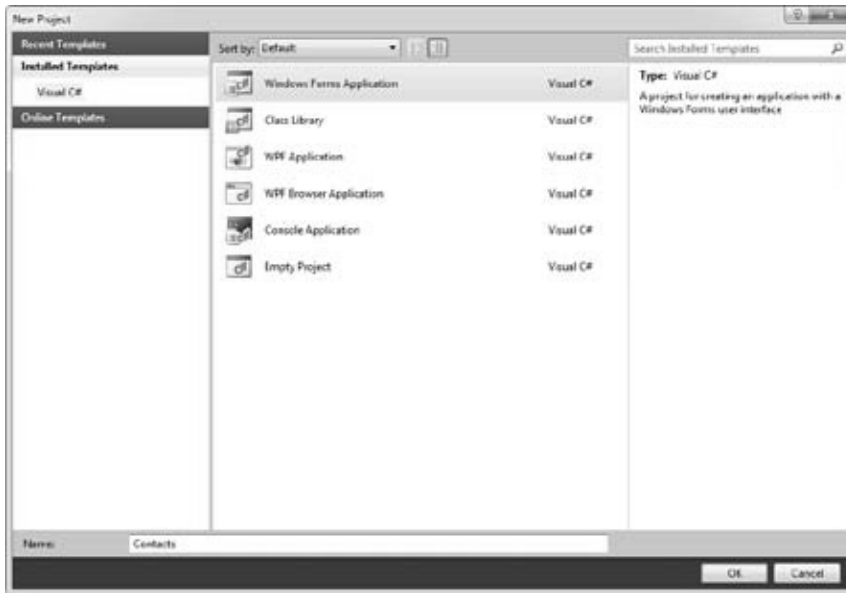
Deployment Package



The sales department will just need to point and click to install and then use his program.

What you do in Visual Studio...

Go ahead and start up Visual Studio, if you haven't already. Skip over the start page and select New Project from the **File** menu. Name your project "Contacts" and click OK. There are several project types to choose from. Select **Windows Forms Application** and choose "Contacts" as the name for your new project by entering it in the "Name" box at the bottom of the "New Project" window.



Watch it!

Things may look a bit different in your IDE.

This is what the "New Project" window looks like in Visual Studio 2010 Express Edition. If you're using the Professional or Team Foundation edition, it might be a bit different. But don't worry, everything still works exactly the same.

What Visual Studio does for you...

As soon as you save the project, the IDE creates `Form1.cs`, `Form1.Designer.cs`, and `Program.cs` file, when you create a new project. It adds these to the Solution Explorer window, and by default, puts those files in `My Documents\Visual Studio 2010\Projects\Contacts\`.

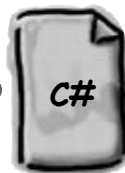
Make sure that you save your project as soon as you create it by selecting "Save All" from the File menu—that'll save all of the project files out to the folder. If you select "Save", it just saves the one you're working on.

This file contains the C# code that defines the behavior of the form.



Form1.cs

This has the code that starts up the program and displays the form.



Program.cs

The code that defines the form and its objects lives here.



Form1.Designer.cs

Visual Studio creates all three of these files automatically.

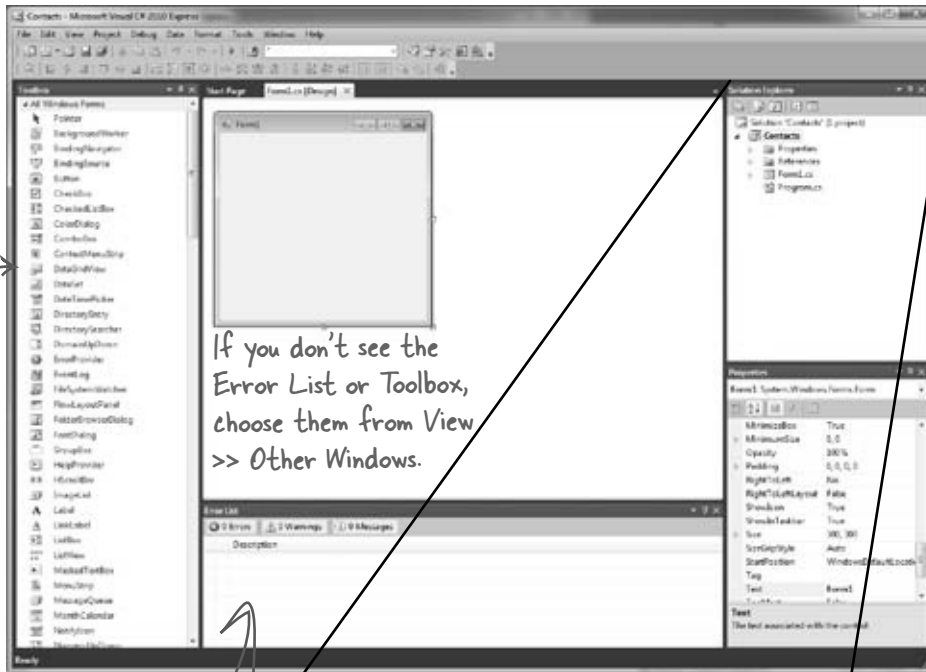
Sharpen your pencil

Below is what your screen probably looks like right now. You should be able to figure out the purpose of most of these windows and files based on what you already know. Make sure you open the Toolbox and Error List windows by **choosing them from the View >> Other Windows menu**. Then in each of the blanks, try and fill in an annotation saying what that part of the IDE does. We've done one to get you started.

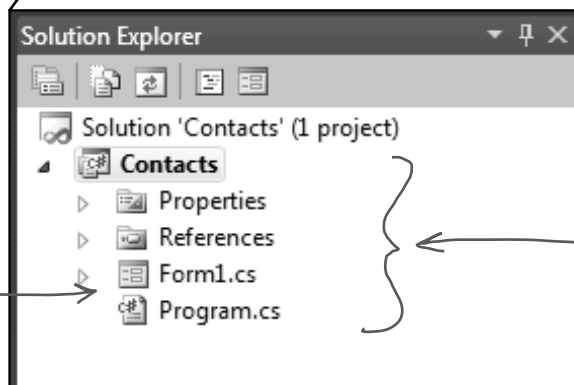
This toolbar has buttons that apply to what you're currently doing in the IDE.

If your IDE doesn't look exactly like this picture, you can select "Reset Window Layout" from the Window menu.

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



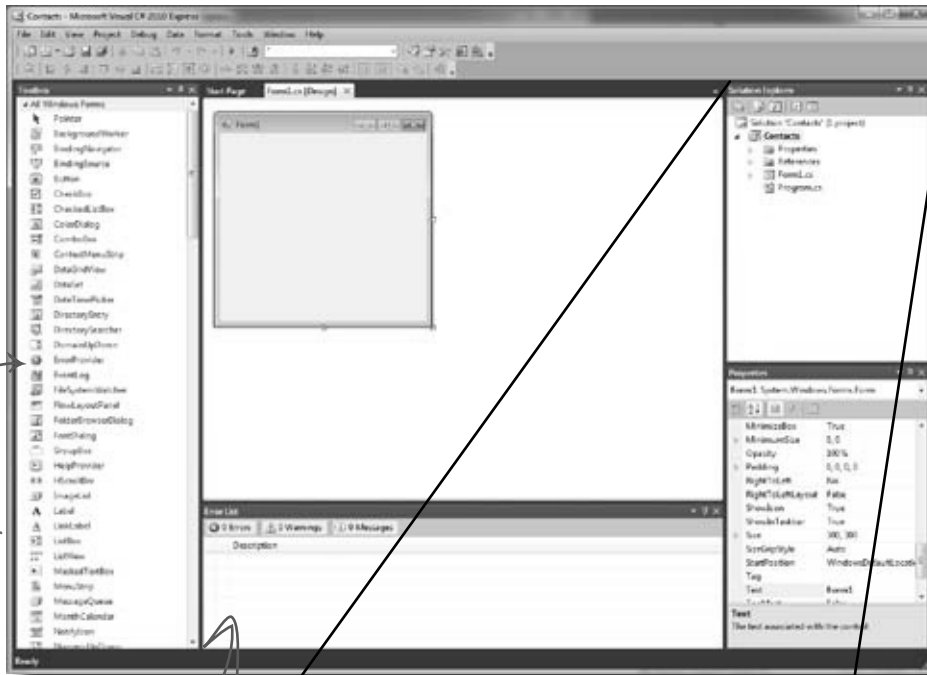
If you don't see the Error List or Toolbox, choose them from View >> Other Windows.



know your ide

Sharpen your pencil Solution

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

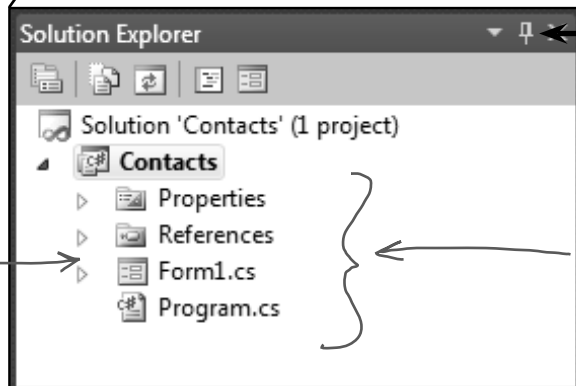


This is the toolbox. It has a bunch of visual controls that you can drag onto your form.

This window shows properties of the control currently selected on your form.

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

The Form1.cs and Program.cs files that the IDE created for you when you added the new project appear in the Solution Explorer.



See this little pushpin icon? If you click it, you can turn auto-hide on or off. The Toolbox window has auto-hide turned on by default.

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

there are no Dumb Questions

Q: So if 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 forms. 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: I created a new project in Visual Studio, but when I went into the “Projects” folder under My Documents, I didn't see it there. What gives?

A: When you first create a new project in Visual Studio 2010 Express, the IDE creates the project in your `Local Settings\Application Data\Temporary Projects` folder. When you save the project for the first time, it will prompt you for a new filename, and save it in the `My Documents\Visual Studio 2010\Projects` folder. If you try to open a new project or close the temporary one, you'll be prompted to either save or discard the temporary project. (NOTE: The other, non-Express versions of Visual Studio do not use a temporary projects folder. They create the project directly in Projects!)

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 (Professional and Team Foundation) aren't going to get in the way of writing C# and creating fully functional, complete applications.

Q: Can I change the names of the files the IDE generates for me?

A: Absolutely. When you create a new project, the IDE gives you a default form called `Form1` (which has files called `Form1.cs`, `Form1.Designer.cs`, and `Form1.resx`). But you can use the Solution Explorer to change the names of the files to whatever you want. By default, the names of the files are the same as the name of the form. If you change the names of the files, you'll be able to see in the Properties window that the form will still be called `Form1`. You can change the name of the form by changing the “(Name)” line in the Properties window. If you do, the filenames won't change.

C# doesn't care what names you choose for your files or your forms (or any other part of the program), although there are a few rules for this. But if you choose good names, it makes your programs easier to work with. For now, don't worry about names—we'll talk a lot more about how to choose good names for parts of your program later on.

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.

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

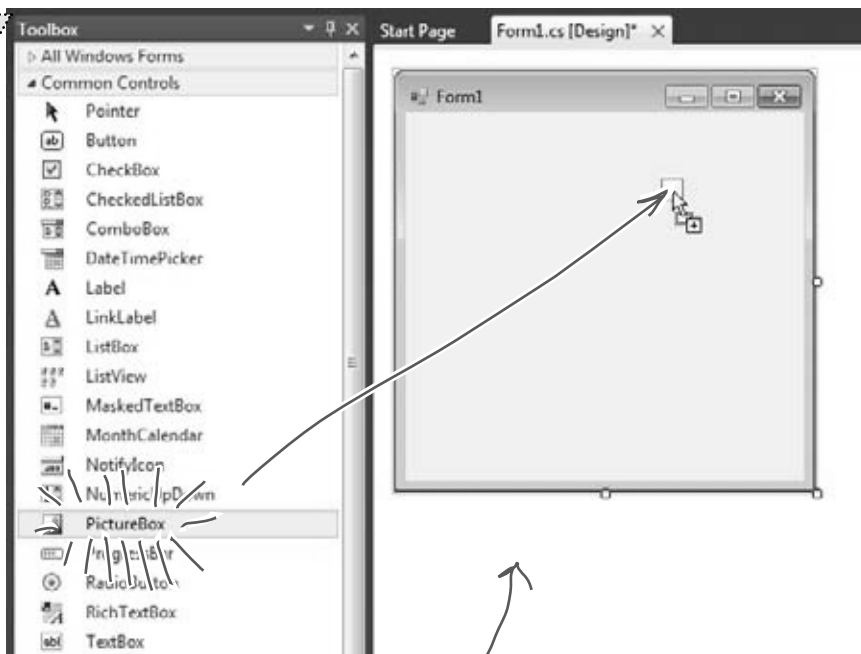
Develop the user interface

Adding controls and polishing the user interface is as easy as dragging and dropping with the Visual Studio IDE. Let's add a logo to the form:

1 Use the PictureBox control to add a picture.

Click on the PictureBox control in the Toolbox, and drag it onto your form. In the background, the IDE added code to `Form1.Designer.cs` for a new picture control.

If you don't see the toolbox, try hovering over the word "Toolbox" that shows up in the upper left-hand corner of the IDE. If it's not there, select "Toolbox" from the View menu to make it appear.



Every time you make a change to a control's properties on the form, the code in `Form1.Designer.cs` is getting changed by the IDE.

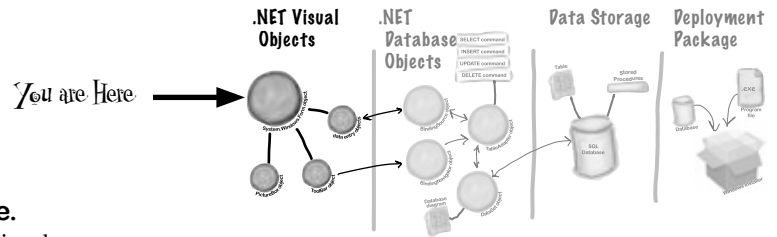


Form1.Designer.cs



It's OK if you're not a pro at user interface design.

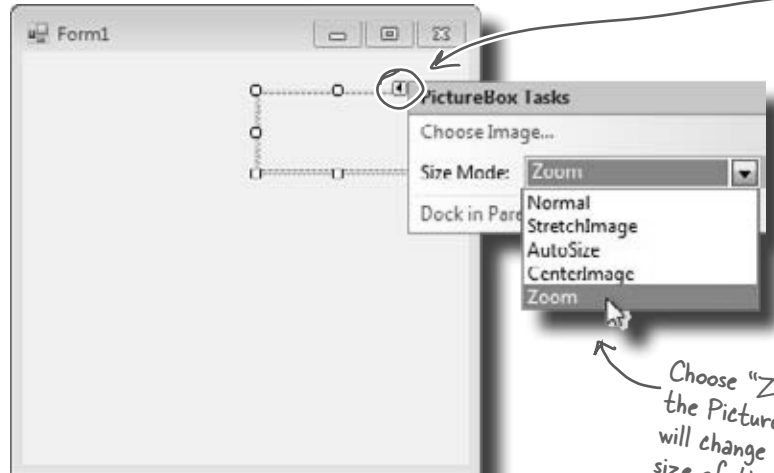
We'll talk a lot more about designing good user interfaces later on. For now, just get the logo and other controls on your form, and worry about **behavior**. We'll add some style later.



2 Set the PictureBox to Zoom mode.

Every control on your form has properties that you can set. Click the little black arrow for a control to access these properties. Change the PictureBox's Size property to "Zoom" to see how this works:

You can also use the "Properties" window in the IDE to set the Size property. The little black arrow is just there to make it easy to access the most common properties of any control.



Click on this little black arrow to access a control's properties.

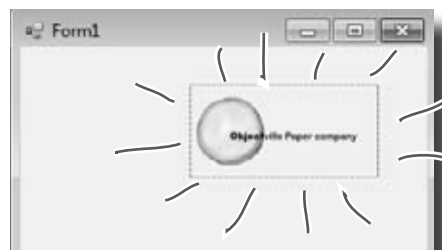
Choose "Zoom" so that the PictureBox frame will change to match the size of the picture you put in it.

Then click "Choose Image" to bring up the Select Resource dialog box so you can import a local resource.

3 Download the Objectville Paper Company logo.

Download the Objectville Paper Co. logo from Head First Labs (<http://www.headfirstlabs.com/books/hfcsharp>) and save it to your hard drive.

Then click the PictureBox properties arrow, and select Choose Image. You'll see a Select Resources window pop up. Click the "Local Resource" radio button to enable the "Import..." button at the top of the form. Click that button, find your logo, and you're all set.



Here's the OPC logo, and the PictureBox zooms to get the size just right.

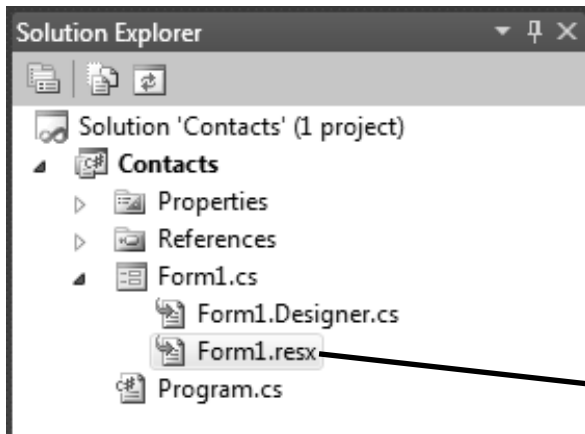
Visual Studio, behind the scenes

Every time you do something in the Visual Studio IDE, the IDE is **writing code for you**. When you created the logo and told Visual Studio to use the image you downloaded, Visual Studio created a resource and associated it with your application. A **resource** is any graphics file, audio file, icon, or other kind of data file that gets bundled with your application. The graphics file gets integrated into the program, so that when it's installed on another computer, the graphic is installed along with it and the PictureBox can use it.

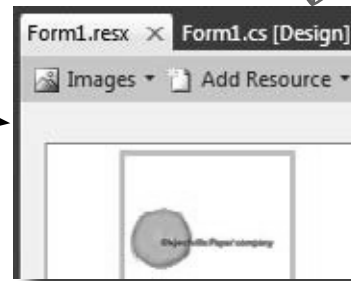
When you dragged the PictureBox control onto your form, the IDE automatically created a resource file called `Form1.resx` to store that resource and keep it in the project. Double-click on this file, and you'll be able to see the newly imported image.



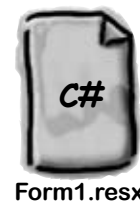
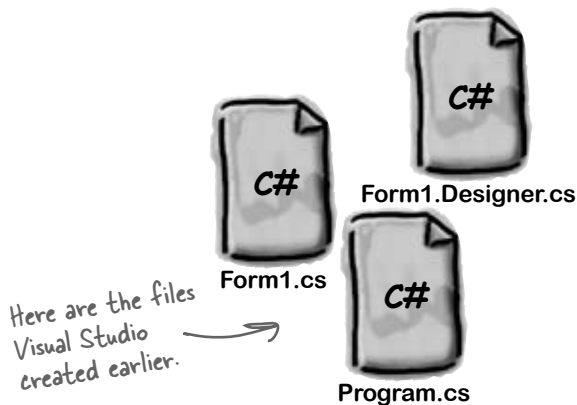
This image is now a resource of the Contact List application.



Go to the Solution Explorer and click on the “expand” icon next to `Form1.cs` to expand it (if it's not already expanded). This will display two files: `Form1.Designer.cs` and `Form1.resx`. Double-click on `Form1.resx`, click on the arrow next to “Strings”, and select “Images” from the drop-down list (or hit Ctrl-2) to see the logo that you imported. That file is what links it to the PictureBox, and the IDE added code to do the linking.



If you chose the other “Import.” button from the Select Resource dialog on the last page, then your image will show up in the Resources folder in the Solution Explorer instead. Don't worry—just go back to Select Resources, choose “Local Resource,” and reimport the image into the resources, and it'll show up here.

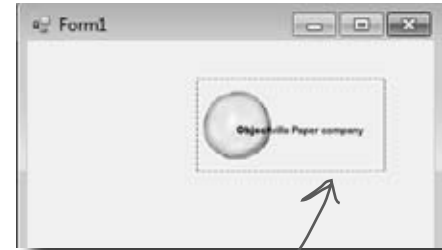


When you imported the image, the IDE created this file for you. It contains all of the resources (graphics, video, audio and other stored data) associated with `Form1`.

Add to the auto-generated code

The IDE creates lots of code for you, but you'll still want to get into this code and add to it. Let's set the logo up to show an About message when the users run the program and click on the logo.

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



```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    private void pictureBox1_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Contact List 1.0.\nWritten by: Your Name", "About");
    }
}
```

When you double-clicked on the PictureBox control, the IDE created this method. It will run every time a user clicks on the logo in the running application.

This method name gives you a good idea about when it runs: when someone clicks on this PictureBox control.

When you double-click on the PictureBox, it will open this code up with a cursor blinking right here. Ignore any windows the IDE pops up as you type; it's trying to help you, but we don't need that right now.

Type in this line of code. It causes a message box to pop up with the text you provide. The box will be titled "About".

Once you've typed in the line of code, save it using the Save icon on the IDE toolbar or by selecting "Save" from the File menu. Get in the habit of doing "Save All" regularly!

there are no Dumb Questions

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.

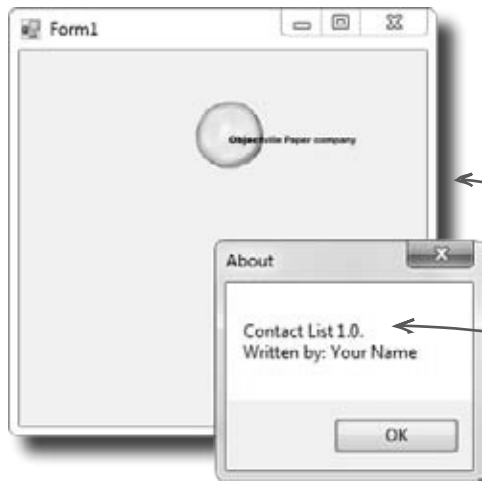
Q: What does that \n thing do?

A: That's a line break. It tells C# to put "Contact List 1.0." on one line, and then start a new line for "Written by:".

run the app (already!)

You can already run your application

Press the F5 key on your keyboard, or click the green arrow button (▶) on the toolbar to check out what you've done so far. (This is called "debugging," which just means running your program using the IDE.) You can stop debugging by selecting "Stop Debugging" from the Debug menu or clicking this toolbar button: .



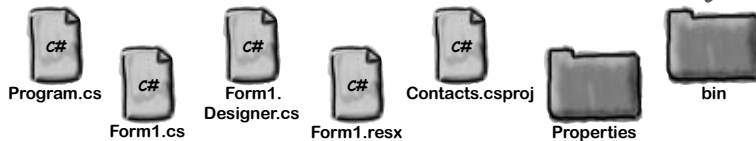
All three of these buttons work—and you didn't have to write any code to make them work.

Clicking on the OPC logo brings up the About box you just coded.

Where are my files?

When you run your program, Visual Studio copies your files to My Documents\Visual Studio 2010\Projects\Contacts\Contacts\bin\debug. You can even hop over to that directory and run your program by double-clicking on the .exe file the IDE creates.

C# turns your program into a file that you can run, called an **executable**. You'll find it in here, in the debug folder.



This isn't a mistake; there are two levels of folders. The inner folder has the actual C# code files.

there are no Dumb Questions

Q: In my IDE, the green arrow is marked as "Debug." Is that a problem?

A: No. Debugging, at least for our purposes right now, just means running your application inside the IDE. We'll talk a lot more about debugging later, but for now, you can simply think about it as a way to run your program.

Q: I don't see the Stop Debugging button on my toolbar. What gives?

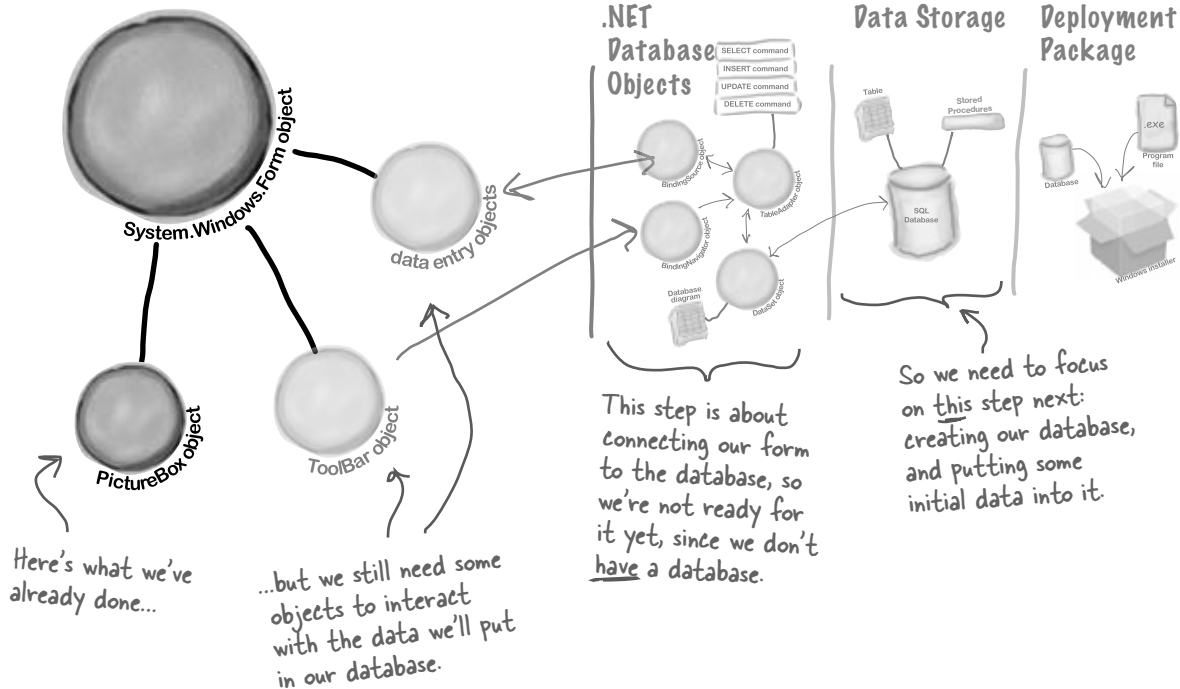
A: The Stop Debugging button shows up in a special toolbar that **only shows up** when your program is running. Try starting the application again, and see if it appears.

Here's what we've done so far

We've built a form and created a PictureBox object that pops up a message box when it's clicked on. Next, we need to add all the other fields from the card, like the contact's name and phone number.

Let's store that information in a database. Visual Studio can connect fields directly to that database for us, which means we don't have to mess with lots of database access code (which is good). But for that to work, we need to create our database so that the controls on the form can hook up to it. So we're going to jump from the .NET Visual Objects straight to the Data Storage section.

.NET Visual Objects



Visual Studio can generate code to connect your form to a database, but you need to have the database in place **BEFORE** generating that code.

We need a database to store our information

Before we add the rest of the fields to the form, we need to create a database to hook the form up to. The IDE can create lots of the code for connecting our form to our data, but we need to define the database itself first.

Make sure you've stopped debugging before you continue.

1 Add a new SQL database to your project.

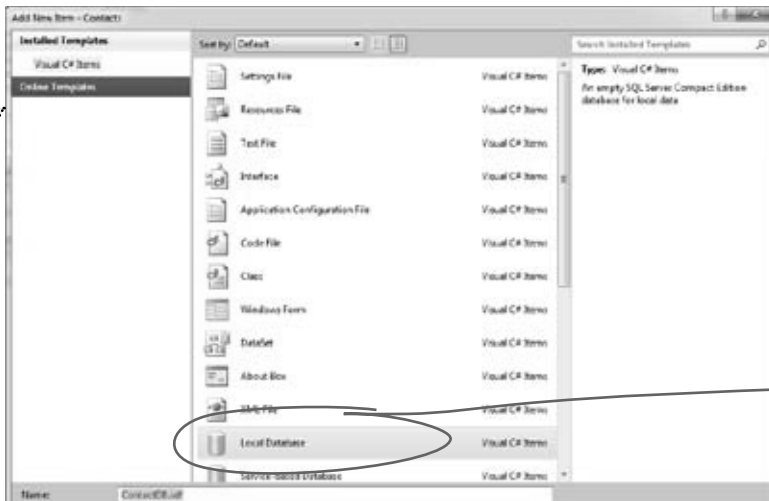
In the Solution Explorer, **right-click the Contacts project**, select **Add**, and then choose **New Item**. Choose the **SQL Database** icon, and name it **ContactDB.sdf**.

This file is our new database.



ContactDB.sdf

Choose **Local Database** to create a **SQL Server Compact Edition** file, which will hold your entire database. Name your file **ContactDB.sdf**.



A **Local Database** is actually a **SQL Server Compact Edition** database file, which typically has the extension **SDF**. It gives you an easy way to embed a database into your program.

2 Click on the Add button in the Add New Item window.

3 Cancel the Data Source Configuration Wizard.

For now, we want to skip configuring a data source, so click the **Cancel** button. We'll come back to this once we've set up our database structure.

4 View your database in the Solution Explorer.

Go to the **Solution Explorer**, and you'll see that **ContactDB** has been added to the file list. Double-click **ContactDB.sdf** in the **Solution Explorer** and look at the left side of your screen. The **Toolbox** has changed to a **Database Explorer**.



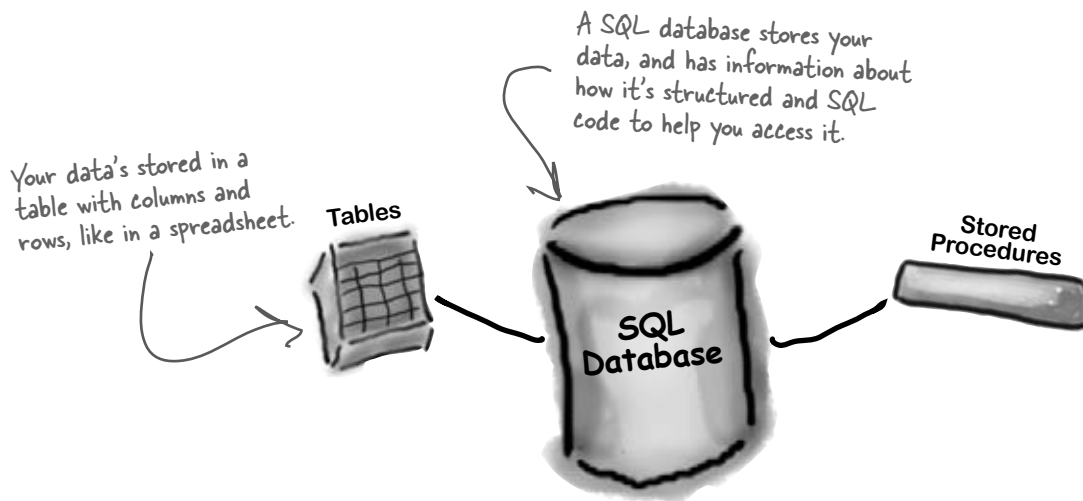
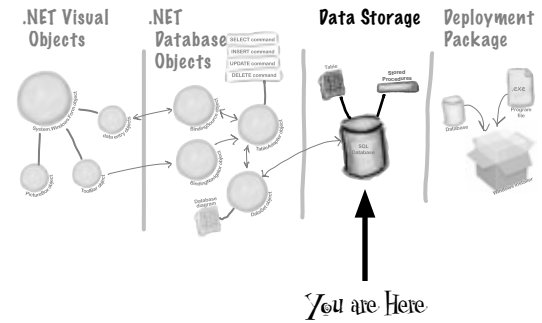
If you're not using the **Express** edition, you'll see "**Server Explorer**" instead of "**Database Explorer**."

The **Visual Studio 2010 Professional** and **Team Foundation** editions don't have a **Database Explorer** window. Instead, they have a **Server Explorer** window, which does everything the **Database Explorer** does, but also lets you explore data on your network.

The IDE created a database

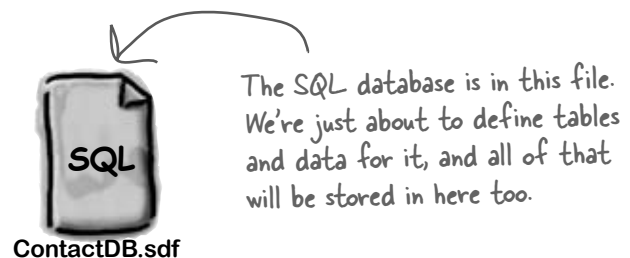
When you told the IDE to add a new SQL database to your project, the IDE created a new database for you. A **SQL database** is a system that stores data for you in an organized, interrelated way. The IDE gives you all the tools you need to maintain your data and databases.

Data in a SQL database lives in tables. For now, you can think of a table like a spreadsheet. It organizes your information into columns and rows. The columns are the data categories, like a contact's name and phone number, and each row is the data for one contact card.



SQL is its own language

SQL stands for **Structured Query Language**. It's a programming language for accessing data in databases. It's got its own syntax, keywords, and structure. SQL code takes the form of **statements** and **queries**, which access and retrieve the data. A SQL database can hold **stored procedures**, which are a bunch of SQL statements and queries that are stored in the database and can be run at any time. The IDE generates SQL statements and stored procedures for you automatically to let your program access the data in the database.

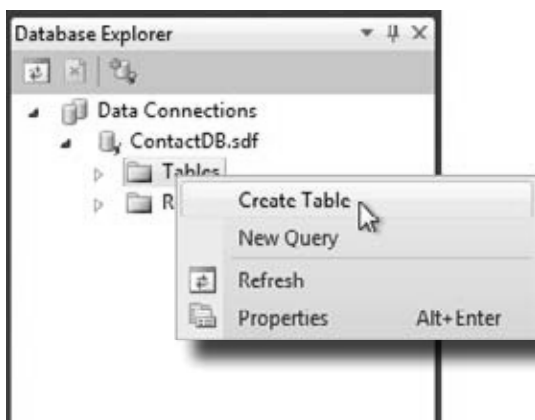


← [note from marketing: Can we get a plug for Head First SQL in here?]

Creating the table for the Contact List

We have a database, and now we need to store information in it. But our information actually has to go into a table, the data structure that databases use to hold individual bits of data. For our application, let's **create a table called "People"** to store all the contact information:

- 1 Add a table to the ContactDB database.**
Right-click on Tables in the Database Explorer, and select Create Table. This will open up a window where you can define the columns in the table you just created.



Now we need to add columns to our table. First, let's add a column called ContactID to our new People table, so that each Contact record has its own unique ID.

- 2 Add a ContactID column to the People table.**
Type "ContactID" in the Column Name field, and select Int from the Data Type drop-down box. Be sure to select "No" for Allow Nulls.

Finally, let's make this the primary key of our table. Highlight the ContactID column you just created, and click the Primary Key button. This tells the database that each entry will have a unique primary key entry.



Add a new column called "ContactID" with data type "int". Make sure to set "Allow Nulls" to No, "Unique" to Yes, and Primary Key to "Yes."

there are no Dumb Questions

Q: What's a column again?

A: A column is one field of a table. So in a People table, you might have a FirstName and LastName column. It will always have a data type, too, like String or Date or Bool.

Q: Why do we need this ContactID column?

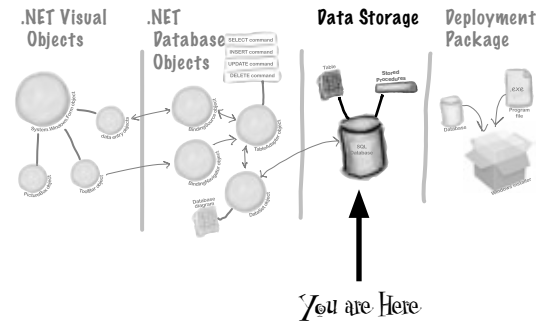
A: It helps to have a unique ID for each record in most database tables. Since we're storing contact information for individual people, we decided to create a column for that, and call it ContactID.

Q: What's that Int from Data Type mean?

A: The data type tells the database what type of information to expect for a column. Int stands for integer, which is just a whole number. So the ContactID column will have whole numbers in it.

Q: This is a lot of stuff. Should I be getting all of this?

A: No, it's OK if you don't understand everything right now. Your goal right now should be to start to get familiar with the basics of using the Visual Studio IDE to lay out your form and run your program. (If you're dying to know more about databases, you can always pick up **Head First SQL**.)



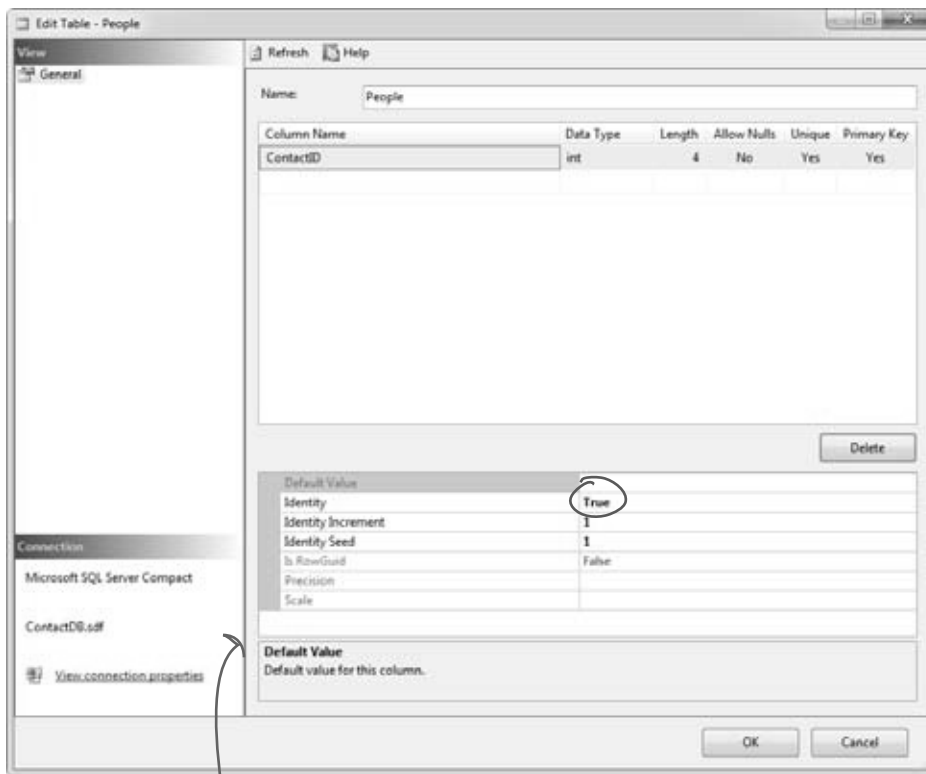
3 Tell the database to autogenerate IDs.

Since ContactID is a number for the database, and not our users, we can tell our database to handle creating and assigning IDs for us automatically. That way, we don't have to worry about writing any code to do this.

In the properties below your table, set Identity to "True" to make ContactID an identity column for your table.

And make sure you specify the table name "People" in the Name box at the top of the window.

This window is what you use to define your table and the data it will store.



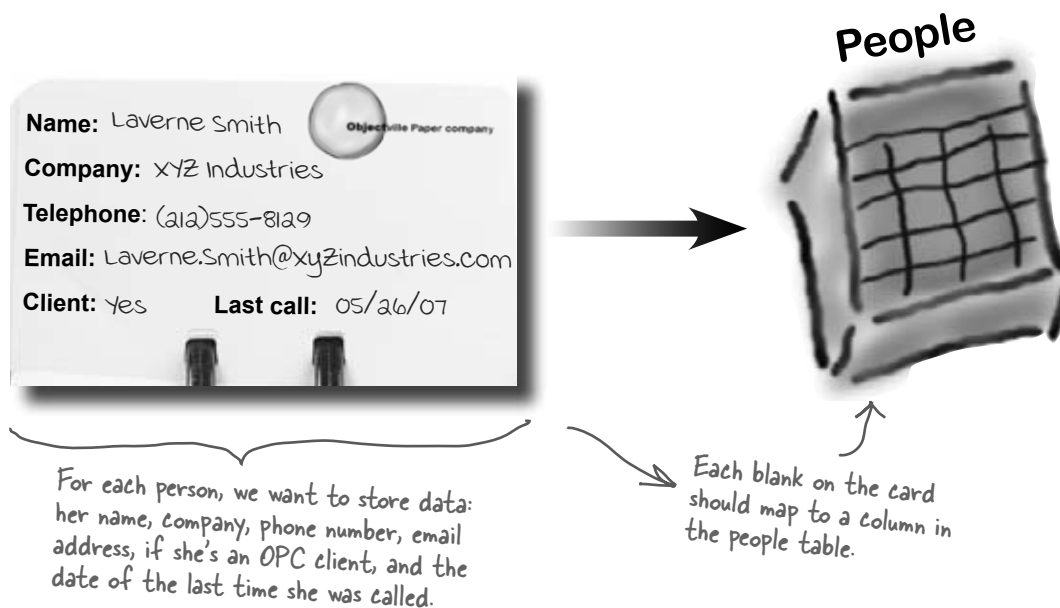
A primary key helps your database look up records quickly. Since the primary key is the main way your program will locate records, it always needs to have a value.

This will make it so that the ContactID field updates automatically whenever a new record is added.

You'll need to click on the right column and select "True" from the drop-down next to Identity to designate ContactID as your table's record identifier.

The blanks on the contact card are columns in our People table

Now that you've created a primary key for the table, you need to define all of the fields you're going to track in the database. Each field on our written contact card should become a column in the People table.



What kinds of problems could result from having multiple rows stored for the same person?

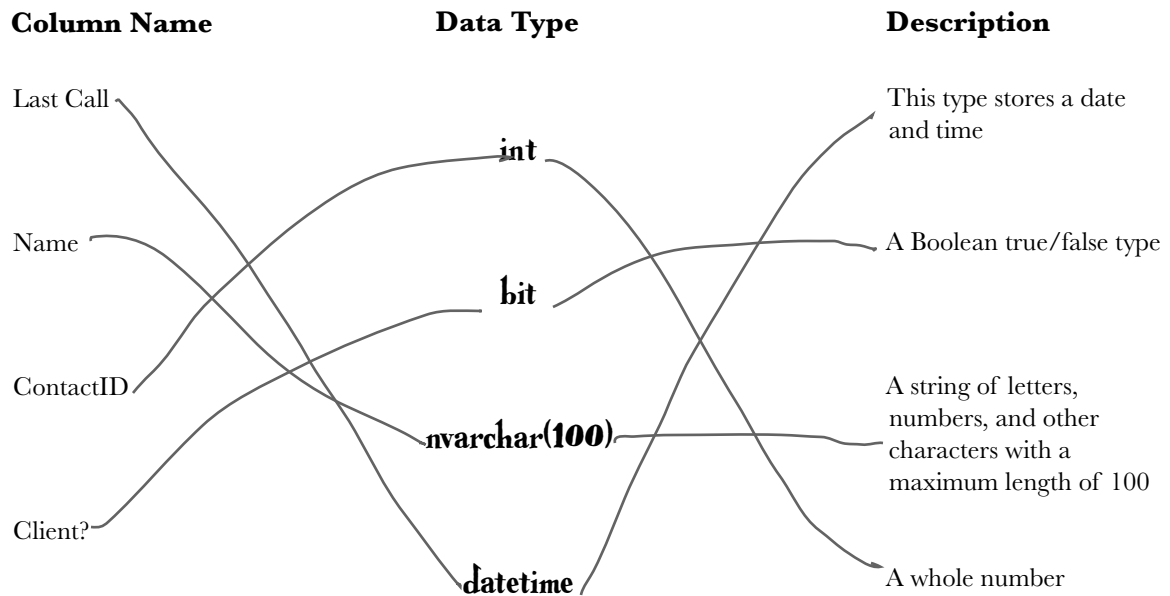
WHO DOES WHAT?

Now that you've created a People table and a primary key column, you need to add columns for all of the data fields. See if you can work out which data type goes with each of the columns in your table, and also match the data type to the right description.

Column Name	Data Type	Description
Last Call	int	This type stores a date and time
Name	bit	A Boolean true/false type
ContactID	nvarchar(100)	A string of letters, numbers, and other characters with a maximum length of 100
Client?	datetime	A whole number

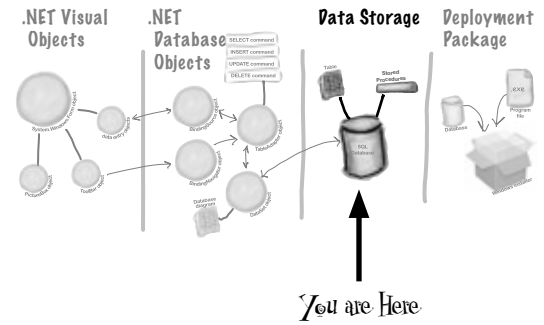
* WHO DOES WHAT? *

Now that you've created a People table and a primary key column, you need to add columns for all of the data fields. See if you can work out which data type goes with each of the columns in your table, and also match the data type to the right description.



Finish building the table

Go back to where you entered the ContactID column and add the other five columns from the contact card. Here's what your database table should look like when you're done:



Name:

Column Name	Data Type	Length	Allow Nulls	Unique	Primary Key
ContactID	int	4	No	Yes	Yes
Name	nvarchar	100	Yes	No	No
Company	nvarchar	100	Yes	No	No
Telephone	nvarchar	100	Yes	No	No
Email	nvarchar	100	Yes	No	No
Client	bit	1	Yes	No	No
LastCall	datetime	8	Yes	No	No

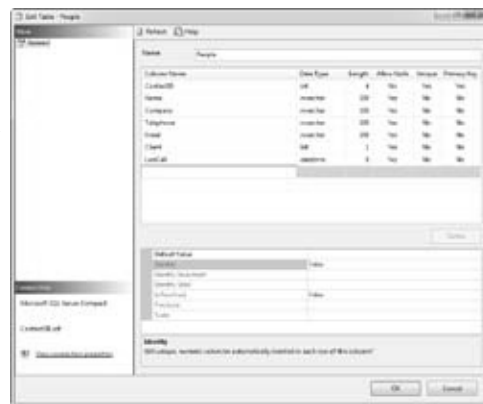
Bit fields hold True or False values and can be represented as a checkbox.

If you set Allow Nulls to No, the column must have a value.

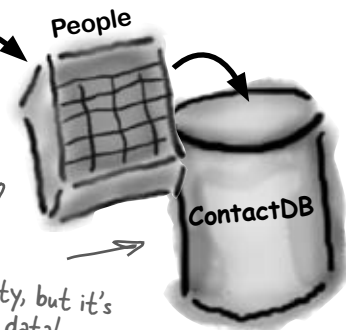
Some cards might have some missing information, so we'll let certain columns be blank.

Click on the OK button to save your new table. This will add an empty table to your database. (If you clicked OK earlier, you can open the Edit Table window again by right-clicking on the table in the Database Explorer and choosing "Edit Table Schema" from the menu.)

Once you click OK, Visual Studio adds a new People table to the database.



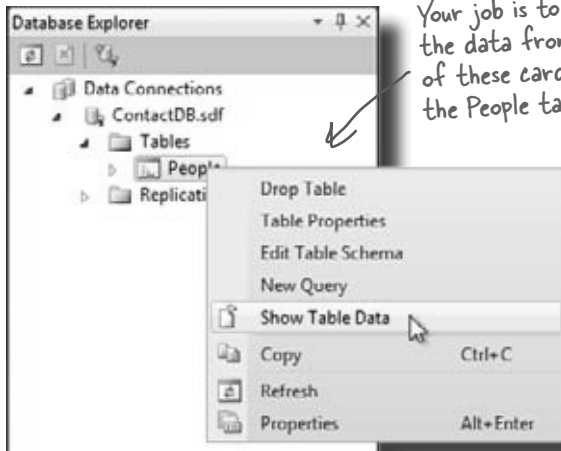
This new table is empty, but it's ready for you to add data!



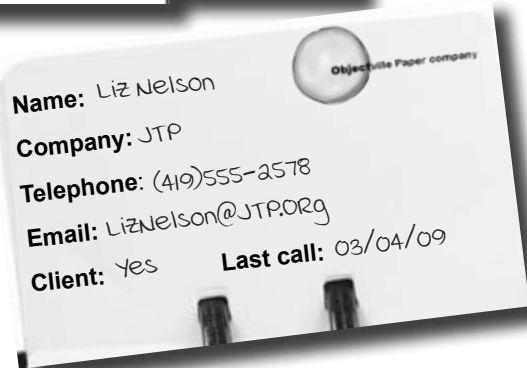
Insert your card data into the database

Now you're ready to start entering cards into the database. Here are some of the boss's contacts—we'll use those to set up the database with a few records.

- 1 Expand Tables and then right-click on the People table in the Database Explorer (or Server Explorer) and select Show Table Data.
- 2 Once you see the Table grid in the main window, go ahead and add all of the data below. (You'll see all null values at first—just type over them when you add your first row. And ignore the exclamation points that appear next to the data.) You don't need to fill in the ContactID column; that happens automatically.



Type "True" or "False" in the Client column. That'll get translated to the way SQL stores yes or no info.





Objectville Paper Company is in the United States, so the CEO writes dates so that 05/26/10 means May 26, 2010. If your machine is set to a different location, you may need to enter dates differently; you might need to use 26/05/10 instead.

- ③ Once you've entered all six records, select Save All from the File menu again. That should save the records to the database.

"Save All" tells the IDE to save everything in your application. That's different from "Save", which just saves the file you're working on.

there are no Dumb Questions

Q: So what happened to the data after I entered it? Where did it go?

A: The IDE automatically stored the data you entered into the People table in your database. The table, its columns, the data types, and all of the data inside it is all stored in the SQL Server Compact database file, ContactDB.sdf. That file is stored as part of your project, and the IDE updates it just like it updates your code files when you change them.

Q: OK, I entered these six records. Will they be part of my program forever?

A: Yes, they're as much a part of the program as the code that you write and the form that you're building. The difference is that instead of being compiled into an executable program, the ContactDB.sdf file is copied and stored along with the executable. When your application needs to access data, it reads and writes to ContactDB.sdf, in the program's output directory.

This file is actually a SQL database, and your program can use it with the code the IDE generated for you.



Connect your form to your database objects with a data source

We're finally ready to build the .NET database objects that our form will use to talk to your database. We need a **data source**, which is really just a collection of SQL statements your program will use to talk to the ContactDB database.

1 Go back to your application's form.

Close out the People table and the ContactDB database diagram. You should now have the Form1.cs [Design] tab visible.

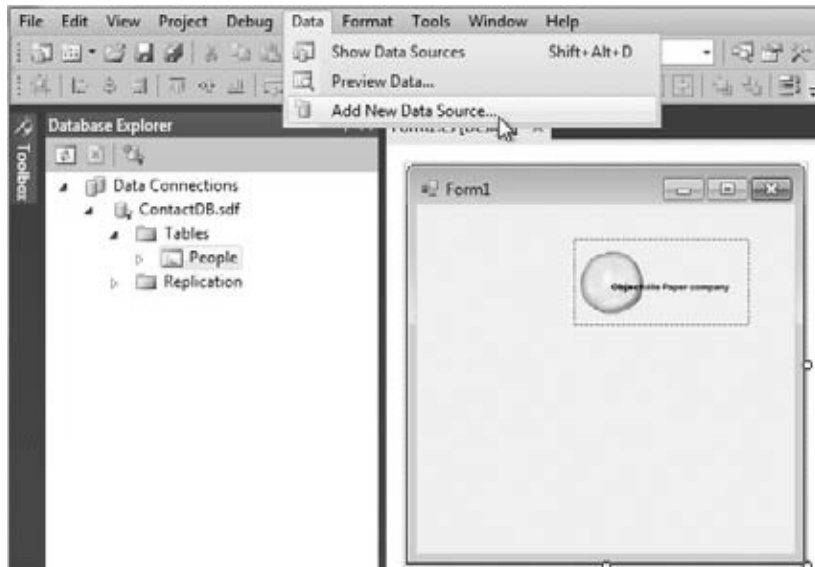
Once you're done entering data, close the data entry window to get back to your form.



ContactID	Name	Company	Telephone	Email	Client	LastCall
1	Lloyd Jones	Black Box Inc.	(718)555 5638	LJones@xblack...	True	5/26/2010 12:00...
2	Lucinda Ericson	Ericson Events	(212)555-9523	lucy@ericson...	False	5/17/2010 12:00...
3	Liz Nelson	JTP	(419)555-2578	liznelson@JTP...	True	3/4/2009 12:00:...
4	Matt Franks	XYZ Industries	(212)555-8125	Matt.Franks@x...	True	5/26/2010 12:00...
5	Sarah Kalter	Kalter, Riddle a...	(614)555-5641	sarah@krs.org	False	12/10/2008 12:0...
6	Laverne Smith	XYZ Industries	(212)555-8129	Laverne.Smith...	True	4/11/2010 12:00...
**	NULL	NULL	NULL	NULL	NULL	NULL

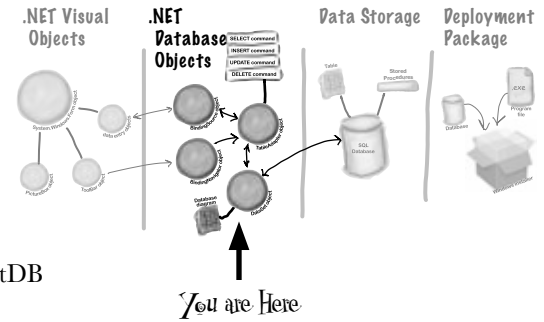
2 Add a new data source to your application.

This should be easy by now. Click the Data menu, and then select Add New Data Source... from the drop-down.



The data source you're creating will handle all the interactions between your form and your database.





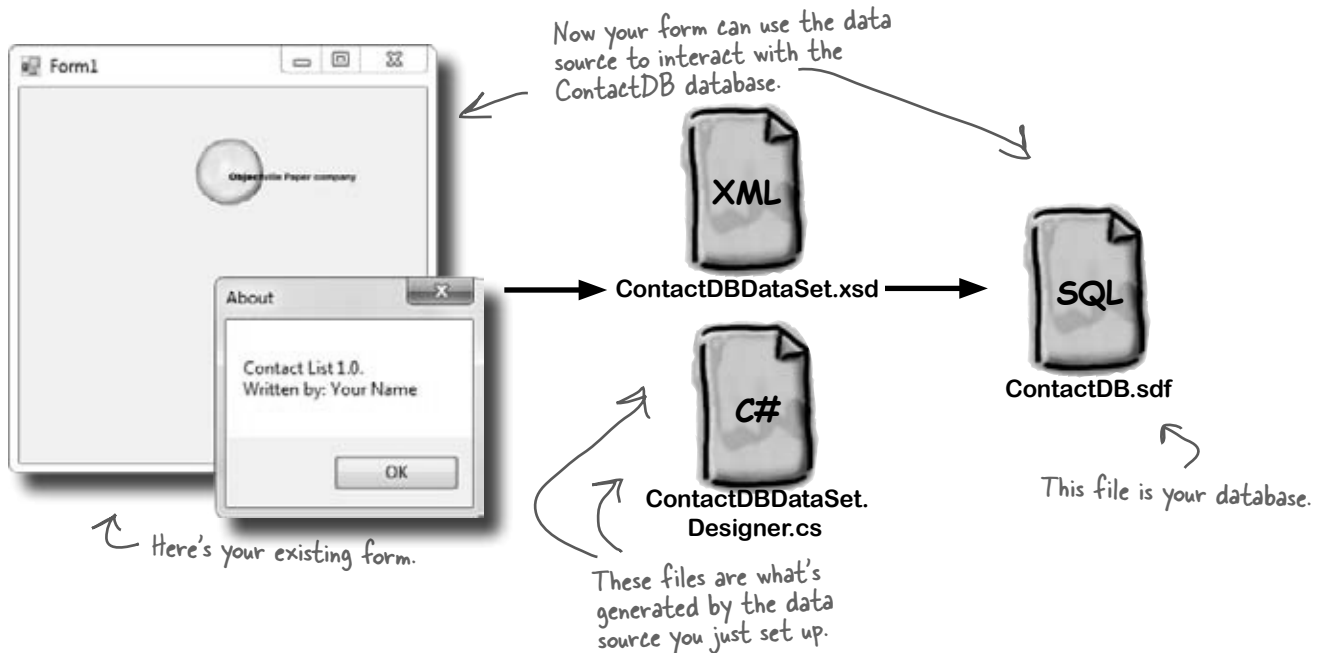
3 Configure your new data source.

Now you need to set up your data source to use the ContactDB database. Here's what to do:

- ★ Step 1: Choose a Data Source Type. Select **Database** and click the Next button.
- ★ Step 2: Choose a Database Model. Select **Dataset** and click the Next button.
- ★ Step 3: Choose Your Data Connection. You should see your Contact database in the drop-down. Click Next.
- ★ Step 4: Choose Your Database Objects. Click the **Tables** checkbox.
- ★ In the Dataset Name field, make sure it says "ContactDBDataSet" and **click Finish**.

These steps connect your new data source with the People table in the ContactDB database.

In the non-Express editions, you may be asked to save the connection in the app config. Answer "Yes."



Add database-driven controls to your form

Now we can go back to our form and add some more controls. But these aren't just any controls—they are controls that are *bound* to our database and the columns in the People table. That just means that a change to the data in one of the controls on the form automatically changes the data in the matching column in the database.

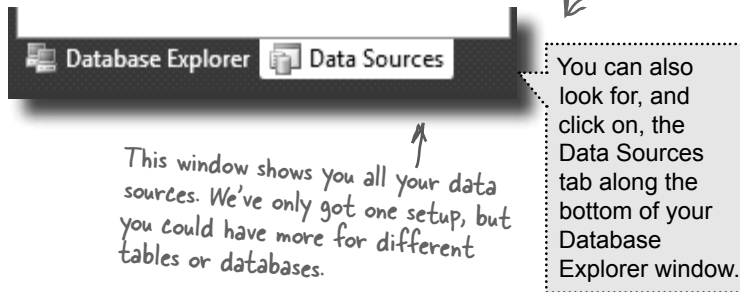
It took a little work, but now we're back to creating form objects that interact with our data storage.

Here's how to create several database-driven controls:

1 Select the data source you want to use.

Select Show Data Sources from the Data pull-down menu. This will bring up the Data Sources window, showing the sources you have set up for your application.

If you don't see this tab, select Show Data Sources from the Data menu.

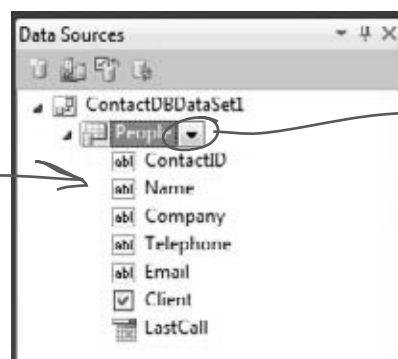


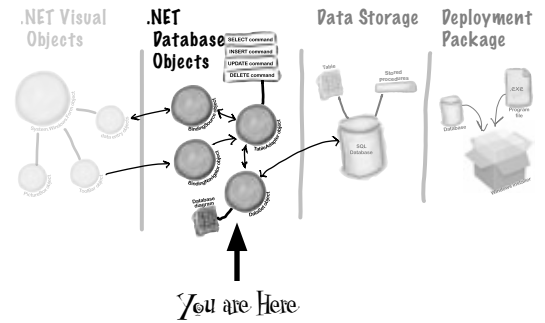
2 Select the People table.

Under the ContactDBDataSet, you should see the People table and all of the columns in it. Click the “expand” icon next to the People table to expand it—you'll see the columns that you added to your table. When you click on the People table in the Data Sources window and drag it onto your form, the IDE automatically adds data controls to your form that the user can use to browse and enter data. By default it adds a DataGridView, which lets the user work with the data using one big spreadsheet-like control. Click the arrow next to the People table and select Details—that tells the IDE to add individual controls to your form for each column in the table.

Click this arrow and choose Details to tell the IDE to add individual controls to your form rather than one large spreadsheet-like data control.

All of the columns you created should show up here.





3 Create controls that bind to the People table.

Drag and drop the People table onto your form in the form designer window. You should see controls appear for each column in your database. Don't worry too much about how they look right now; just make sure that they all appear on the form.

If you accidentally click out of the form you're working on, you can always get back to it by clicking the "Form1.cs [Design]" tab, or opening Form1.cs from the Solution Explorer.

The IDE creates this toolbar for navigating through the People table.

When you dragged the People table onto the form, a control was created for each column in the table.

This adapter allows your controls to interact with SQL commands that the IDE and data source generated for you.

These won't show up on your form, but represent the code that the IDE created to interact with the People table and ContactDB database.

This object connects the form to your People table.

The binding navigator connects the toolbar controls to your table.



Good programs are intuitive to use

Right now, the form works. But it doesn't look that great. Your application has to do more than be functional. It should be easy to use. With just a few simple steps, you can make the form look a lot more like the paper cards we were using at the beginning of the chapter.

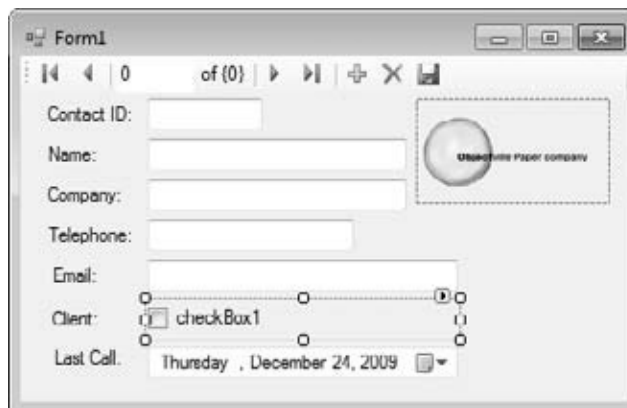
Our form would be more intuitive if it looked a lot like the contact card.



1 Line up your fields and labels.

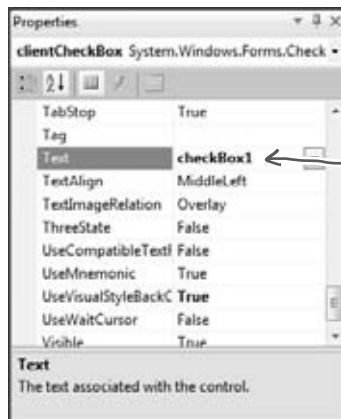
Line up your fields and labels along the left edge of the form. Your form will look like other applications, and make your users feel more comfortable using it.

Blue lines will show up on the form as you drag controls around. They're there to help you line the fields up.

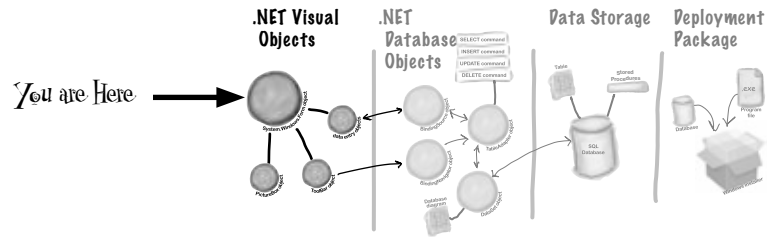


2 Change the Text Property on the Client checkbox.

When you first drag the fields onto the form, your Client checkbox will have a label to the right that needs to be deleted. Right below the Solution Explorer, you'll see the Properties window. Scroll down to the Text property and delete the "checkbox1" label.



Delete this word to make the label go away.



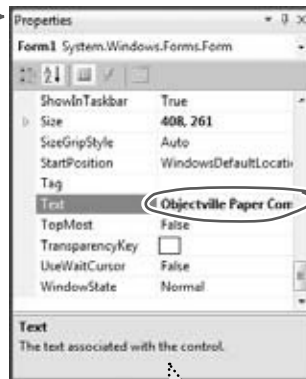
3 Make the application look professional.

You can change the name of the form by clicking on any empty space within the form, and finding the Text property in the Properties window of your IDE. Change the name of the form to *Objectville Paper Company Contact List*.

You can also turn off the Maximize and Minimize buttons in this same window, by looking for the MaximizeBox and MinimizeBox properties. Set these both to False.

The reason you want to turn off the Maximize button is that maximizing your form won't change the positions of the controls, so it'll look weird.

The Properties window should be right below Solution Explorer, in the lower right pane of your IDE.




The Text property controls the heading on your form's title bar.

If you don't have a Properties window, you can turn it on by selecting it from the View drop-down menu.

A good application not only works, but is easy to use. It's always a good idea to make sure it behaves as a typical user would expect it to.

ok, one last thing...

Test drive

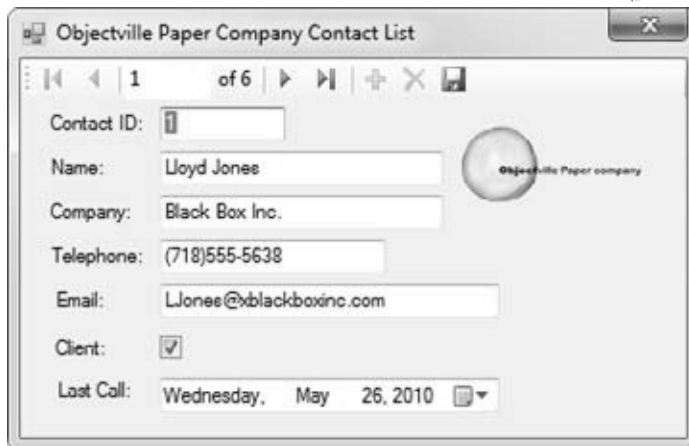
OK, just one more thing to do... run your program and make sure it works the way you think it should! Do it the same way you did before—press the F5 key on your keyboard, or click the green arrow button  on the toolbar (or choose “Run” from the Debug menu).

You can always run your programs at any time, even when they’re not done—although if there’s an error in the code, the IDE will tell you and stop you from executing it.

Click the X box in the corner to stop the program so you can move on to the next step.

Building your program overwrites the data in your database.

These controls let you page through the different records in the database.



We'll spend more time on this in the next chapter.

The IDE builds first, then runs

When you run your program in the IDE it actually does two things. First it **builds** your program, then it **executes** it. This involves a few distinct parts. It **compiles** the code, or turns it into an executable file. Then it places the compiled code, along with any resources and other files, into a subdirectory underneath the bin folder.

In this case, you'll find the executable and SQL database file in bin/debug. Since it copies the database out each time, any changes you make will be lost the next time you run inside the IDE. But if you run the executable from Windows, it'll save your data—until you build again, at which point the IDE will overwrite the SQL database with a new copy that contains the data you set up from inside the Database Explorer.



Watch it!

Every time you build your program, the IDE puts a fresh copy of the database in the bin

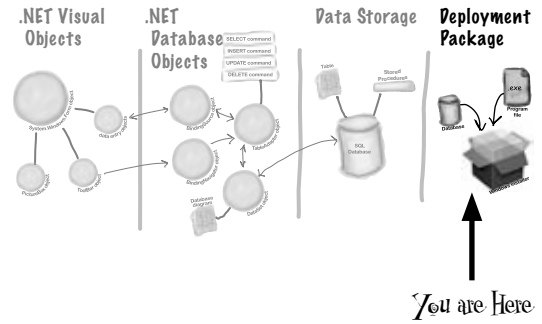
folder. This will overwrite any data you added when you ran the program.

When you debug your program, the IDE rebuilds it if the code has changed—which means that your database will sometimes get overwritten when you run your program in the IDE. If you run the program directly from the bin/debug or bin/release folder, or if you use the installer to install it on your machine, then you won't see this problem.

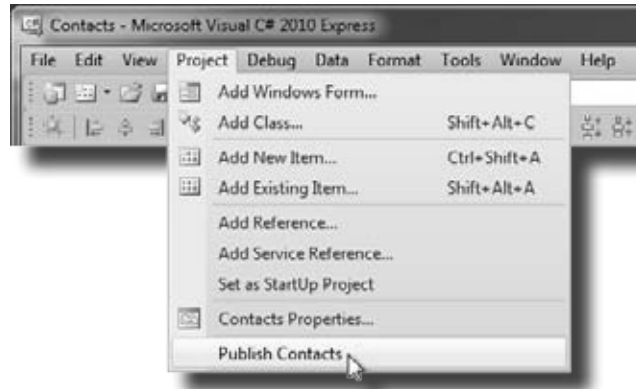
How to turn YOUR application into EVERYONE'S application

At this point, you've got a great program. But it only runs on your machine. That means that nobody else can use the app, pay you for it, see how great you are and hire you... and your boss and customers can't see the reports you're generating from the database.

C# makes it easy to take an application you've created, and **deploy** it. Deployment is taking an application and installing it onto other machines. And with the Visual C# IDE, you can set up a deployment with just two steps.



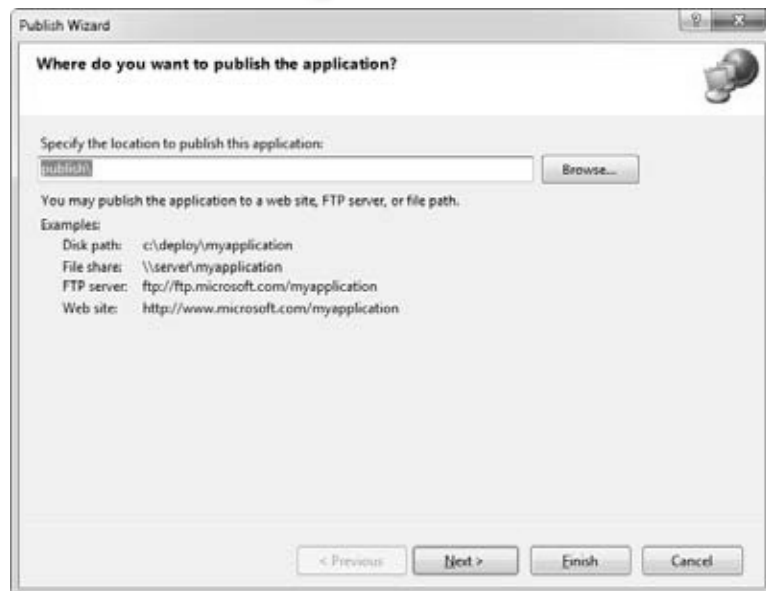
- 1 Select *Publish Contacts* from the Project menu.



Building the solution just copies the files to your local machine. Publish creates a Setup executable and a configuration file so that any machine could install your program.

- 2 Just accept all of the defaults in the Publish Wizard by clicking Finish. You'll see it package up your application and then show you a folder that has your Setup.exe in it.

If you're using Visual Studio Express, you'll find "Publish" in the Project menu, but in other editions it may be in the Build menu.



Give your users the application

Once you've created a deployment, you'll have a new folder called `publish/`. That folder has several things in it, all used for installation. The most important for your users is `setup`, a program that will let them install your program on their own computers.

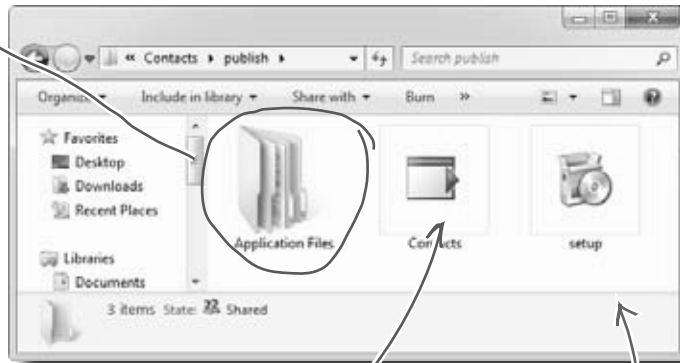


Watch it!

You may need to run the installer as administrator.

If SQL Server Compact isn't already installed on the machine, the installer will automatically download and install it. On some machines, this won't work unless you run the setup as administrator, so right-click on "setup" and choose "Run as administrator" to install it. If you don't have access to do that, don't worry! You don't need to in order to move forward in the book.

This is where all of the supporting files for the installer are stored.



This file tells the installer everything that needs to be included when the program is installed.

This is how your users will install the program on their computers!

My secretary just told me that you've got the new contact database working already. Pack your bags—we've got room on the jet to Aspen for a go-getter like you!

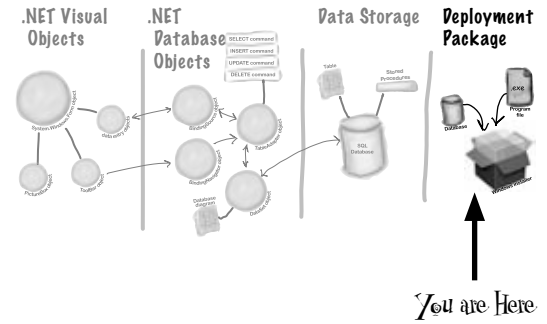


Sounds like the boss is pleased. Good job! There's just one more thing to do before you can jet off to the slopes, though...

You're NOT done: test your installation

Before you pop the cork on any champagne bottles, you need to test your deployment and installation. You wouldn't give anyone your program without running it first, would you?

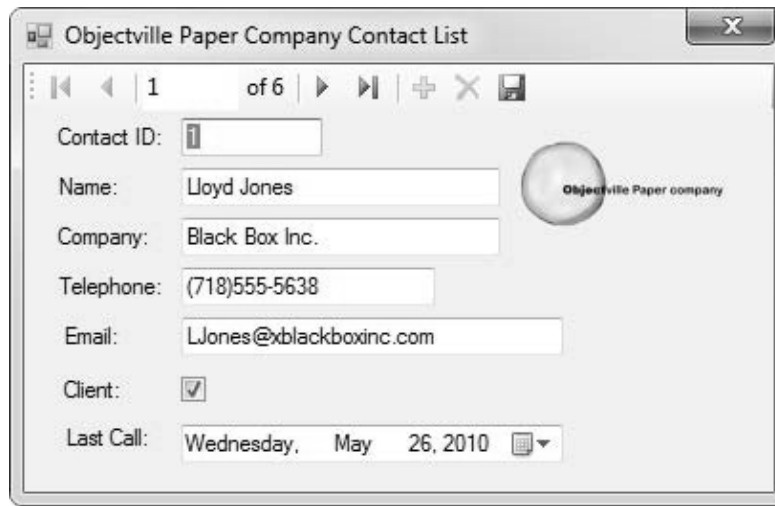
Close the Visual Studio IDE. Click the setup program, and select a location on your own computer to install the program. Now run it from there, and make sure it works like you expect. You can add and change records, too, and they'll be saved to the database.



Now you can make changes to the data, and they'll get saved to the database.

You can use the arrows and the text field to switch between records.

Go ahead...make some changes. You've deployed it so this time, they'll stick.



The contacts you entered are all there. They're part of the ContactDB.sdf database file, which gets installed along with your program.

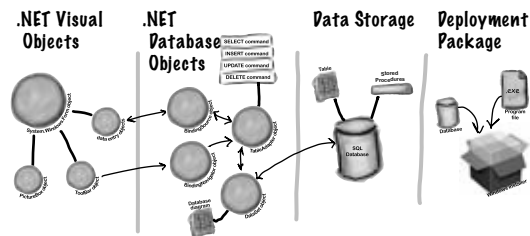
TEST EVERYTHING!

Test your program, test your deployment, test the data in your application.

super fast!

You've built a complete data-driven application

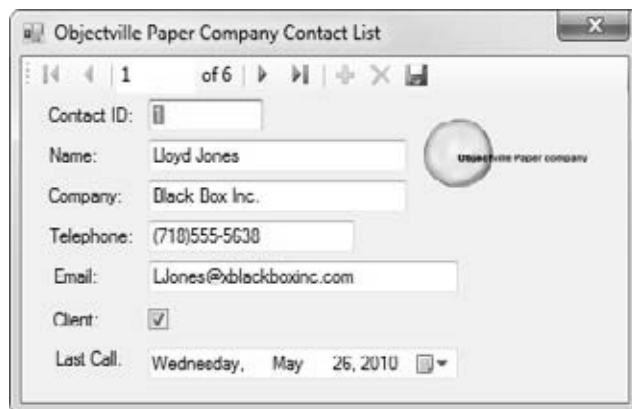
The Visual Studio IDE made it pretty easy to create a Windows application, create and design a database, and hook the two together. You even were able to build an installer with a few extra clicks.



From *this*



to *this*



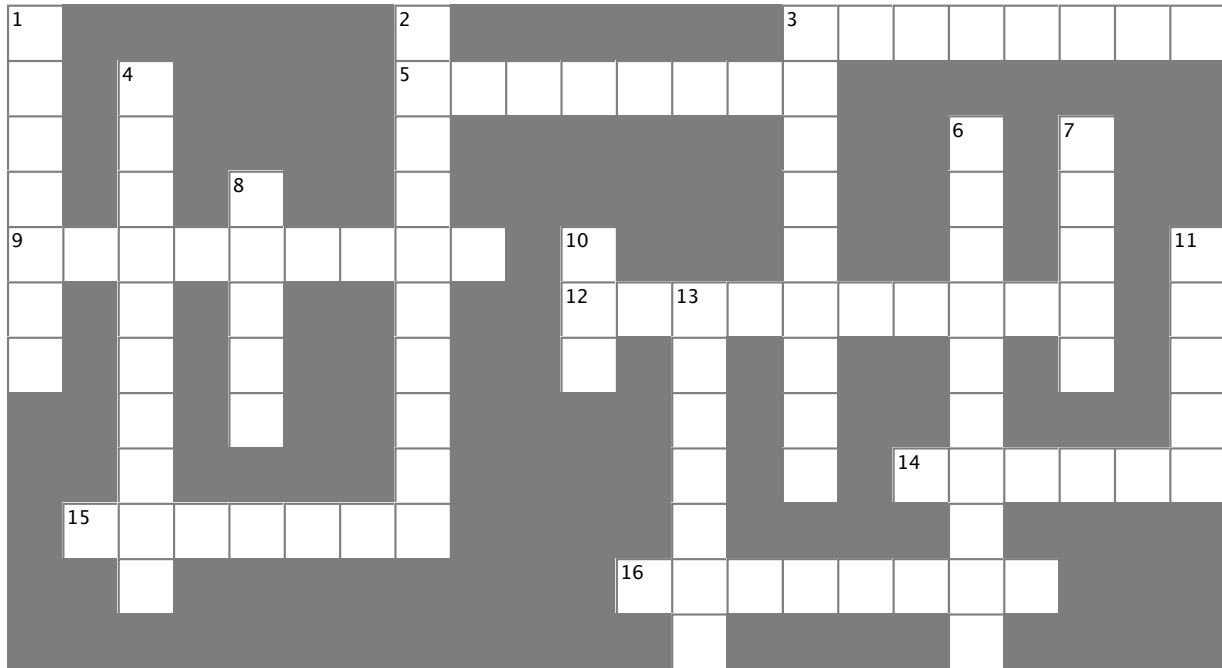
in no time flat.

The power of Visual C# is that you can quickly get up and running, and then focus on what your program's supposed to do...not lots of windows, buttons, and SQL access code.



C# Sharpcross

Take some time to sit back and exercise your C# vocabulary with this crossword; all of the solution words are from this chapter.



Across

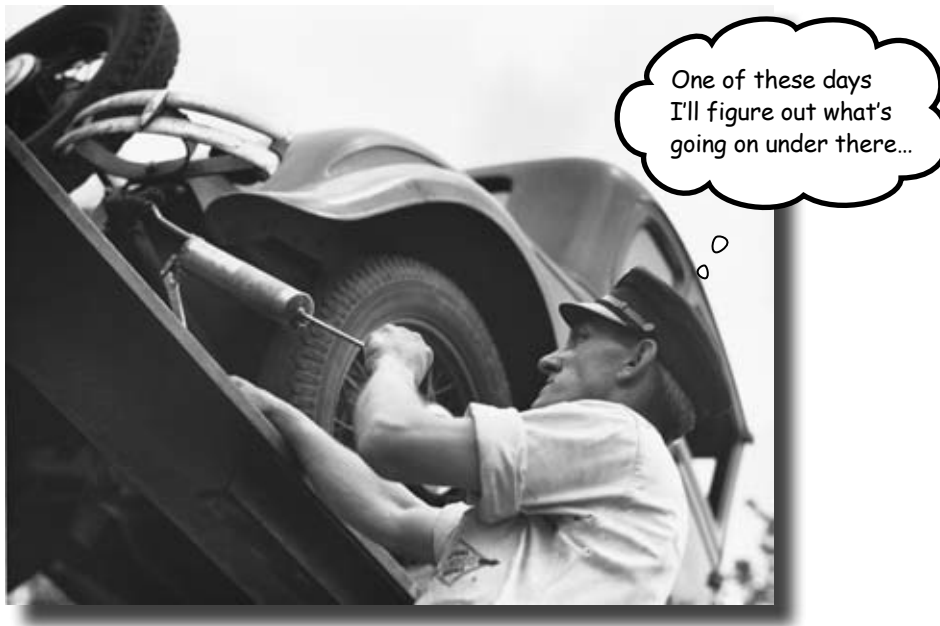
3. The _____ explorer is where you edit the contents of your SQL tables and bind them to your program
5. An image, sound, icon, or file that's attached to your project in a way that your code can access easily
9. You build one of these so you can deploy your program to another computer
12. What the "I" in IDE stands for
14. When you double-clicked on a control, the IDE created this for you and you added code to it
15. Every row contains several of these, and all of them can have different data types
16. The _____ Explorer shows you all of the files in your project

Down

1. What's happening when code is turned into an executable
2. What you change to alter the appearance or behavior of controls on your form
3. What you're doing when you run your program from inside the IDE
4. The "About" box in the Objectville Paper Company Contact List program was one of these
6. You displayed the Objectville Paper Company logo with one of these
7. Before you start building a program, you should always think about users and their _____
8. A database can use many of these to store data
10. The data type in a SQL database that you use to store true/false values
11. Before you can run your program, the IDE does this to create the executable and move files to the output directory
13. You drag controls out of this and onto your form

2 it's all just code

* *Under the hood* *



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

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

When you're doing this...

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

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

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

1 Creating a Windows Forms Application project

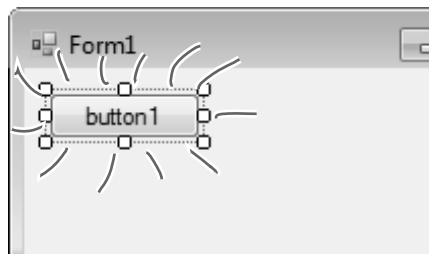
There are several kinds of applications the IDE lets you build, but we'll be concentrating on Windows Forms applications for now. Those are programs that have visual elements, like forms and buttons.

Make sure you always create a Windows Forms Application project—that tells the IDE to create an empty form and add it to your new project.



2 Dragging a button out of the toolbox and onto your form, and then double-clicking it

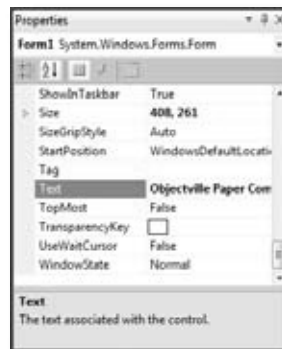
Buttons are how you make things happen in your form. We'll use a lot of buttons to explore various parts of the C# language. They're also a part of almost every C# application you'll write.



3 Setting a property on your form

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 form, attributes of your databases, and even options on your project itself.

The Properties window in the IDE is a really easy way to edit a specific chunk of code in `Form1.Designer.cs` automatically. It would take a lot longer to do it by hand. Use the **F4** 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.



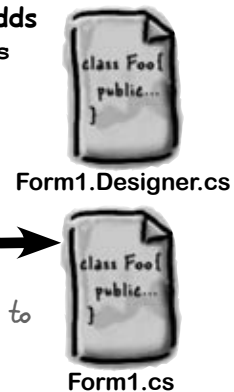
These files are created from a predefined template that contains the basic code to create and display a form.

- 2 ...the IDE adds code to the Form1.Designer.cs file that adds the button to the form, and then adds code to the Form1.cs file to handle the button click.

```
private void button1_Click(object sender, EventArgs 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.



This code gets added to Form1.cs.

- 3 ...the IDE opens the Form1.Designer.cs file and updates a line of code.

```
partial class Form1
{
:
:
this.Text = "Objectville Paper Company Contact List";
:
}

```

The IDE went into this file...



...and updated this line of code.

Where programs come from

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

Every program starts out as source code files

You’ve already seen how to edit a program, and how the IDE saves your program to files in a folder. Those files **are** your program—you can copy them to a new folder and open them up, and everything will be there: forms, 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.



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

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. Remember that Maximize button you turned off for the Contacts form? When you click the Maximize button on a window, there’s code that tells the window how to maximize itself and take up the whole screen. That code is part of the .NET Framework. Buttons, checkboxes, lists... those are all pieces of the .NET Framework. So are the internal bits that hooked your form up to the database. It’s got tools to 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.

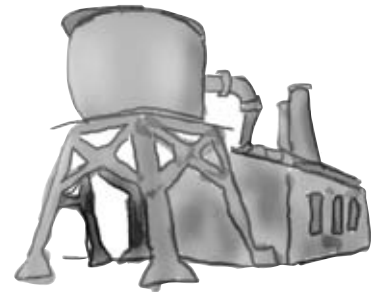


The tools in the .NET Framework are divided up into **namespaces**. You’ve seen these namespaces before, at the top of your code in the “using” lines. One namespace is called `System.Windows.Forms`—it’s where your buttons, checkboxes, and forms come from. Whenever you create a new Windows Forms Application project, the IDE will add the necessary files so that your project contains a form, and those files have the line “using `System.Windows.Forms;`” at the top.

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 what you double-click on to run your program. 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 the folder you’re publishing to.

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.



Your program runs inside the CLR

When you double-click on the executable, Windows runs your program. But there’s an extra “layer” between Windows 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.



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

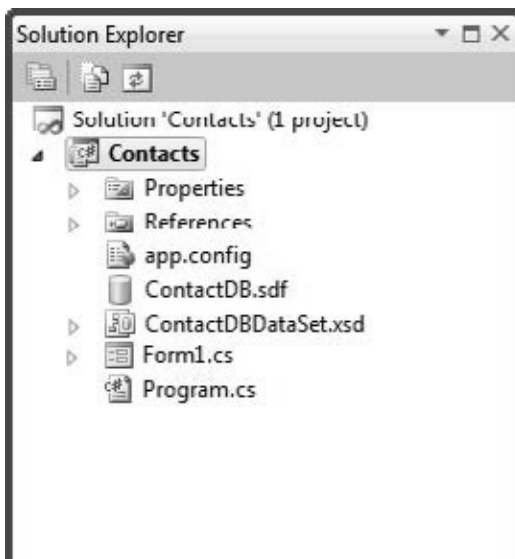
The IDE helps you code

You've already seen a few of the things that the IDE can do. Let's take a closer look at some of the tools it gives you.



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 looks like after creating the Objectville Paper Company Contact List program:



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



Use the tabs to switch between open files

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



Here's the form's resource file that you added the Objectville Paper Company logo to.

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

★ **The IDE helps you write code**

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

MessageBox.

- ◆ Equals
- ◆ ReferenceEquals
- ◆ Show

The IDE knows that `MessageBox` has three methods called `Equals`, `ReferenceEquals`, and `Show`. If you type `S`, it selects `Show`. 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 `Show` and type `(`, the IDE's IntelliSense will show you information about how you can complete the line:

This means that there are 21 different ways that you can call the `MessageBox`'s `Show` method (like ways to display different buttons or icons).

MessageBox.Show(

▲ 3 of 21 ▼ System.Windows.Forms.DialogResult MessageBox.Show(string text, string caption)
 Displays a message box with specified text and caption.
text: The text to display in the message box.

The IDE also has shortcuts called **snippets** that let you type an abbreviation to tell it to fill in the rest of the code. Here's a useful one: type `mbox` and press the Tab key twice, and the IDE will fill in the `MessageBox.Show` method for you:

```
MessageBox.Show("Test");
```

When you use Start Debugging 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:

A missing semicolon at the end of a statement is one of the most common errors that keeps your program from building!

Error List					
2 Errors 0 Warnings 0 Messages					
	Description	File	Line	Column	Project
1	'System.Windows.Forms.MessageBox' does not contain a definition for 'XYZ'	Form1.cs	45	74	Contacts
2	; expected	Form1.cs	45	33	Contacts

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

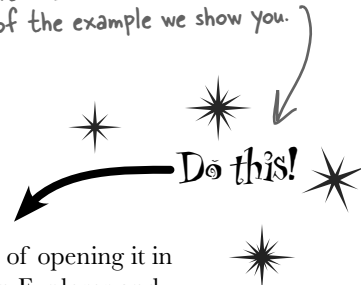
```
private void pictureBox1_Click(object sender, EventArgs e)
{
    MessageBox.XYZ("hi")
}
```

The IDE will show a red underscore to show you that there's an error.

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.

When you see a "Do this!", pop open the IDE and follow along. We'll tell you exactly what to do, and point out what to look for to get the most out of the example we show you.



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** onto a new form.

3 Find and expand the designer-generated code for the PictureBox control

Then go back to the `Form1.Designer.cs` tab in the IDE. Scroll down and look for this line in the code:

Click on the plus sign

```
Windows Form Designer generated code
```

Click on the + on the left-hand side of the line to expand the code. Scroll down and find these lines:

```
//  
// pictureBox1  
//  
this.pictureBox1.Location = new System.Drawing.Point(276, 28);  
this.pictureBox1.Name = "pictureBox1";  
this.pictureBox1.Size = new System.Drawing.Size(100, 50);  
this.pictureBox1.TabIndex = 0;  
this.pictureBox1.TabStop = false;
```

Don't worry if the numbers in your code for the Location and Size lines are a little different than these...

Wait, wait! What did that say?

Scroll back up for a minute. There it is, at the top of the Windows Form Designer-generated code section:

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

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, 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?

Most comments only start with two slashes (`//`). But the IDE sometimes adds these three-slash comments.

These are XML comments, and you can use them to document your code. Flip to "Leftovers" section #1 in the Appendix of this book to learn more about them.

You don't have to save the form or run the program to see the changes. Just make the change in the code editor, and then click on the tab labeled "Form1.cs [Design]" to flip over to the form designer—the changes should show up immediately.

It's always easier to use the IDE to change your form's Designer-generated code. But when you do, any change you make in the IDE ends up as a change to your project's code.

your program makes a statement

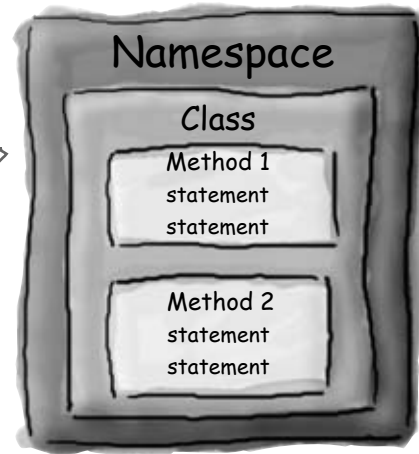
Anatomy of a program

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.

Every time you make a new program, you define a namespace for it so that its code is separate from the .NET Framework classes.



Let's take a closer look at your code

Open up the code from your Contacts project's `Form1.cs` so we can go through it piece by piece.

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 to use. If you use other classes that are in other namespaces, then you'll add `using` lines for them, too. Since forms often use a lot of different tools from the .NET Framework, the IDE automatically adds a bunch of `using` lines when it creates a form and adds it to your project.

```
using System;  
using System.Collections.Generic;  
using System.ComponentModel;  
using System.Data;  
using System.Drawing;  
using System.Linq;  
using System.Text;  
using System.Windows.Forms;
```

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

One thing to keep in mind: you don't actually *have* to use a `using` statement. You can always use the fully qualified name. So if you leave out `using System.Windows.Forms`, you can still show a message box by calling `System.Windows.Forms.MessageBox.Show()`, and the compiler will know what namespace you're talking about.

2 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 Form1 that displays a form.

namespace Contacts

{

public partial class Form1 : Form

{

When you called your program Contacts, the IDE created a namespace for it called Contacts by adding the namespace keyword at the top of your code file. Everything inside its pair of curly brackets is part of the Contacts namespace.

This is a class called Form1. It contains all of the code to draw the form and the Toolbox controls on it. The IDE created it when you told it to create a new Windows Forms Application project.

3 Classes contain methods that perform actions

When a class needs to do something, it uses a **method**. A method takes an 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.

Look for the matching pairs of brackets. Every { is eventually paired up with a }. Some pairs can be inside others.

public Form1 ()

{

InitializeComponent ();

}

This line calls a method named InitializeComponent(), which the IDE also created for you.

4 A statement performs one single action

When you added the `MessageBox.Show ()` line to your program, you were adding a **statement**. 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.

This is a method called `pictureBox1_Click()` that gets called when the user clicks on the picture box.

This method has two parameters called `sender` and `e`.

private void pictureBox1_Click(object sender, EventArgs e)

{

MessageBox.Show("Contact List 1.0", "About");

}

This is a **statement**. You already know what it does—it pops up a little message box window.

Your statement called the `Show()` method, which is part of the `MessageBox` class, which is inside the `System.Windows.Forms` namespace.

Your statement passed two parameters to the `Show()` method. The first one was a string of text to display in the message box, and the second one was a string to display in its title bar.

}

Your program knows where to start

When you created the new Windows Application solution, 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.

Every C# program can only have one entry point method, and it's always called **Main()**. That's how it knows where to start when you run it.



Your Code Up Close

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



```
1 using System;
using System.Linq;
using System.Collections.Generic;
using System.Windows.Forms;

2 namespace Contacts
{
3     static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            4 Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

The namespace for all this code is **Contacts**. We'll talk about namespaces more in a few pages.

Lines that begin with two or more slashes are comments, which you can add anywhere you want. The slashes tell C# to ignore them.

Every time you run your program, it starts here, at the entry point.

This statement creates and displays the **Contacts** form, and ends the program when the form's closed.

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

Remember, this is just a starting point for you to dig into the code. But before you do, you'll need to know what you're looking at.

1 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.

Your programs will use more and more namespaces like this one as you learn about C# and .NET's other built-in features throughout the book.

If you didn't specify the "using" line, you'd have to explicitly type out `System.Windows.Forms` every time you use anything in that namespace.

2 The IDE chose a namespace for your code.

Here's the namespace the IDE created for you—it chose `Contacts` based on your project's name. All of the code in your program lives in this namespace.

Namespaces let you use the same name in different programs, as long as those programs aren't also in the same namespace.

3 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 `Contacts` form.

You can have multiple classes in a single namespace.

4 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 `Contacts` form. Methods are where the action happens—every method **does** something.

Technically, a program can have more than one `Main()` method, and you can tell C# which one is the entry point... but you won't need to do that now.

5 Each program has a special kind of method called the entry point.

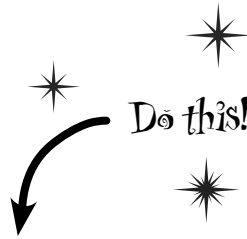
Every C# program **must** have exactly one method called `Main`. Even though your program has a lot of methods, only one can be the first one that gets executed, and that's your `Main` method. C# checks every class in your code for a method that reads `static void Main()`. Then, when the program is run, the first statement in this method gets executed, and everything else follows from that first statement.

Every C# program 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. **Open up the program you wrote in Chapter 1**, remove the `Main` method in `Program.cs`, and create a new entry point.



- 1 Go back to `Program.cs` and change the name of the `Main` method to `NotMain`. Now **try to build and run** the program. What happens?

Write down what happened when you changed the method name, and why you think that 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:

Right-click on the project in Properties and select "Add" and "Class..."

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;

namespace Contacts
{
    class AnotherClass
    {
    }
}
```

These four standard using lines were added to the file.

This class is in the same Contacts namespace that the IDE added when you first created the Windows Application project.

The IDE automatically named the class based on the filename.

- 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:

MessageBox is a class that lives in the System.Windows.Forms namespace, which is why you had to add the using line in step #3. Show() is a method that's part of the MessageBox class.

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




So what happened?

Instead of popping up the Contacts application, 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.

- 5 Figure out how to fix your program so it pops up Contacts again.

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

Sharpen your pencil

Fill in the annotations so they describe the lines in this C# file that they're pointing to. We've filled in the first one for you.

```
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

C# classes have these "using" lines to add methods from other namespaces

```
namespace SomeNamespace .....
{
    class MyClass {
        public static void DoSomething() {
            MessageBox.Show("This is a message");
        }
    }
}
```

there are no Dumb Questions

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: 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()`.

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.

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

Sharpen your pencil Solution

Fill in the annotations so they describe the lines in this C# file that they're pointing to. We've filled in the first one for you.

```
using System;  
using System.Linq;  
using System.Text;  
using System.Windows.Forms;
```

C# classes have these "using" lines to add methods from other namespaces.

```
namespace SomeNamespace
```

All of the code lives in classes, so the program needs a class here.

```
{
```

```
    class MyClass {
```

This class has one method. Its name is "DoSomething," and when it's called it pops up a MessageBox..

```
        public static void DoSomething() {
```

```
            MessageBox.Show("This is a message");
```

This is a statement. When it's executed, it pops up a little window with a message inside of it.

```
        }
```

```
    }
```

```
}
```

WHAT'S MY PURPOSE?

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

```
partial class Form1
{
    :
    this.BackColor = Color.DarkViolet;
    :
}
```

Set properties for a label

```
// 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

```
partial class Form1
{
    private void InitializeComponent()
    {
        :
    }
}
```

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

```
number_of_pit_stopsLabel.Name
    = "number_of_pit_stopsLabel";
number_of_pit_stopsLabel.Size
    = new System.Drawing.Size(135, 17);
number_of_pit_stopsLabel.Text
    = "Number of pit stops:";
```

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 the `Form1` window

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

A block of code that executes whenever a program opens up a `Form1` window

* WHAT'S MY PURPOSE? *

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

```
partial class Form1
{
    :
    this.BackColor = Color.DarkViolet;
    :
}
```

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

```
partial class Form1
{
    private void InitializeComponent()
    {
        :
    }
}
```

```
number_of_pit_stopsLabel.Name
    = "number_of_pit_stopsLabel";
number_of_pit_stopsLabel.Size
    = new System.Drawing.Size(135, 17);
number_of_pit_stopsLabel.Text
    = "Number of pit stops:";
```

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

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

Set properties for a label

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

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

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

Change the background color of the Form1 window

A block of code that executes whenever a program opens up a Form1 window

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 class is "public" it means every other class in the program can access its methods.

MoreClasses.cs

```
namespace PetFiler2 {

    class Fish {

        public void Swim() {
            // statements
        }

    }

    partial class Cat {

        public void Purr() {
            // statements
        }

    }

}
```

SomeClasses.cs

```
namespace PetFiler2 {

    class Dog {

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

    }

    partial class Cat {

        public void Meow() {
            // more statements
        }

    }

}
```

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

You can only split a class up into different files if you use the `partial` keyword. You probably won't do that in any of the code you write in this book, but the IDE used it to split your form up into two files, `Form1.cs` and `Form1.Designer.cs`.

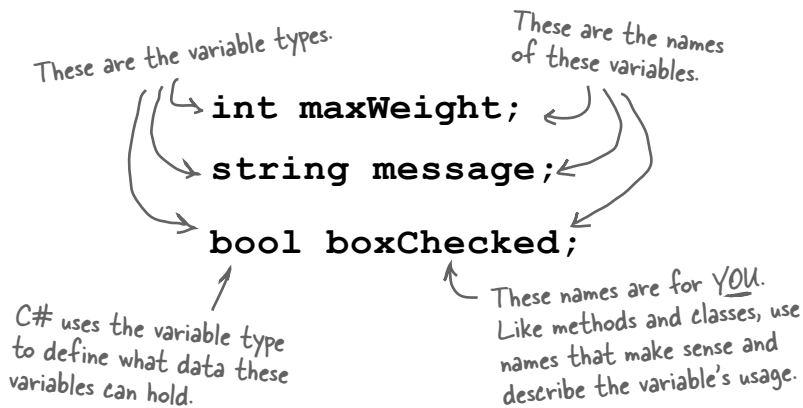
There's more to namespaces and class declarations, but you won't need them for the work you're doing right now. Flip to #2 in the "Leftovers" appendix to read more.

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.



Variables vary

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

```
int myHeight = 63;
```

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

```
myHeight = 12;
```

C# will replace `myHeight` with 12—but the variable is still called `myHeight`.



Watch it!

Are you already familiar with another language?

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

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:

```
int z;
MessageBox.Show("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:

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

These values are assigned to the variables.

Each declaration has a type, exactly like before.

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

A few useful types

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

var-i-a-ble, adjective.
able to be changed or adapted.
*The drill's **variable** speed bit let Bob change the drill speed from slow to fast based on the job he had to do.*

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

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:

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

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

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

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.

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

```
number *= 3;
number = 71 / 3;
```

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.

```
int count = 0;
```

```
count ++;
count --;
```

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

This MessageBox will pop up a box that says "hello again hello"

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

```
result += " again " + result;
```

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

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

```
MessageBox.Show(result);
```

```
result = "the value is: " + count;
```

```
result = "";
```

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

```
bool yesNo = false;
```

```
bool anotherBool = true;
```

```
yesNo = !anotherBool;
```



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 Windows Forms Application project

Drag a button onto your form and double-click it. Enter all of the code on the previous page. Then take a look at the comments in the screenshot below:

```

private void button1_Click(object sender, EventArgs e)
{
    // Here's a great way to use the IDE to see how this code works!
    //
    // First, create a new project in the IDE add a button. Next, double-click on the
    // button so the IDE adds a button1_Click method to your program. Fill in that
    // method with all of the code, starting with "int number = 15;".
    //
    // Now put a breakpoint on the first line by right-clicking on it and choosing
    // "Breakpoint >> Insert Breakpoint". The line should turn red.
    //
    // Next, start debugging your program. You'll see it break on the line where you
    // inserted the breakpoint. Your program's just paused! If you click the Run
    // toolbar button (or hit F5), it will continue. Right-click on "number" and
    // choose "Expression: 'number' >> Add Watch" from the menu. The bottom panel in
    // the IDE should change to the Watches window, and there should be a line in that
    // window for "number". Step through the program line by line using Step Over (F10).
    // You can see the value of the "number" variable change as you go!
    //
    // Do the same for the count, result, yesNo, and anotherBool variables.

    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;
    MessageBox.Show(result);
    result = "the value is: " + count;
    result = "";

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

```

Debug this!

Creating a new Windows Forms Application project will tell the IDE to create a new project with a blank form and an entry point. You might want to name it something like "Chapter 2 program 1"—you'll be building a whole lot of programs throughout the book.

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

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

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.

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`.)

Flip the page and keep going!

stop bugging me!

3 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 pop up the form.

4 Click on the button to trigger the breakpoint

As soon as your program gets to the line of code that has the breakpoint, the IDE automatically brings up the code editor and highlights the current line of code in yellow.

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

5 Add a watch for the number variable

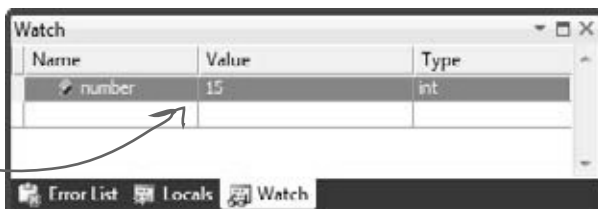
Right-click on the number variable (any occurrence of it will do!) and choose **Expression: 'number' >> Add Watch** from the menu. The Watch window should appear in the panel at the bottom of the IDE:



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 says open!

6 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:



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

7 Continue running the program

When you want to resume, just press F5 (or Debug >> Continue), and the program will resume running as usual.

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



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;
}
```

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

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

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

```
for (int i = 0; i < 8; i = i + 2)
{
    MessageBox.Show("I'll pop up 4 times");
}
```

IDE Tip: Brackets

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

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

Use a code snippet to write simple for loops

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

Press tab to get the cursor to jump to the length. The number of times this loop runs is determined by whatever you set length to. You can change length to a number or a variable.

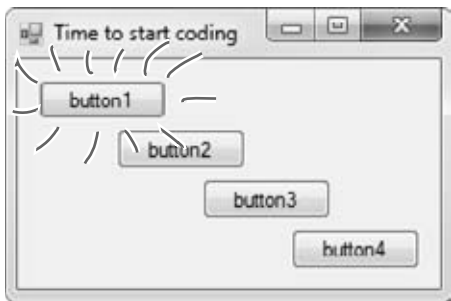
```
for (int i = 0; i < length; i++)
{
}
```

If you change the variable to something else, the snippet automatically changes the other two occurrences of it.

Time to start coding

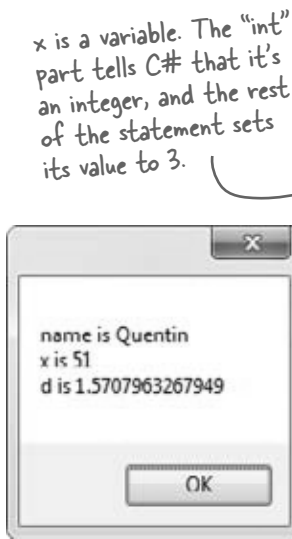
The real work of any program is in its statements. But statements don't exist in a vacuum. So let's set the stage for digging in and getting some code written. **Create a new Windows Forms Application project.**

Build this form



Add statements to show a message

Get started by double-clicking on the first button. Then add these statements to the `button1_Click()` method. Look closely at the code and the output it produces.



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

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

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 message box.

A few helpful tips

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

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

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

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

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

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

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

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

- ★ Most of the time, extra whitespace is fine:

```
int j      =      1234  ;
```

is the same as:

```
int j = 1234;
```

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.

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

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

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



Watch it!

Don't confuse the two equals sign operators!

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

Set up conditions and see if they're true

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

Use logical operators to check conditions

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

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

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

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.

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

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

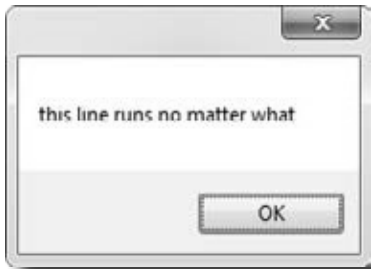
First we set up a variable called x and make it equal to 5. Then we check if it's equal to 10.



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

Add another conditional test

The third button makes this output. Now make a change to two lines of code so that it pops up both message boxes.



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

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

Add loops to your program

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

```
private void button4_Click(object sender, EventArgs e)
{
    int count = 0;
    while (count < 10)
    {
        count = count + 1;
    }
    for (int i = 0; i < 5; i++)
    {
        count = count - 1;
    }
    MessageBox.Show("The answer is " + count);
}
}
```

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

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

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

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

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



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

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

We filled in the first one for you.

More about conditional tests

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

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

These are the ones you'll use most often.



Wait up! There's a flaw in your logic. What happens to my loop if I write a conditional test that never becomes false?

Then your loop runs forever!

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

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

Sharpen your pencil

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

Loop #1

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

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

Loop #3

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

Loop #4

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

Loop #5

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

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

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

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

BRAIN POWER

Can you think of a reason that you'd want to write a loop that never stops running? (Hint: You'll use one in Chapter 13....)

Sharpen your pencil Solution

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

```
int result = 0; // this variable will hold the final result
int x = 6; // declare a variable x and set it to 6
while (x > 3) {
    // execute these statements as long as x is greater than 3
    result = result + x; // add x to the result variable
    x = x - 1; // subtract 1 from the value of x
}
for (int z = 1; z < 3; z = z + 1) {
    // start the loop by declaring a variable z and setting it to 1
    // keep looping as long as z is less than 3
    // after each loop, add 1 to z
    result = result + z; // add the value of z to result
}
// The next statement will pop up a message box that says
// The result is 18
MessageBox.Show("The result is " + result);
```

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

Sharpen your pencil Solution

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

Loop #1
This loop executes once

Loop #3
This loop executes 7 times

Loop #5
This loop executes 8 times

Loop #2
This loop runs forever

Loop #4
Another infinite loop

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

there are no Dumb Questions

Q: 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 an object on your form—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 are not recommended to use. 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 .NET Framework. It's where you find all of your important tools to add power to your programs, like `System.Data`, which lets you work with tables and databases, 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 form—it keeps the code you edit in one file (like `Form1.cs`), and the code it modifies automatically for you in another file (`Form1.Designer.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 form, 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 form, change properties. Then try to undo it. What happens? Well, for simple things you'll see that the IDE is smart enough to undo it itself. But for more complex things, like adding a new SQL database to your project, you'll be given a warning message. It still knows how to undo the action, but it may not be able to redo it.

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.

your code... now in magnet form



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 message box? 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 Result has no characters in it yet.

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

```
string Result = "";
```

```
MessageBox.Show(Result);
```

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

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

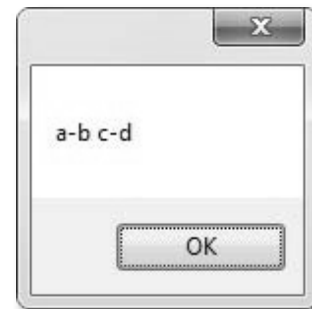
```
if (x > 2) {  
    Result = Result + "a";  
}
```

```
int x = 3;
```

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

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

Output:



→ Answers on page 82.

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

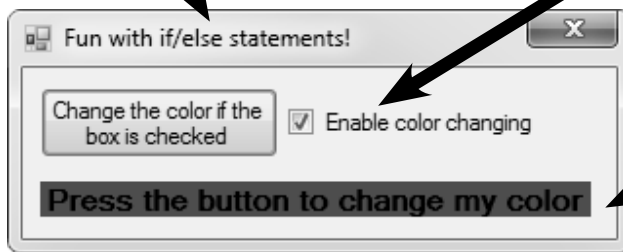
You'll be creating a lot of applications throughout this book, and you'll need to give each one a different name. We recommend naming this one "2 Fun with if-else statements" based on the chapter number and the text in the title bar of the form.



Exercise

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

Here's the form.



Add this checkbox.

Drag it out of the toolbox and onto your form. Use the **Text** property to change the text that's next to it. (You also use the **Text** property to change the button and label text.)

This is a label.

You can use the properties to change the font size and make it boldface. Use the **BackColor** property to set to red—choose "Red" from the selection of web colors.

Pop up this message if the user clicks the button but the box IS NOT checked.

If your checkbox is named `checkBox1` (you can change the **Name** property if you want), then here's the conditional test to see if it's checked:

```
checkBox1.Checked == true
```



If the user clicks the button and the box IS checked, change the background color of the label.

If the label background color is red, change it to blue when the button is clicked. If it's blue, change it back to red. Here's a statement that sets the background color of a label called `label1`:

```
label1.BackColor = Color.Red;
```

(Hint: The conditional test to check whether a label's background color is red looks a lot like that statement—but with one important difference!)



Exercise

Let's build something **flashy**! Start by creating a new Windows Forms Application in the IDE.

1 Here's the form to build

Hint: 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 loops. 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:

```
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 seeing what happens. The form doesn't redraw itself, because it's waiting until the loop is done before it deals with those events.

For now, you'll use `Application.DoEvents()` to make sure your form stays responsive while it's in a loop, but it's kind of a hack. You shouldn't use this code outside of a toy program like this. Later on in the book, you'll learn about a much better way to let your programs do more than one thing at a time!

3 Make it slower

Slow down the flashing by adding this line after the `Application.DoEvents()` line:

```
System.Threading.Thread.Sleep(3);
```

This statement inserts a 3 millisecond delay in the loop. It's a part of the .NET library, and it's in the `System.Threading` namespace.

Color me impressed!

.NET 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.

4 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.

5 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).

6 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.

When you're working with a form or control, `Visible` is true as long as the form or control is being displayed. If you set it to false, it makes the form or control disappear.

Hint: The `&&` operator means "AND". It's how you string a bunch of conditional tests together into one big test that's true only if the first test is true AND the second is true AND the third, etc. And it'll come in handy to solve this problem.

Can you figure out what's causing that delay? Can you fix it so the program ends immediately when you close the window?



Exercise Solution

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

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Fun_with_If_Else
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (checkBox1.Checked == true)
            {
                if (label1.BackColor == Color.Red)
                {
                    label1.BackColor = Color.Blue;
                }
                else
                {
                    label1.BackColor = Color.Red;
                }
            }
            else
            {
                MessageBox.Show("The box is not checked");
            }
        }
    }
}
```

Here's the code for the form. We named our solution "Fun with If Else", so the IDE made the namespace Fun_with_If_Else. If you gave your solution a different name, it'll have a different namespace.

The IDE added the method called button1_Click() to your form when you double-clicked on the button. The method gets run every time the button's clicked.

The outer if statement checks the checkbox to see if it's been checked. Check!

The inner if statement checks the label's color. If the label is currently red, it executes a statement to turn it blue.

This statement's run if the label's background color is not red to make it set back to red.

This MessageBox pops up if the checkbox isn't checked.

You can download the code for all of the exercise solutions in this book from www.headfirstlabs.com/books/hfcsharp/



Exercise Solution

Sometimes we won't show you the entire code in the solution, just the bits that changed. All of the logic in the FlashyThing project is in this `button1_Click()` method that the IDE added when you double-clicked the button in the form designer.

Let's build something **flashy!**

When the IDE added this method, it added an extra return before the curly bracket. Sometimes we'll put the bracket on the same line like this to save space—but C# doesn't care about extra space, so this is perfectly valid.

Consistency is generally really important to make it easy for people to read code. But we're purposefully showing you different ways, because you'll need to get used to reading code from different people using different styles.

```
private void button1_Click(object sender, EventArgs e) {
```

```
    while (Visible) {
```

The outer loop keeps running as long as the form is visible. As soon as it's closed, `Visible` is false, and the while will stop looping.

```
        for (int c = 0; c < 254 && Visible; c++) {
```

```
            this.BackColor = Color.FromArgb(c, 255 - c, c);
```

```
            Application.DoEvents();
```

```
            System.Threading.Thread.Sleep(3);
```

The first for loop makes the colors cycle one way, and the second for loop reverses them so they look smooth.

```
        }
```

We used `==`

Visible instead of `!=` Visible

`== true`. It's just like saying "if it's visible" instead of "if it's true that it's visible"—they mean the same thing.

```
        for (int c = 254; c >= 0 && Visible; c--) {
```

```
            this.BackColor = Color.FromArgb(c, 255 - c, c);
```

```
            Application.DoEvents();
```

```
            System.Threading.Thread.Sleep(3);
```

```
        }
```

We fixed the extra delay by using the `==` operator to make each of the for loops also check `Visible`. That way the loop ends as soon as `Visible` turns false.

Can you figure out what's causing that delay? Can you fix it so the program ends immediately when you close the window?

The delay happens because the for loops need to finish before the while loop can check if `Visible` is still true. You can fix it by adding `&& Visible` to the conditional test in each for loop.

Was your code a little different than ours? There's more than one way to solve any programming problem—like you could have used while loops instead of for loops. If your program works, then you got the exercise right!

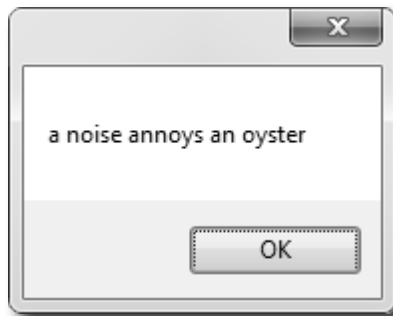
this puzzle's tougher than it looks

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



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!

```
x > 0
x < 1
x > 1
x > 3
x < 4
Poem = Poem + " ";
Poem = Poem + "a ";
Poem = Poem + "n ";
Poem = Poem + "an ";
x = x + 1;
x = x + 2;
x = x - 2;
x = x - 1;
MessageBox.Show(Poem);
Poem = Poem + "noys ";
Poem = Poem + "oise ";
Poem = Poem + " oyster ";
Poem = Poem + "annoys ";
Poem = Poem + "noise";
```

```
int x = 0;
String Poem = "";

while ( _____ ) {

    _____

    if ( x < 1 ) {
        _____
    }

    _____

    if ( _____ ) {
        _____
    }

    _____

    if ( x == 1 ) {
        _____
    }

    _____

    if ( _____ ) {
        _____
    }

    _____

}

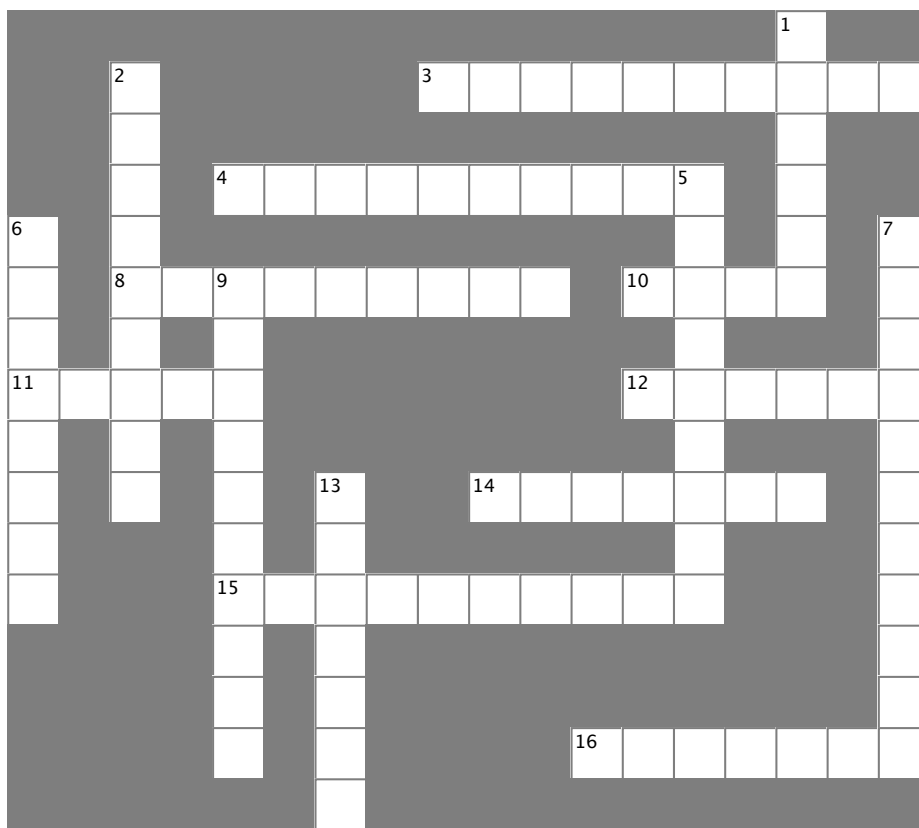
_____

_____
```



Csharpcross

How does a crossword help you learn C#? Well, all the words are C#-related and from this chapter. The clues also provide mental twists and turns that will help you burn alternative routes to C# right into your brain.



Across

3. You give information to a method using these _____
4. `button1.Text` and `checkBox3.Name` are examples of _____
8. Every statement ends with one of these _____
10. The name of every C# program's entry point _____
11. Contains methods _____
12. Your code statements live in one of these _____
14. A kind of variable that's either true or false _____
15. A special method that tells your program where to start _____
16. This kind of class spans multiple files _____

Down

1. The output of a method is its _____ value
2. `System.Windows.Forms` is an example of one of these _____
5. A tiny piece of a program that does something _____
6. A block of code is surrounded by _____
7. The kind of test that tells a loop when to end _____
9. You can call _____.`Show()` to pop up a simple Windows dialog box
13. The kind of variable that contains a whole number _____



Code Magnets Solution

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 message box? 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!

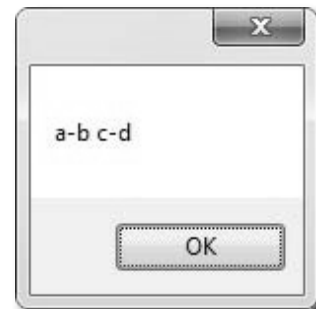
```
string Result = "";  
int x = 3;  
while (x > 0) {  
    if (x > 2) {  
        Result = Result + "a";  
    }  
    x = x - 1;  
    Result = Result + "-";  
    if (x == 2) {  
        Result = Result + "b c";  
    }  
    if (x == 1) {  
        Result = Result + "d";  
        x = x - 1;  
    }  
}  
MessageBox.Show(Result);
```

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

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.

Output:



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 a class that will compile and run.

```
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";

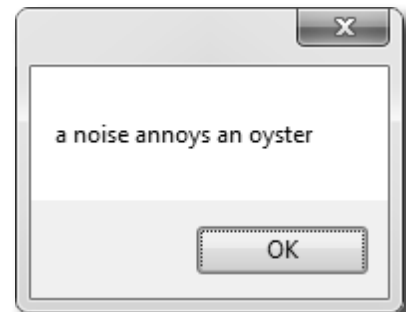
        x = x + 2;
    }
    if ( x == 1 ) {

        Poem = Poem + "noys ";
    }
    if ( x < 1 ) {

        Poem = Poem + "oise ";
    }

    x = x + 1;
}
MessageBox.Show(Poem);
```

Output:



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 it is! Here's a hint: There's another solution that keeps the word fragments in order.

3 objects: get oriented!



Making code make sense



...and that's why my Husband class doesn't have a `HelpOutAroundTheHouse()` method or a `PullHisOwnWeight()` method.



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.



I'll take the 31st Street bridge, head up Liberty Avenue, and go through Bloomfield.

Mike sets his destination, then comes up with a route.

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

Mike gets new information about a street he needs to avoid.



This is Frank Loudly with your eye-in-the-sky shadow traffic report. It looks like a three-car pileup on Liberty has traffic backed up all the way to 32nd Street.

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

Now he can come up with a new route to the interview.

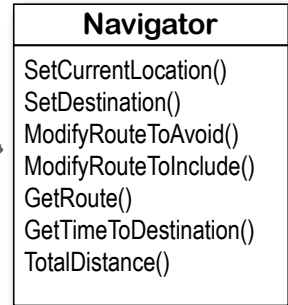
No problem. If I take Route 28 instead, I'll still be on time!



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.

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



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

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

Here's the output from the GetRoute() method—it's a string that contains the directions Mike should follow.

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

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

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

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

```
string route;
route = GetRoute();
```

"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.

Mike chose method names that would make sense to someone who was thinking about how to navigate a route through a city.

```
class Navigator {  
    public void SetCurrentLocation(string locationName) { ... }  
    public void SetDestination(string destinationName) { ... };  
    public void ModifyRouteToAvoid(string streetName) { ... };  
    public string GetRoute() { ... };  
}  
  
string route =  
    GetRoute();
```

This is the return type of the method. It means that the statement calling the `GetRoute()` method can use it to set a string variable that will contain the directions. When it's void, that means the method doesn't return anything.

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 just ends with a semicolon, 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.

```
public int MultiplyTwoNumbers(int firstNumber, int secondNumber) {  
    int result = firstNumber * secondNumber;  
    return result;  
}
```

Here's an example of a method that has a return type—it returns an `int`. The method uses the two **parameters** to calculate the result and uses the **return** statement to pass the value back to the statement that called it.

Here's a statement that calls a method to multiply two numbers. It returns an `int`:

```
int myResult = MultiplyTwoNumbers(3, 5);
```

Methods can take values like 3 and 5. But you can also use variables to pass values to a method.



BULLET POINTS

- Classes have methods that contain statements that perform actions. You can design a class that is easy to use by choosing methods that make sense.
- Some methods have a **return type**. You set a method's return type in its declaration. A method with a declaration that starts "public int" returns an int value. Here's an example of a statement that returns an int value: `return 37;`
- When a method has a return type, it **must** have a `return` statement that returns a value that matches a return type. So if you've got a method that's declared "public string" then you need a `return` statement that returns a string.
- As soon as a `return` statement in a method executes, your program jumps back to the statement that called the method.
- Not all methods have a return type. A method with a declaration that starts "public void" doesn't return anything at all. You can still use a `return` statement to exit a void method: `if (finishedEarly) { return; }`

Use what you've learned to build a program that uses a class

Let's hook up a form to a class, and make its button call a method inside that class.

Do this!

- 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:

```
class Talker {
    public static int BlahBlahBlah(string thingToSay, int numberOfTimes)
    {
        string finalString = "";
        for (int count = 1; 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.

This line of code adds the contents of `thingToSay` and a line break ("`\n`") onto the end of it to the `finalString` variable.

This is called a **property**. Every string has a property called `Length`. When it calculates the length of a string, a line break ("`\n`") counts as one character.

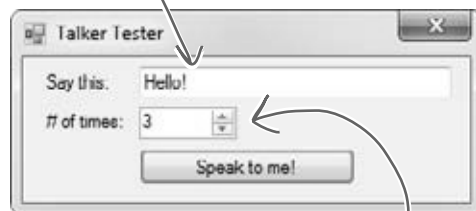
→ **Flip the page to keep going!**

So what did you just build?

The new class has one method called `BlahBlahBlah()` that takes two parameters. The first parameter is a string that tells it something to say, and the second is the number of times to say it. When it's called, it pops up a message box with the message repeated a number of times. Its return value is the length of the string. The method needs a string for its `thingToSay` parameter and a number for its `numberOfTimes` parameter. It'll get those parameters from a form that lets the user enter text using a **TextBox** control and a number using **NumericUpDown** control.

Now add a form that uses your new class!

Set the default text of the TextBox to "Hello!" using its Text property.



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`:

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

This is a **NumericUpDown** control. Set its **Minimum** property to 1, its **Maximum** property to 10, and its **Value** property to 3.

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.

The `BlahBlahBlah()` method pops up this message box based on what's in its parameters.



When the method returns a value, the form pops it up in this message box.



You can add a class to your project and share its methods with the other classes in the project.

Mike gets an idea

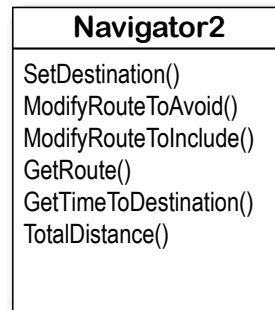
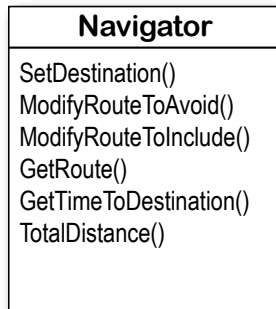
The interview went great! But the traffic jam this morning got Mike thinking about how he could improve his navigator.



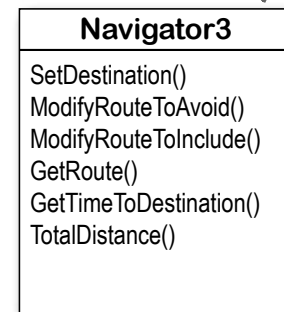
It'd be great if I could compare a few routes and figure out which is fastest....

He could create three different Navigator classes...

Mike *could* copy the Navigator class code and paste it into two more classes. Then his program could store three routes at once.



This box is a **class diagram**. It lists all of the methods in a class, and it's an easy way to see everything that it does at a glance.



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.

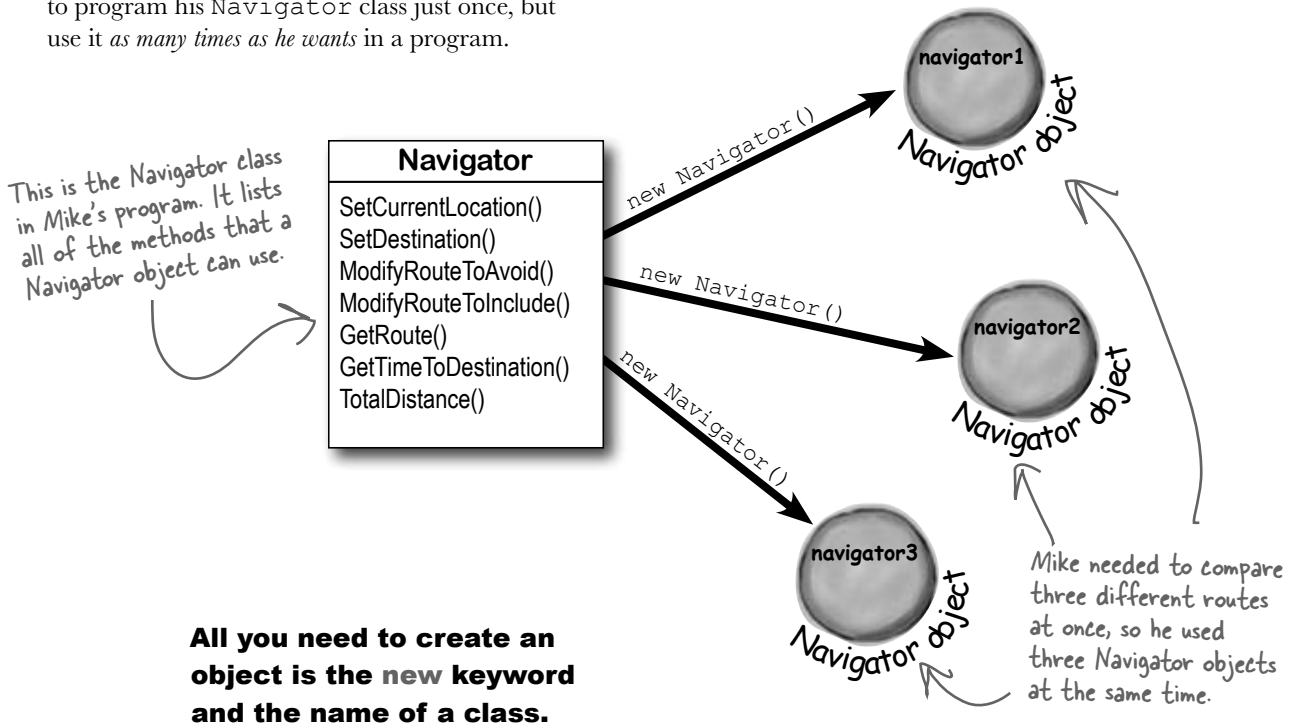


Right! Maintaining three copies of the same code is really messy. A lot of problems you need to solve need a way to represent one *thing* a bunch of different times. In this case, it's a bunch of routes. But it could be a bunch of turbines, or dogs, or music files, or anything. All of those programs 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.

for instance...

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.

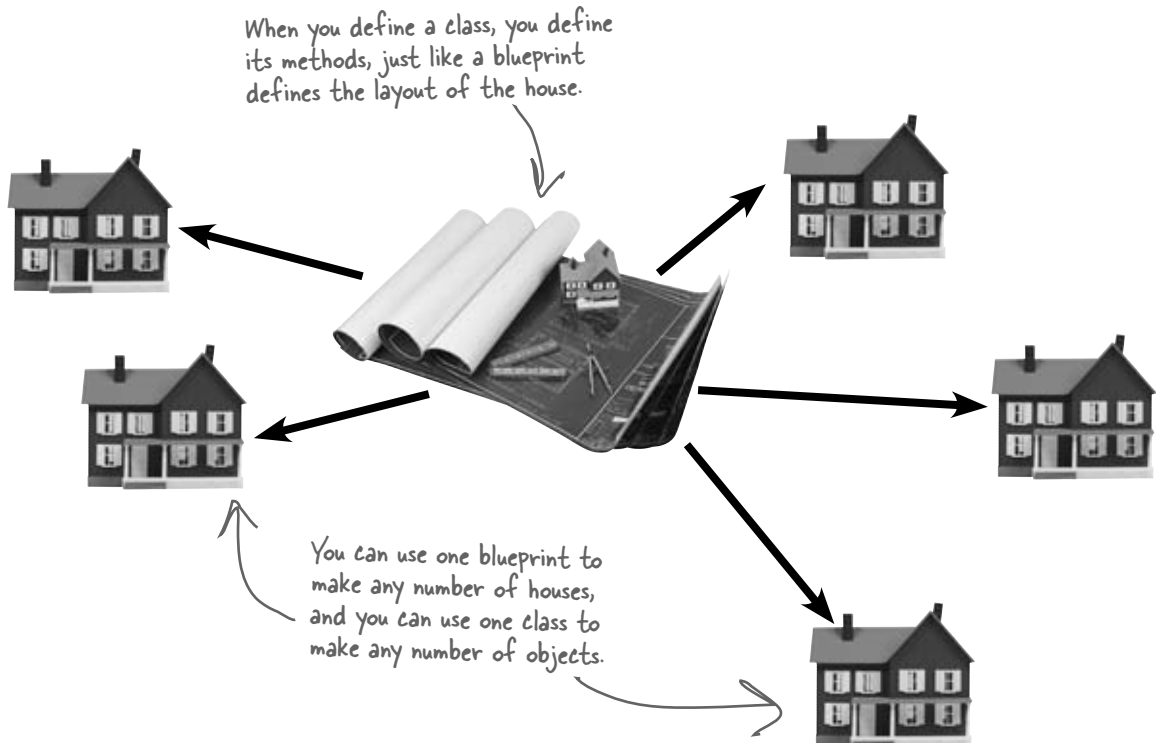


```
Navigator navigator1 = new Navigator();  
navigator1.SetDestination("Fifth Ave & Penn Ave");  
string route;  
route = navigator1.GetRoute();
```

Now you can use the object! When you create an object from a class, that object has all of the methods from that class.

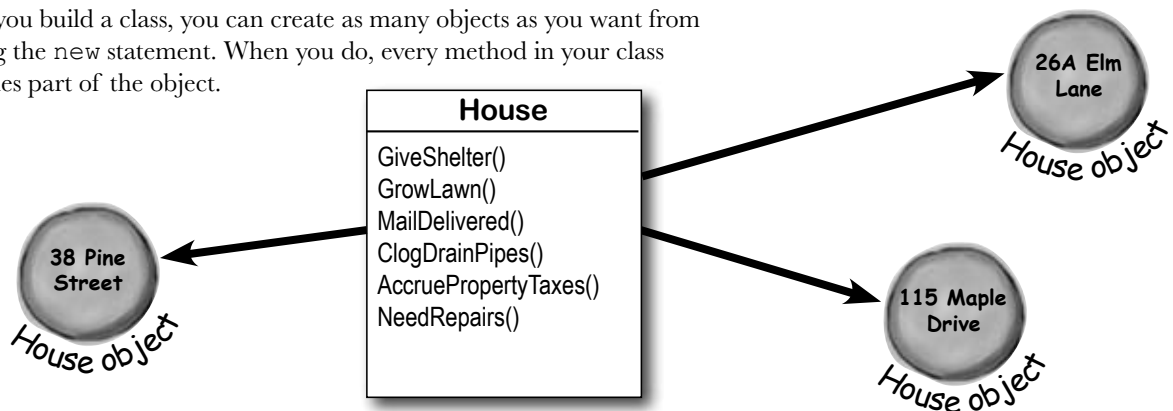
You use a class to build an object

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



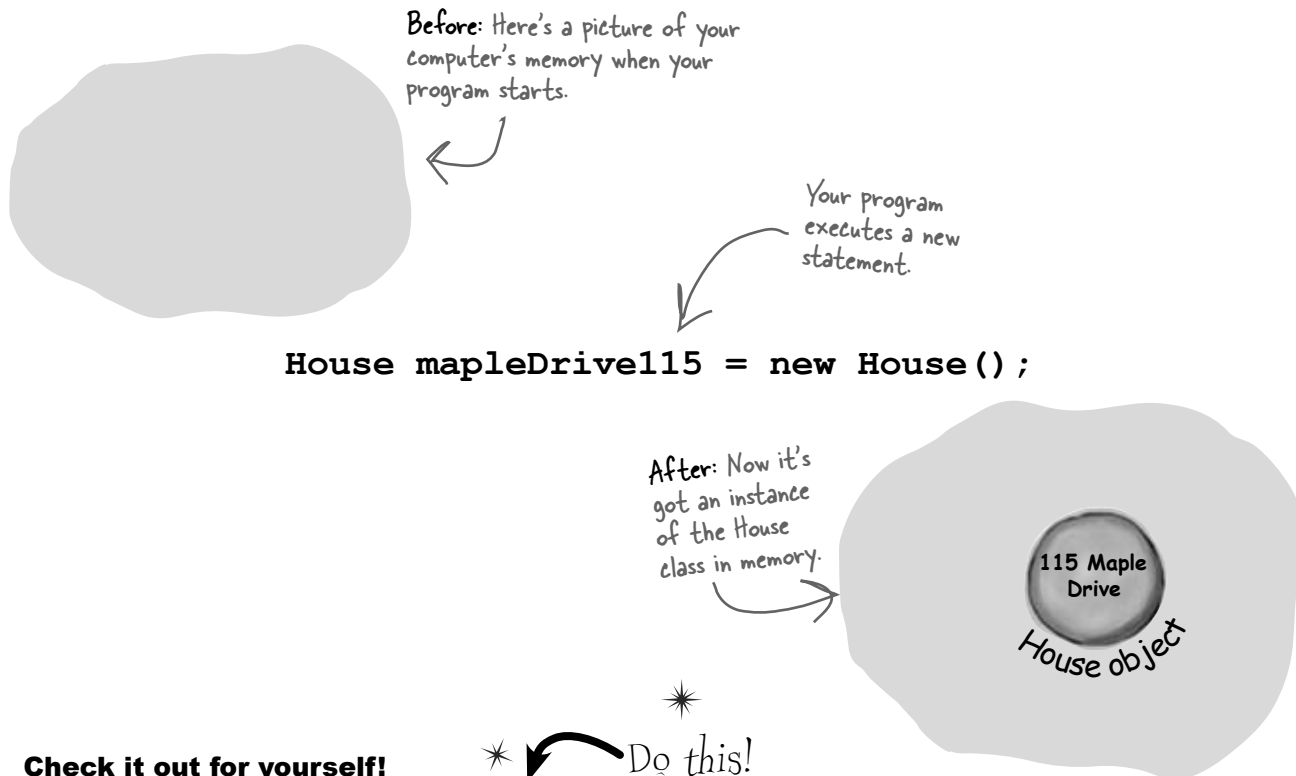
An object gets its methods from its class

Once you build a class, you can create as many objects as you want from it using the new statement. When you do, every method in your class becomes part of the object.



When you create a new object from a class, it's called an instance of that class

Guess what...you already know this stuff! Everything in the toolbox is a class: there's a Button class, a TextBox class, a Label class, etc. When you drag a button out of the toolbox, the IDE automatically creates an instance of the Button class and calls it `button1`. When you drag another button out of the toolbox, it creates another instance called `button2`. Each instance of Button has its own properties and methods. But every button acts exactly the same way, because they're all instances of the same class.



Check it out for yourself!

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.

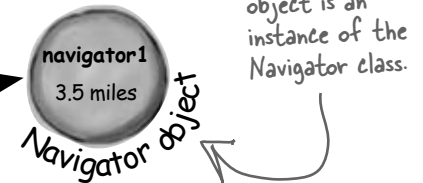
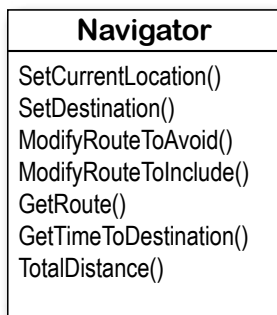
in-stance, noun.
an example or one occurrence of something. *The IDE search-and-replace feature finds every **instance** of a word and changes it to another.*

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.

GUI stands for Graphical User Interface, which is what you're building when you make a form in the form designer.

- 1 Mike set up a GUI with a text box—`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.



The `navigator1` object is an instance of the `Navigator` class.

```
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.

The `SetDestination()`, `ModifyRouteToAvoid()`, and `ModifyRouteToInclude()` methods all take a string as a parameter.



- 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!

Any time you create a new object from a class, it's called creating an instance of that class.



Wait a minute! You didn't give me nearly enough information to build the navigator program.

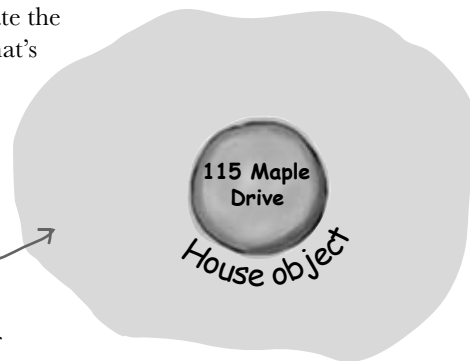
That's right, we didn't. A geographic navigation program is a really complicated thing to build. But complicated programs follow the same patterns as simple ones. Mike's navigation program is an example of how someone would use objects in real life.

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 small code excerpts to demonstrate the idea. This is your opportunity to take a step back and try to understand what's going on without having to worry about getting a program to work.

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

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



After we've introduced a concept, we'll give you a chance to get it into your brain. Sometimes we'll follow up the theory with a writing exercise—like the *Sharpen your pencil* exercise on the next page. Other times we'll jump straight into code. This combination of theory and practice is an effective way to get these concepts off of the page and stuck in your brain.

A little advice for the code exercises

If you keep a few simple things in mind, it'll make the code exercises go smoothly:

- ★ It's easy to get caught up in syntax problems, like missing parentheses or quotes. One missing bracket can cause many build errors.
- ★ It's ***much better*** to look at the solution than 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 2010! 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/hfcsharp>

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.



Follow the same steps that Mike followed on the facing page to write the code to create `Navigator` objects and call their methods.

```
string destination = textBox1.Text;
string route2StreetToAvoid = textBox2.Text;
string route3StreetToInclude = textBox3.Text;
```

We gave you a head start. Here's the code Mike wrote to get the destination and street names from the textboxes.

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

And here's the code to create the navigator object, set its destination, and get the distance.

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

Navigator navigator2 =

navigator2......

navigator2......

int distance2 =

2. Create the **navigator3** object, set its destination, call its **ModifyRouteToInclude()** method, and use its **TotalDistance()** method to set an integer variable called **distance3**.

.....

.....

.....

.....

The `Math.Min()` method built into the .NET Framework compares two numbers and returns the smallest one. Mike used it to find the shortest distance to the destination.

```
int shortestDistance = Math.Min(distance1, Math.Min(distance2, distance3));
```



Sharpen your pencil Solution

Follow the same steps that Mike followed on the facing page to write the code to create `Navigator` objects and call their methods.

```
string destination = textBox1.Text;
string route2StreetToAvoid = textBox2.Text;
string route3StreetToInclude = textBox3.Text;

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

We gave you a head start. Here's the code Mike wrote to get the destination and street names from the textboxes.

And here's the code to create the navigator object, set its destination, and get the distance.

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

```
Navigator navigator2 = new Navigator()
navigator2.SetDestination(destination);
navigator2.ModifyRouteToAvoid(route2StreetToAvoid);
int distance2 = navigator2.TotalDistance();
```

2. Create the **navigator3** object, set its destination, call its **ModifyRouteToInclude()** method, and use its **TotalDistance()** method to set an integer variable called **distance3**.

```
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));
```

I've written a few classes now, but I haven't used "new" to create an instance yet! So does that mean I can call methods without creating objects?



Yes! That's why you used the `static` keyword in your methods.

Take another look at the declaration for the `Talker` class you built a few pages ago:

```
class Talker
{
    public static int BlahBlahBlah(string thingToSay, int numberOfTimes)
    {
        string finalString = "";
    }
}
```

When you called the method you didn't create a new instance of `Talker`. You just did this:

```
Talker.BlahBlahBlah("Hello hello hello", 5);
```

That's how you call `static` methods, and you've been doing that all along. If you take away the `static` keyword from the `BlahBlahBlah()` method declaration, then you'll have to create an instance of `Talker` in order to call the method. Other than that distinction, `static` methods are just like object methods. You can pass parameters, they can return values, and they live in classes.

There's one more thing you can do with the `static` keyword. You can mark your **whole class** as `static`, and then all of its methods **must** be `static` too. If you try to add a non-`static` method to a `static` class, it won't compile.

there are no Dumb Questions

Q: When I think of something that's "static," I think of something that doesn't change. Does that mean non-`static` methods can change, but `static` methods don't? Do they behave differently?

A: No, both `static` and non-`static` methods act exactly the same. The only difference is that `static` methods don't require an instance, while non-`static` 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

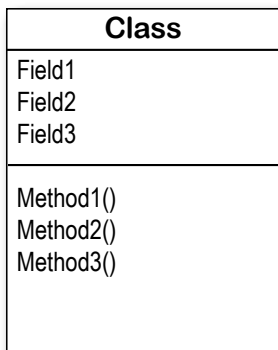
You change the text on a button by setting its Text property in the IDE. When you do, the IDE adds code like this to the designer:

```
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.

Technically, it's setting a property. A property is very similar to a field—but we'll get into all that a little later on.

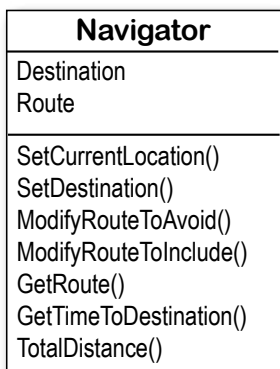
This is where a class diagram shows the fields. Every instance of the class uses them to keep track of its state.



Add this line to separate the fields from the methods.

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.



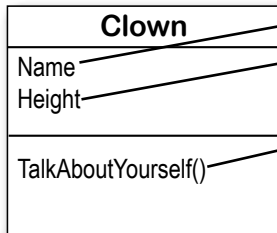
Every instance of `Navigator` knows its destination and its route.

What a `Navigator` object does is let you set a destination, modify its route, and get information about that route.

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.



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

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

Remember, when you see "void" in front of a method, it means that it doesn't return any value.

When you want to create instances of your class, don't use the static keyword in either the class declaration or the method declaration.

Remember, the *= operator tells C# to take whatever's on the left of the operator and multiply it by whatever's on the right.

Sharpen your pencil

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;
```

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

"My name is _____ and I'm _____ inches tall."

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

```
anotherClown.TalkAboutYourself();
```

"My name is _____ and I'm _____ inches tall."

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

```
clown3.TalkAboutYourself();
```

"My name is _____ and I'm _____ inches tall."

```
anotherClown.Height *= 2;
```

```
anotherClown.TalkAboutYourself();
```

"My name is _____ and I'm _____ inches tall."

Thanks for the memory

When your program creates an object, it lives in a part of the computer's memory called the **heap**. When your code creates an object with a new statement, C# immediately reserves space in the heap so it can store the data for that object.

Here's a picture of the heap before the project starts. Notice that it's empty.



Let's take a closer look at what happened here

Sharpen your pencil Solution

```
Clown oneClown = new Clown();  
oneClown.Name = "Boffo";  
oneClown.Height = 14;  
oneClown.TalkAboutYourself();  
  
Clown anotherClown = new Clown();  
anotherClown.Name = "Biff";  
anotherClown.Height = 16;  
anotherClown.TalkAboutYourself();  
  
Clown clown3 = new Clown();  
clown3.Name = anotherClown.Name;  
clown3.Height = oneClown.Height - 3;  
clown3.TalkAboutYourself();  
  
anotherClown.Height *= 2;  
anotherClown.TalkAboutYourself();
```

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

Each of these new statements creates an instance of the Clown class by reserving a chunk of memory on the heap for that object and filling it up with the object's data.

"My name is Boffo and I'm 14 inches tall."

"My name is Biff and I'm 16 inches tall."

"My name is Biff and I'm 11 inches tall."

"My name is Biff and I'm 32 inches tall."

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();`

The first object is created, and its fields are set.

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

These statements create the second object and fill it with data.

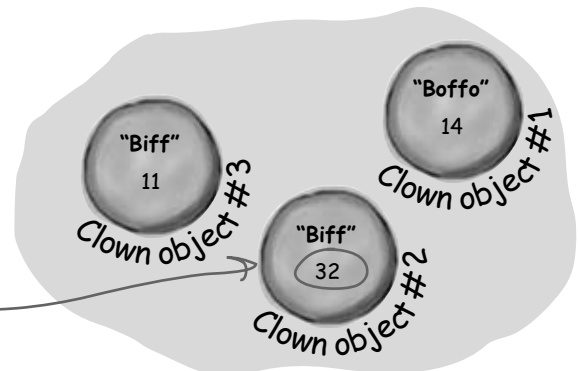
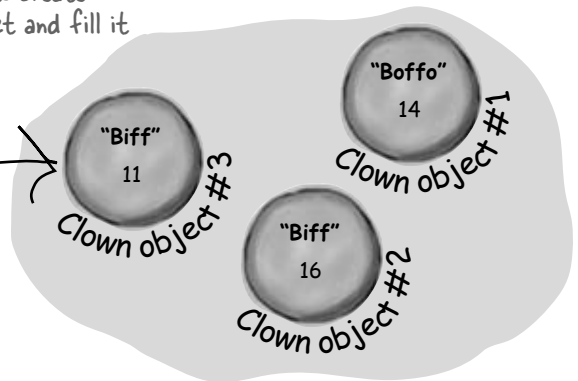
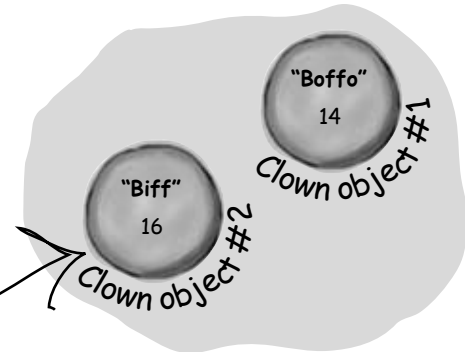
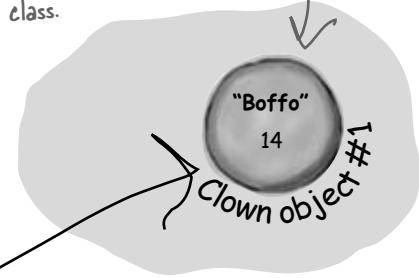
3 `Clown clown3 = new Clown();`
`clown3.Name = anotherClown.Name;`
`clown3.Height = oneClown.Height - 3;`
`clown3.TalkAboutYourself();`

Then the third Clown object is created and populated.

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

There's no new command, which means these statements don't create a new object. They're just modifying one that's already in memory.

This object is an instance of the Clown class.



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.

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

```

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

```

“tb”, “ics”, and “m” are terrible names! We have no idea what they do. And what's that T class for?

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

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

Take a second and look at that code. Can you figure out what it does?

- 2** Those statements don't give you any hints about why the code's doing what it's doing. In this case, the programmer was happy with the results because she was able to get it all into one method. But making your code as compact as possible isn't really useful! Let's break it up into methods to make it easier to read, and make sure the classes are given names that make sense. But we'll start by figuring out what the code is supposed to do.

How do you figure out what your code is supposed to do? Well, all code is written for a reason. So it's up to you to figure out that reason! In this case, we can look up the page in the specification manual that the programmer followed.

General Electronics Type 5 Candy Bar Maker Specification Manual

The nougat temperature must be checked every 3 minutes by an automated system. If the temperature **exceeds 160°C**, the candy is too hot, and the system must **perform the candy isolation cooling system (CICS) vent procedure**.

- Close the trip throttle valve on turbine #2
- Fill the isolation cooling system with a solid stream of water
- Vent the water
- Verify that there is no evidence of air in the system

- 3 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.

The `IsNougatTooHot()` method's return type

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

By naming the class "Maker" and the method "CheckNougatTemperature", the code is a lot easier to understand.

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

- 4 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):

A void return type means the method doesn't return any value at all.

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

- 5 Now the code's a lot more intuitive! Even if you don't know that the CICS vent procedure needs to be run if the nougat is too hot, **it's a lot more obvious what this code is doing:**

```
if (IsNougatTooHot () == true) {
    DoCICSVentProcedure ();
}
```

You can make your code easier to read and write by thinking about the problem your code was built to solve. If you choose names for your methods that make sense to someone who understands that problem, then your code will be a lot easier to decipher...and develop!

Give your classes a natural structure

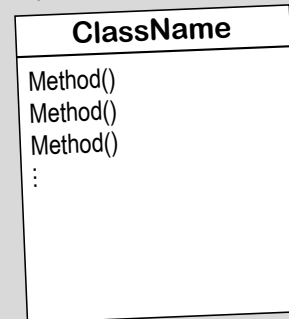
Take a second and remind yourself why you want to make your methods intuitive:

because every program solves a problem or has a purpose. It might not be a business problem—sometimes a program’s purpose (like `FlashyThing`) is just to be cool or fun! But no matter what your program does, the more you can make your code resemble the problem you’re trying to solve, the easier your program will be to write (and read, and repair, and maintain...).

Use class diagrams to plan out your classes

A class diagram is a simple way to draw your classes out on paper. It’s a really valuable tool for designing your code **BEFORE** you start writing it.

Write the name of the class at the top of the diagram. Then write each method in the box at the bottom. Now you can see all of the parts of the class at a glance!

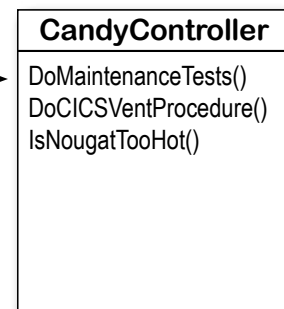


Let’s build a class diagram

Take another look at the `if` statement in #5 on the previous page. You already know that statements always live inside methods, which always live inside classes, right? In this case, that `if` statement was in a method called `DoMaintenanceTests()`, which is part of the `CandyController` class. Now take a look at the code and the class diagram. See how they relate to each other?

```
class CandyController {
    public void DoMaintenanceTests() {
        ...
        if (IsNougatTooHot() == true) {
            DoCICSVentProcedure();
        }
        ...
    }

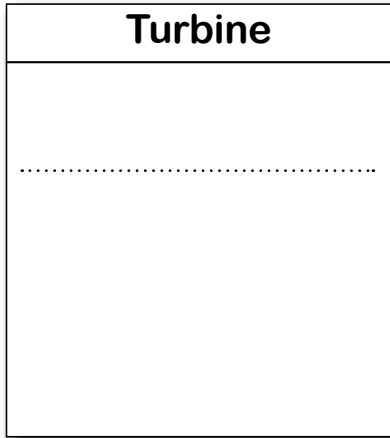
    public void DoCICSVentProcedure() ...
    public boolean IsNougatTooHot() ...
}
```



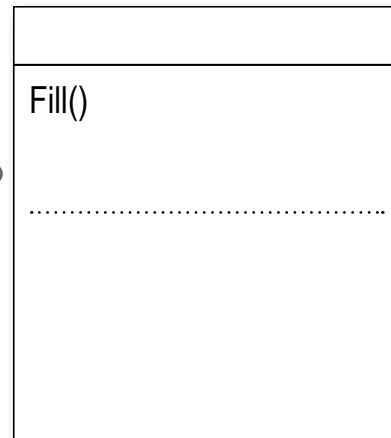

Sharpen your pencil



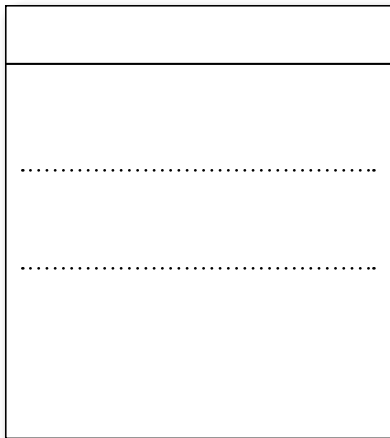

The code for the candy control system we built on the previous page called three other classes. Flip back and look through the code, and fill in their class diagrams.




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.

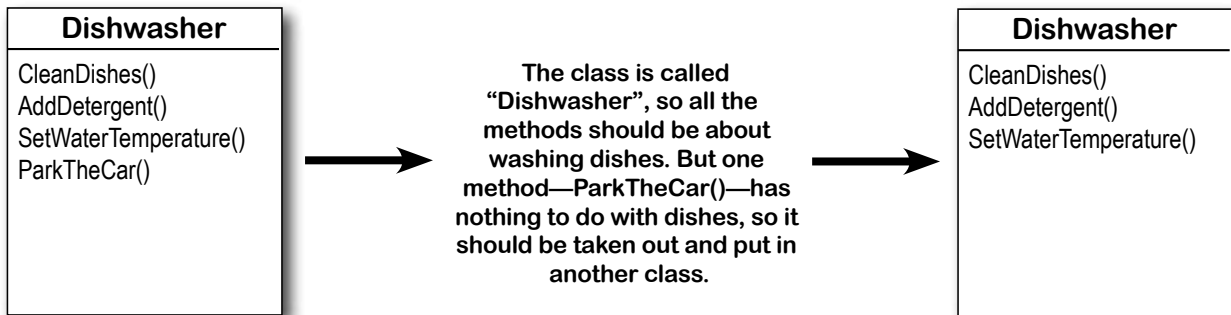


There was one other class in the code on the previous page. Fill in its name and method.



Class diagrams help you organize your classes so they make sense

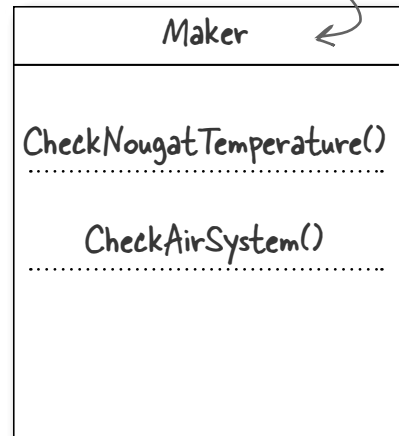
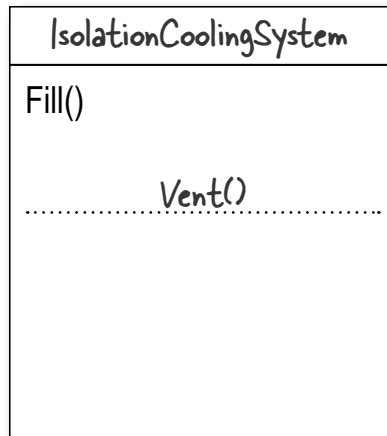
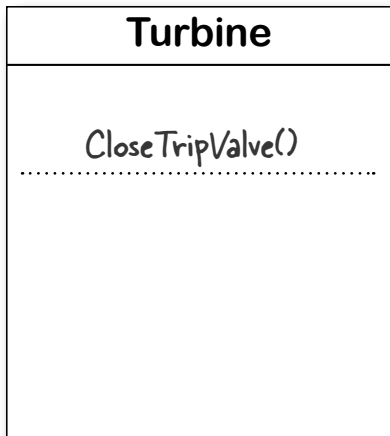
Writing out class diagrams makes it a lot easier to spot potential problems in your classes **before** you write code. Thinking about your classes from a high level before you get into the details can help you come up with a class structure that will make sure your code addresses the problems it solves. 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.



Sharpen your pencil Solution

The code for the candy control system we built on the previous page called three other classes. Flip back and look through the code, and fill in their class diagrams.

You could figure out that *Maker* is a class because it appears in front of a dot in *Maker.CheckAirSystem()*.





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.

```

Class23
CandyBarWeight()
PrintWrapper()
GenerateReport()
Go()

```

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

.....

.....

.....

.....

```

DeliveryGuy
AddAPizza()
PizzaDelivered()
TotalCash()
ReturnTime()

```

```

DeliveryGirl
AddAPizza()
PizzaDelivered()
TotalCash()
ReturnTime()

```

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

.....

.....

.....

.....

```

CashRegister
MakeSale()
NoSale()
PumpGas()
Refund()
TotalCashInRegister()
GetTransactionList()
AddCash()
RemoveCash()

```

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

.....

.....

.....

.....



Here's how we corrected the classes. We show just one possible way to fix the problems—but there are plenty of other ways you could design these classes depending on how they'll be used.

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

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

CandyMaker
CandyBarWeight() PrintWrapper() GenerateReport() MakeTheCandy()

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

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

DeliveryPerson
Gender
AddAPizza() PizzaDelivered() TotalCash() ReturnTime()

We 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.

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

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

CashRegister
MakeSale() NoSale() Refund() TotalCashInRegister() GetTransactionList() AddCash() RemoveCash()


```

public partial class Form1 : Form
{
    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...";
        }
    }
}

```



Pool Puzzle

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make classes that will compile and run and produce the output listed.

Output



Bonus Question!

If the last line of output was **24** instead of **10**, how would you complete the puzzle? You can do it by changing just one statement.

Note: Each snippet from the pool can be used more than once!

	x < 4			
x	x < 5	Echo		
y	x > 0	Tester		
e2	x > 1	Echo()	e2 = e1;	
count		Count()	Echo e2;	
e1 = e1 + 1;		Hello()	Echo e2 = e1;	x == 3
e1 = count + 1;			Echo e2 = new Echo();	x == 4
e1.count = count + 1;				
e1.count = e1.count + 1;				

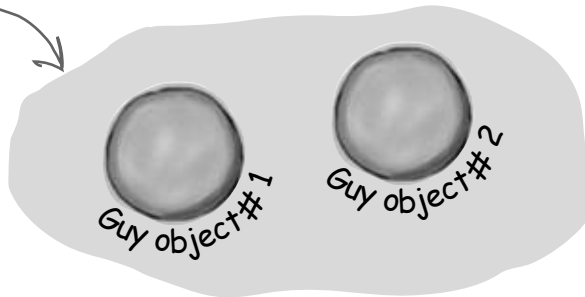
→ Answers on page 122.

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 `joe` (to keep track of the first object), and the other called `bob` (to keep track of the second object).

The new statements that create the two instances live in the code that gets run as soon as the form is created. Here's what the heap looks like after the form is loaded.

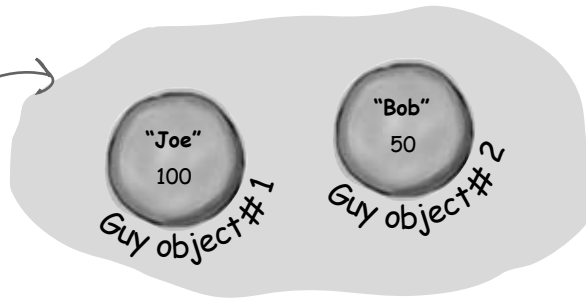


Guy
Name Cash
GiveCash() ReceiveCash()

We chose names for the methods that make sense. You call a Guy object's `GiveCash()` method to tell him to give up some of his cash, and his `ReceiveCash()` method when you want him to take some cash back. We could have called them `GiveCashToSomeone()` and `ReceiveCashFromSomeone()`, but that would have been very long!

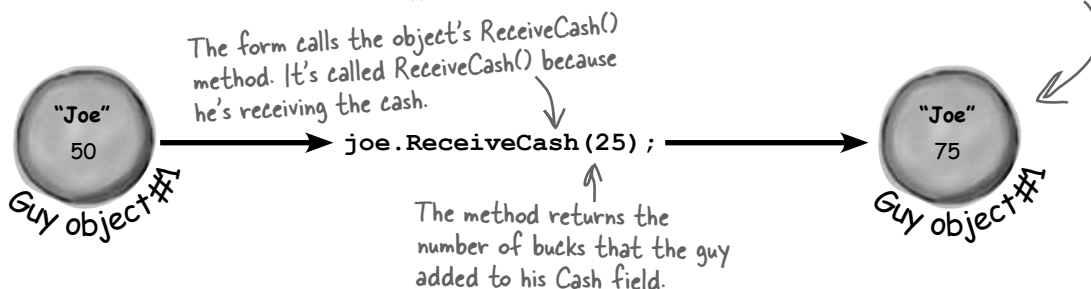
- 2 We'll set each Guy object's cash and name fields**
The two objects represent different guys, each with his own name and a different amount of cash in his pocket.

Each guy has a `Name` field that keeps track of his name, and a `Cash` field that has the number of bucks in his pocket.



When you take an instance of `Guy` and call its `ReceiveCash()` method, you pass the amount of cash the guy will take as a parameter. So calling `joe.ReceiveCash(25)` tells Joe to receive 25 bucks and add them to his wallet.

- 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.



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:



```
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.

The Guy makes sure that you're asking him for a positive amount of cash, otherwise he'd add to his cash instead of taking away from it.

If the guy doesn't have enough cash, he'll tell you so with a message box, and then he'll make GiveCash() return 0.

The ReceiveCash() method works just like the GiveCash() method. It's passed an amount as a parameter, checks to make sure that amount is greater than zero, and then adds it to his cash.

If the amount was positive, then the ReceiveCash() method returns the amount added. If it was zero or negative, the guy shows a message box and then returns 0.

Be careful with your curly brackets. It's easy to have the wrong number—make sure that every opening bracket has a matching closing bracket. When they're all balanced, the IDE will automatically indent them for you when you type the last closing bracket.

Build a form to interact with the guys

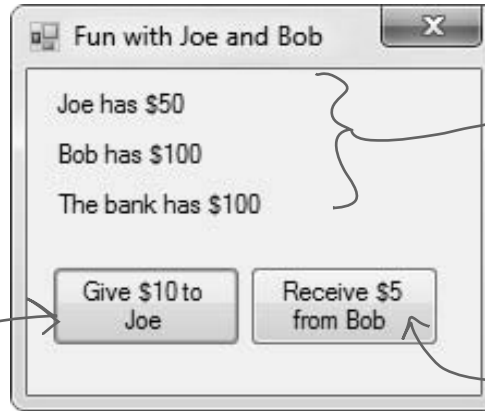
The Guy class is great, but it’s just a start. Now put together a form that uses two instances of the Guy class. It’s got labels that show you their names and how much cash they have, and buttons to give and take cash from them.



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”.

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.

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

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

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 Guy objects.

3 Add a method to the form to update the labels

The labels on the right-hand side of the form show how much cash each guy has and how much is in the bank field. So add the `UpdateForm()` method to keep them up to date—**make sure the return type is `void`** to tell C# that the method doesn't return a value. Type this method into the form right underneath where you added the bank field:

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

Notice how the labels are updated using the Guy objects' Name and Cash fields.

This new method is simple. It just updates the three labels by setting their `Text` properties. You'll have each button call it to keep the labels up to date.

4 Double-click on each button and add the code to interact with the objects

Make sure the left-hand button is called `button1`, and the right-hand 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:

```
private void button1_Click(object sender, EventArgs e) {
    if (bank >= 10) {
        bank -= joe.ReceiveCash(10);
        UpdateForm();
    } else {
        MessageBox.Show("The bank is out of money.");
    }
}
```

When the user clicks the "Give \$10 to Joe" button, the form calls the Joe object's `ReceiveCash()` method—but only if the bank has enough money.

The bank needs at least \$10 to give to Joe. If there's not enough, it'll pop up this message box.

```
private void button2_Click(object sender, EventArgs e) {
    bank += bob.GiveCash(5);
    UpdateForm();
}
```

The "Receive \$5 from Bob" button doesn't need to check how much is in the bank, because it'll just add whatever Bob gives back.

If Bob's out of money, `GiveCash()` will return zero.

5 Start Joe out with \$50 and start Bob out with \$100

It's up to you to **figure out how to get Joe and Bob to start out with their Cash and Name fields set properly**. Put it right underneath `InitializeComponent()` in the form. That's part of that designer-generated method that gets run once, when the form is first initialized. Once you've done that, click both buttons a number of times—make sure that one button takes \$10 from the bank and adds it to Joe, and the other takes \$5 from Bob and adds it to the bank.

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

Add the lines of code here to create the two objects and set their `Name` and `Cash` fields.



Exercise



Exercise Solution

It's up to you to **figure out how to get Joe and Bob to start out with their Cash and Name fields set properly**. Put it right underneath `InitializeComponent()` in the form.

Here's where we set up the first instance of `Guy`. The first line creates the object, and the next two set its fields.

```
public Form1() {
    InitializeComponent();
```

```
    bob = new Guy();
    bob.Name = "Bob";
    bob.Cash = 100;
```

```
    joe = new Guy();
    joe.Name = "Joe";
    joe.Cash = 50;
```

Then we do the same for the second instance of the `Guy` class.

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

```
    UpdateForm();
}
```

there are no Dumb Questions

Make sure you save the project now—we'll come back to it in a few pages.

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`"?

A: You'll run into problems—your form won't work, because it won't ever set the form's `bob` variable. Think about it for a minute, and you'll see why it works that way. If you have this code at the top of your form:

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

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

```
Guy bob = new Guy();
```

then you've declared **two** variables. It's a little confusing, because they both have the same name. But one of them is valid throughout the entire form, and the other one—the new one you added—is only valid inside the method. The next line (`bob.Name = "Bob";`) only updates that **local** variable, and doesn't touch the one in the form. So when you try to run your code, it'll give you a nasty error message ("NullReferenceException not handled"), which just means you tried to use an object before you created it with `new`.

There's an easier way to initialize objects

Almost every object that you create needs to be initialized in some way. And the `Guy` object is no exception—it's useless until you set its `Name` and `Cash` fields. It's so common to have to initialize fields that `C#` gives you a shortcut for doing it called an **object initializer**. And the IDE's IntelliSense will help you do it.

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

- 1 Here's the original code that you wrote to initialize Joe's `Guy` object.

```
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.

```
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.

```
joe = new Guy() {
```



- 4 Press tab to tell it to add the `Cash` field. Then set it equal to 50.

```
joe = new Guy() { Cash = 50
```

- 5 Type in a comma. As soon as you do, the other field shows up.

```
joe = new Guy() { Cash = 50,
```



- 5 Finish the object initializer. Now you've saved yourself two lines of code!

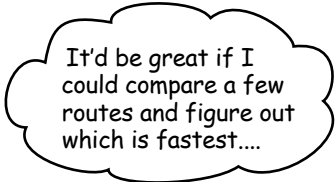
```
joe = new Guy() { Cash = 50, Name = "Joe" };
```

This new declaration does exactly the same thing as the three lines of code you wrote originally. It's just shorter and easier to read.

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.



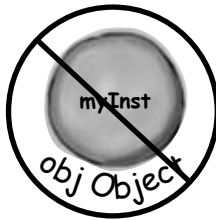
★ **What real-world things will your program use?**

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



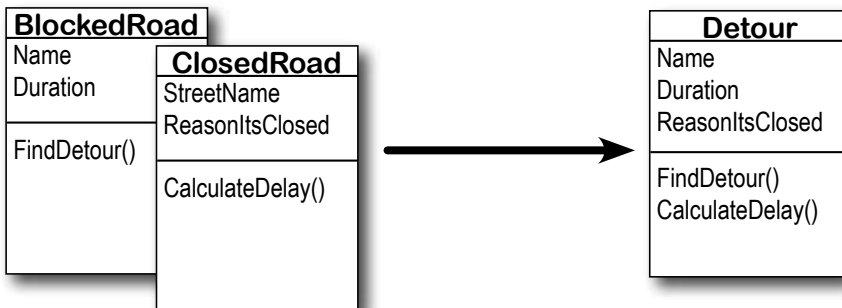
★ **Use descriptive names for classes and methods.**

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.





Exercise

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

1 Use an object initializer to initialize Bob's instance of Guy

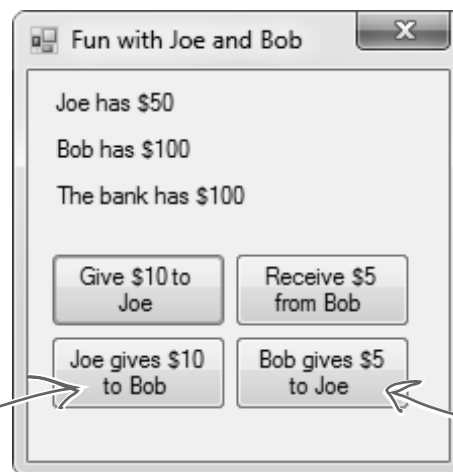
You've already done it with Joe. Now make Bob's instance work with an object initializer too.

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`.

This button tells Joe to give 10 bucks to Bob, so you should use the “(Name)” row in the Properties window to name it `joeGivesToBob`.



This button tells Bob to give 5 bucks to Joe. Name it `bobGivesToJoe`.

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.



Exercise Solution

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

```
public partial class Form1 : Form {
    Guy joe;
    Guy bob;
    int bank = 100;

    public Form1() {
        InitializeComponent();
        bob = new Guy() { Cash = 100, Name = "Bob" };
        joe = new Guy() { Cash = 50, Name = "Joe" };
        UpdateForm();
    }

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

    private void button1_Click(object sender, EventArgs e) {
        if (bank >= 10) {
            bank -= joe.ReceiveCash(10);
            UpdateForm();
        } else {
            MessageBox.Show("The bank is out of money.");
        }
    }

    private void button2_Click(object sender, EventArgs e) {
        bank += bob.GiveCash(5);
        UpdateForm();
    }

    private void joeGivesToBob_Click(object sender, EventArgs e) {
        bob.ReceiveCash(joe.GiveCash(10));
        UpdateForm();
    }

    private void bobGivesToJoe_Click(object sender, EventArgs e) {
        joe.ReceiveCash(bob.GiveCash(5));
        UpdateForm();
    }
}
```

Here are the object initializers for the two instances of the Guy class. Bob gets initialized with 100 bucks and his name.

To make Joe give cash to Bob, we call Joe's GiveCash() method and send its results into Bob's ReceiveCash() method.

Take a close look at how the Guy methods are being called. The results returned by GiveCash() are pumped right into ReceiveCash() as its parameter.

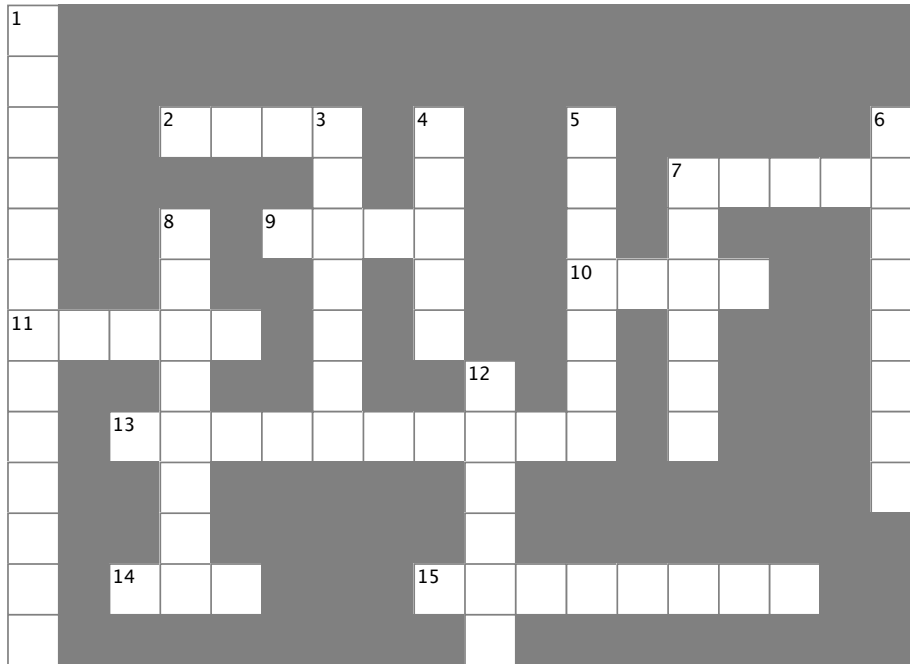
The trick here is thinking through who's giving the cash and who's receiving it.

Before you go on, take a minute and flip to #1 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
{
    private void button1_Click(object sender, EventArgs e)
    {
        String result = "";
        Echo e1 = new Echo();
        Echo e2 = new Echo();
        int x = 0;
        while ( x < 4 ) {
            result = result + e1.Hello() + "\n";
            e1.count = e1.count + 1;
            if ( x == 3 ) {
                e2.count = e2.count + 1;
            }
            if ( x > 0 ) {
                e2.count = e2.count + e1.count;
            }
            x = x + 1;
        }
        MessageBox.Show(result + "Count: " + e2.count);
    }

    class Echo {
        public int count = 0;
        public string Hello() {
            return "helloooo...";
        }
    }
}
```

That's the correct answer.

And here's the bonus answer!

`Echo e2 = e1;`

but wait... there's more!

Hey, you! Are you looking for a way to get productive fast with C#, .NET and Visual Studio 2010?

I'm still hungry for more!

Thanks for reading the first three chapters of Head First C#, the fastest way to learn to program with C# and the .NET framework. Like what you read so far? Then have a look at the next few pages to see what's coming next!



Do you want to get C# programming concepts and ideas stuck in your brain... fast?

Do you want to learn C# by building projects and solving puzzles?

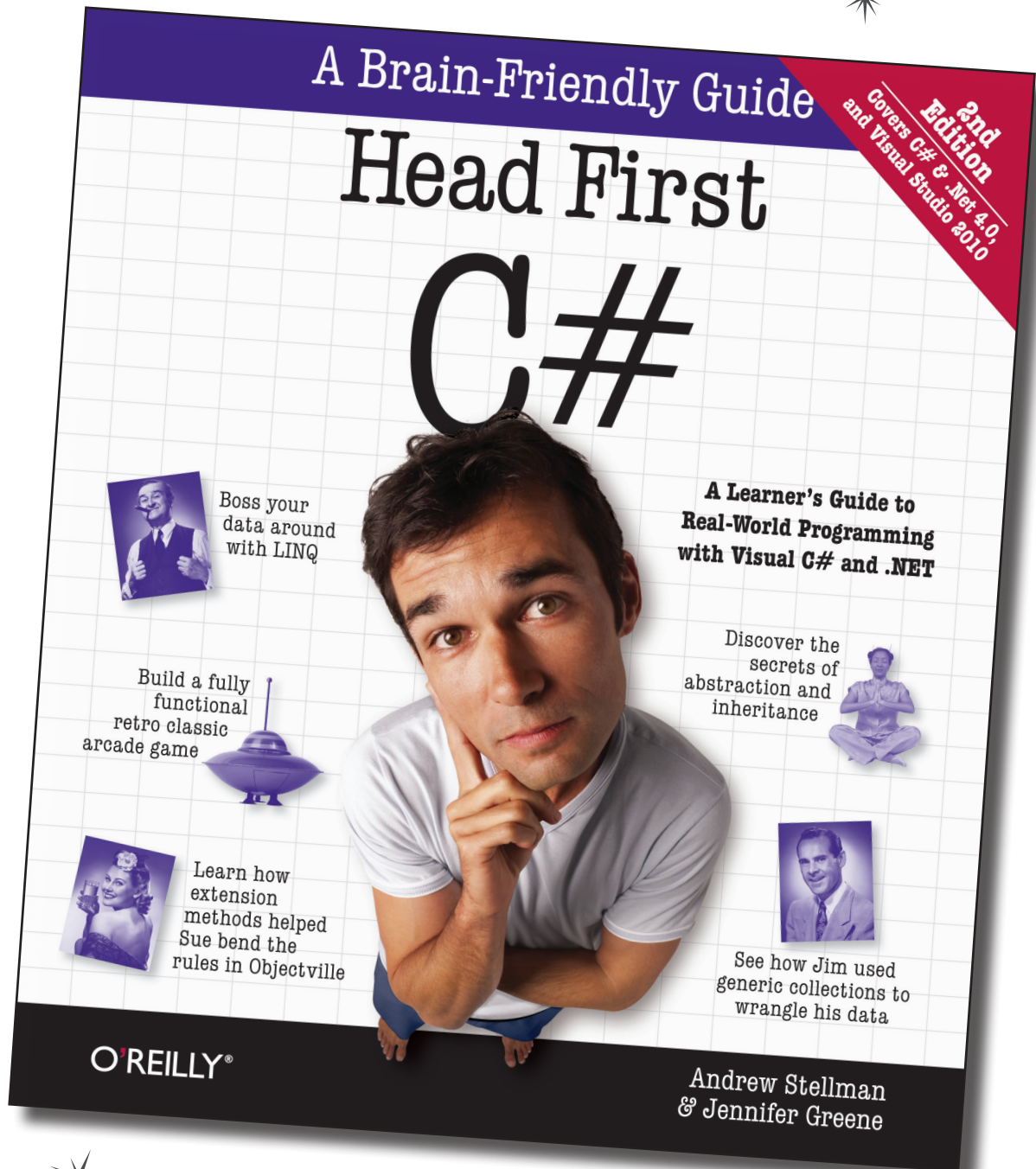
Wouldn't it be dreamy if there was a C# book that was more fun than endlessly debugging code? It's probably nothing but a fantasy....

Are you the kind of person who likes to learn by doing, rather than poring through pages and pages of dry reference material?



Looking for the easiest way to become a great C# programmer?

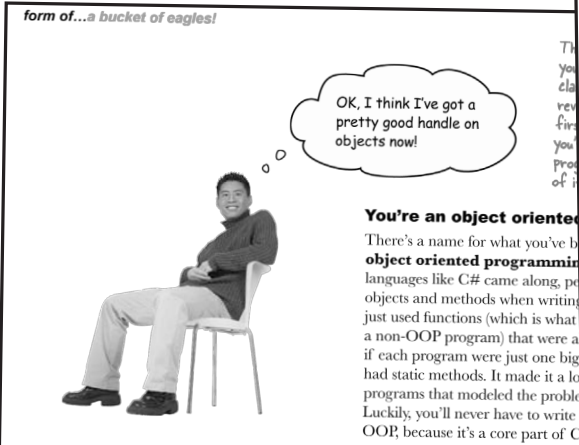
Look no further! All this and more is here today. Introducing
Head First C#, 2nd edition!



intrigued? flip the page to find out more! ▶

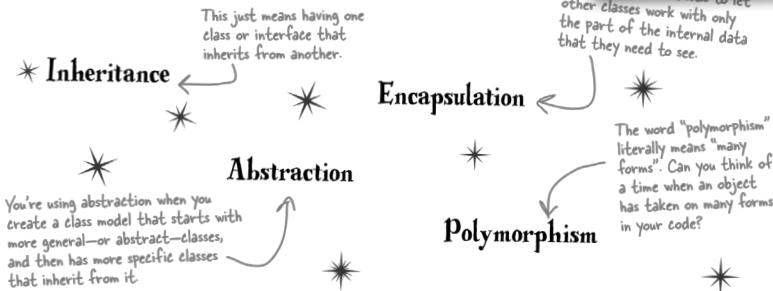
Get a handle on C#

Head First C# is a complete learning experience for programming with C#, the .NET Framework, and the Visual Studio IDE. Built for your brain, this book covers C# and .NET 4.0 and Visual Studio 2010, and teaches everything from inheritance to serialization.



The four principles of object oriented programming

When programmers talk about OOP, they're referring to four important principles. They should seem very familiar to you by now because you've been working with every one of them. You'll recognize the first three principles just from their names: **inheritance**, **abstraction**, and **encapsulation**. The last one's called **polymorphism**. It sounds a little odd, but it turns out that you already know all about it too.



spy versus spy

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

When you make all of your fields and methods public, any other class can access them. Everything your class does and knows about becomes an open book for every other class in your program...and you just saw how that can cause your program to behave in ways you never expected. Encapsulation lets you control what you share and what you keep private inside your class. Let's see how this works:

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



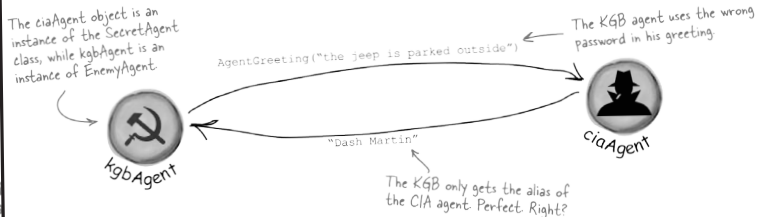
```
RealName: "Herb Jones"
Alias: "Dash Martin"
Password: "the crow flies at midnight"
```

SecretAgent
Alias
RealName
Password
AgentGreeting()

- 2 Agent Jones has a plan to help him evade the enemy KGB agents. He added an `AgentGreeting()` method that takes a password as its parameter. If he doesn't get the right password, he'll only reveal his alias, Dash Martin.

EnemyAgent
Borscht
Vodka
ContactComrades()
OverthrowCapitalists()

- 3 Seems like a foolproof way to protect the agent's identity, right? As long as the agent object that calls it doesn't have the right password, the agent's name is safe.



You'll query your data with LINQ, draw graphics and animation, and learn all about classes and object oriented programming, all through building games, doing hands-on projects, and solving puzzles. You'll become a solid C# programmer, and you'll have a great time along the way!

Flip through the next few pages to get a sneak peek at the topics you'll learn about!



Exercise

Create five random cards and then sort them.

- 1 **Create code to make a jumbled set of cards**
Create a new Console Application and add code to the `Main()` method that creates five random `Card` objects. After you create each object, use the built-in `Console.WriteLine()` method to write its name to the output. Use `Console.ReadKey()` at the end of the program to keep your window from disappearing when the program finishes.
- 2 **Create a class that implements `IComparer<Card>` to sort the cards**
Here's a good chance to use that IDE shortcut to implement an interface:

```
class CardComparer_byValue : IComparer<Card>
```

Then click on `IComparer<Card>` and hover over the `I`. You'll see a box appear underneath it. When you click on the box, the IDE pops up its "Implement interface" window:

Sometimes it's a little hard to get this box to pop up, so the IDE has a useful shortcut: just press `ctrl-period`.

`IComparer<Card>`

Implement
Explicitly in

Click on "Implement interface `IComparer<Card>`" in all of the methods and properties that you need. `Compare()` method to compare two cards, `x` and `y`, `-1` if it's smaller, and `0` if they're the same, and `1` if it's bigger than `y`, `-1` if it's smaller, and `0` if they're the same. `CompareTo()` comes after any jack, which comes after any four, which comes after any three.

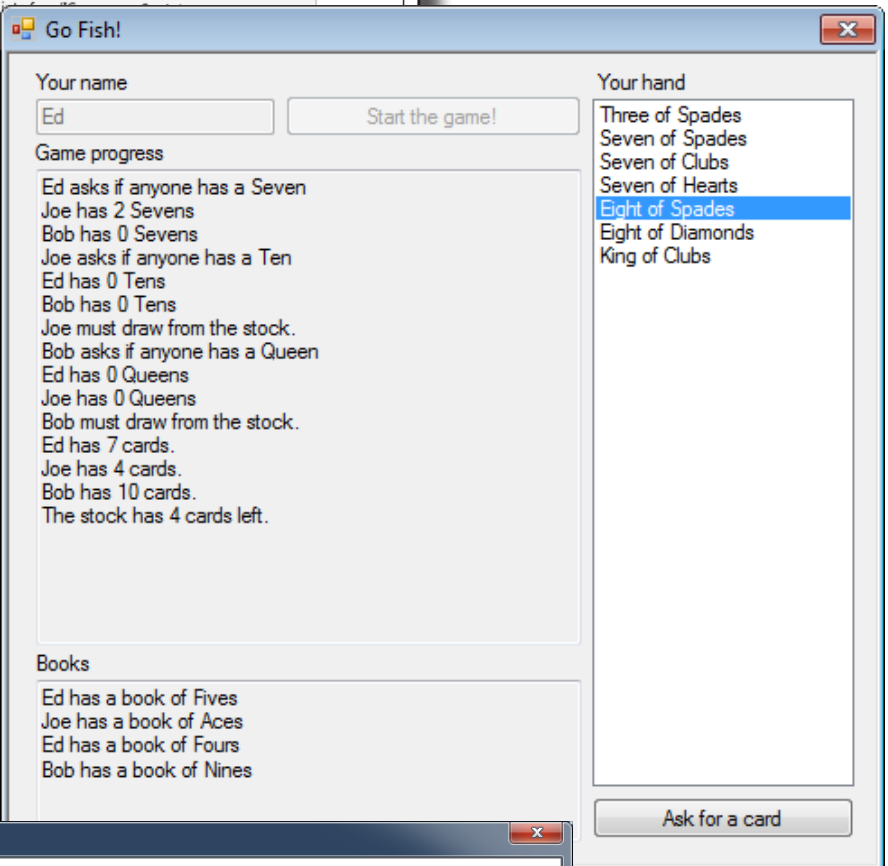
- 3 **Make sure the output looks right**

Here's what your output window should look like at the end of the program:

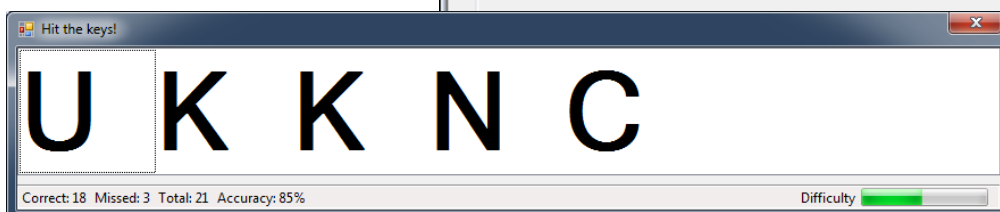
When you use the built-in `Console.WriteLine()` method, it adds a line to this output. `Console.ReadKey()` waits for you to press a key before the program ends.

```
file:///C:/Users/andrew/document
Five random cards:
Queen of Clubs
Two of Diamonds
Seven of Spades
Ten of Diamonds
Three of Clubs
These same cards, sorted:
Two of Diamonds
Three of Clubs
Seven of Spades
Ten of Diamonds
Queen of Clubs
```

Build over **100** different projects! You'll build everything from a card game and a text editor to a full-blown retro classic arcade game.



Test your typing speed with this "Hit the keys!" game that you'll build in chapter 4.



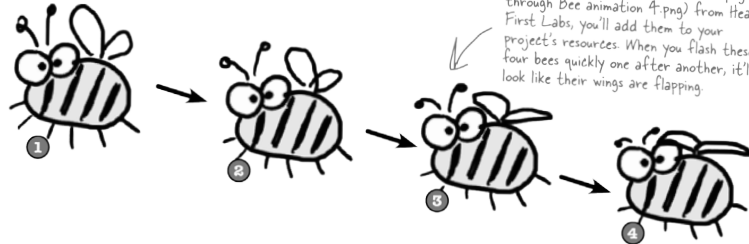
Play `Go Fish!` against computer opponents in this game you'll build in chapter 8.

Build fully animated graphics programs. You'll be amazed at how slick your apps look once you master graphics!

Build your first animated control

You're going to **build your own control** that draws an animated bee picture. If you've never done animation, it's not as hard as it sounds; you draw a sequence of pictures one after another, and produce the illusion of movement. Lucky for us, the way C# and .NET handle resources makes it really easy for us to do animation.

Once you download the four bee animation pictures (Bee animation 1.png through Bee animation 4.png) from Head First Labs, you'll add them to your project's resources. When you flash these four bees quickly one after another, it'll look like their wings are flapping.

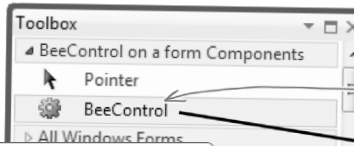


We want a control in the toolbox

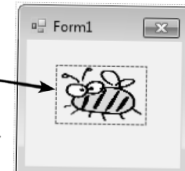
If you build BeeControl right, it'll appear as a control that you can drag out of your toolbox and onto your form. It'll look just like a PictureBox showing a picture of a bee, except that it'll have animated flapping wings.

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

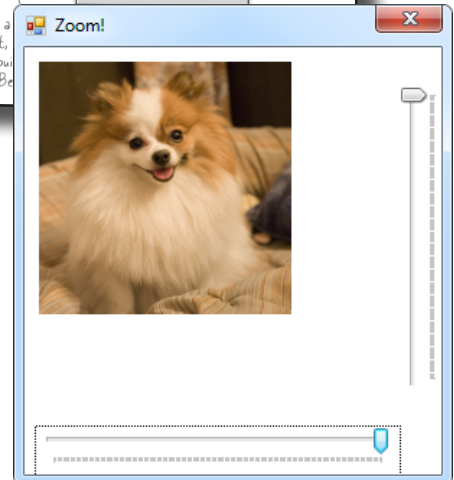
www.headfirstlabs.com/books/hfesharp/



As long as we extend the right classes, NET takes care of showing our control in the IDE toolbox.



This is like a PictureBox when an image is set, that we'll build what class Be...



Simulate life in a beehive with this animated beehive simulator.

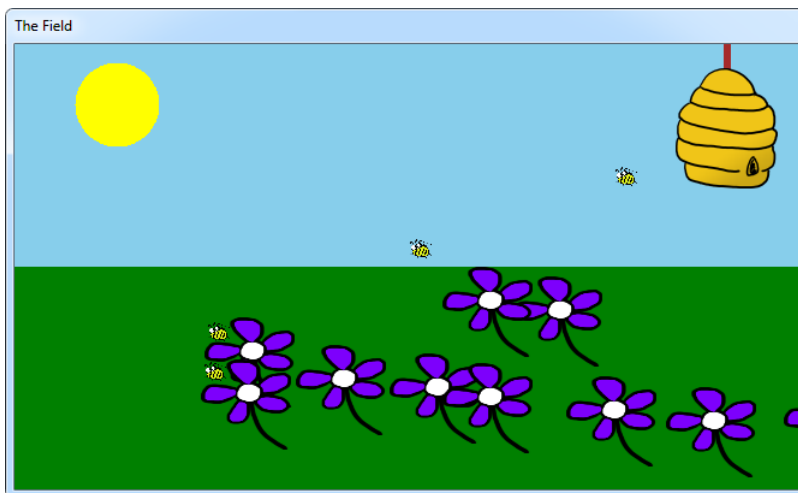
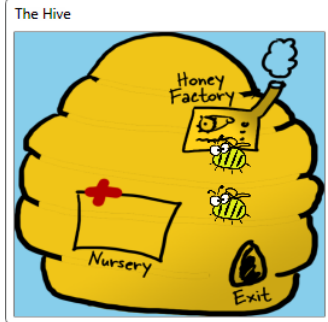
Beehive Simulator

Pause simulation Reset

# Bees	6
# Flowers	11
Total honey in the hive	1,200
Total nectar in the field	34,390
Frames run	983
Frame rate	16 (62.5ms)

Idle: 1 bee
 Flying To Flower: 2 bees
 Gathering Nectar: 1 bee
 Returning To Hive: 2 bees

Bee #1: Idle



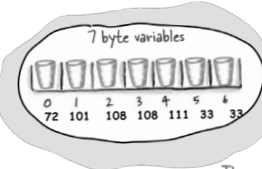
C# can use byte arrays to move data around

Since all your data ends up encoded as **bytes**, it makes sense to think of a file as one **big byte array**. And you already know how to read and write byte arrays.

Here's the code to create a byte array, open an input stream, and read data into bytes 0 through 6 of the array.



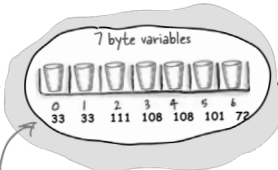
```
byte[] greeting;
greeting = File.ReadAllBytes(filename);
```



These numbers are the Unicode numbers for the characters in "Hello!!"

This is a static method for Arrays that reverses the order of the bytes. We're just using it to show that the changes you make to the byte array get written out to the file exactly.

```
Array.Reverse(greeting);
File.WriteAllBytes(filename, greeting);
```



Now the bytes are in reverse order.

When the program writes the byte array out to a file, the text is in reverse order too.



Reversing the bytes in "Hello!!" only works because each of those characters is one byte long. Can you figure out why this won't work for 017W?

From theory to practice, you'll cover everything from object oriented programming to file I/O to querying your data with LINQ. It's a complete C# learning experience!

Management system

out of control, and he's got a beehive full that need to be done control of which bee the beeper to do

ent system to help it'll work:

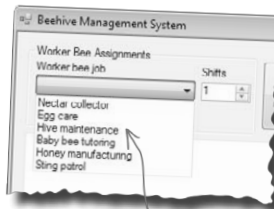


inheritance

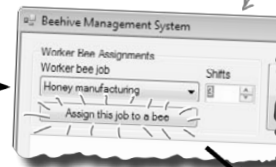


her workers

There are six possible jobs that the workers can do. Some know how to collect nectar and manufacture honey, others can maintain the hive and patrol for enemies. A few bees can do every job in the hive. So your program will need to give her a way to assign a job to any bee that's available to do it.

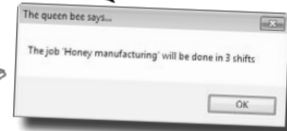


This drop-down list shows all six jobs that the workers can do. The queen knows what jobs need to be done, and she doesn't really care which bee does each job. So she just selects which job has to be done—the program will figure out if there's a worker available to do it and assign the job to him.



The bees work shifts, and most jobs require more than one shift. So the queen enters the number of shifts the job will take, and clicks the "Assign this job" button.

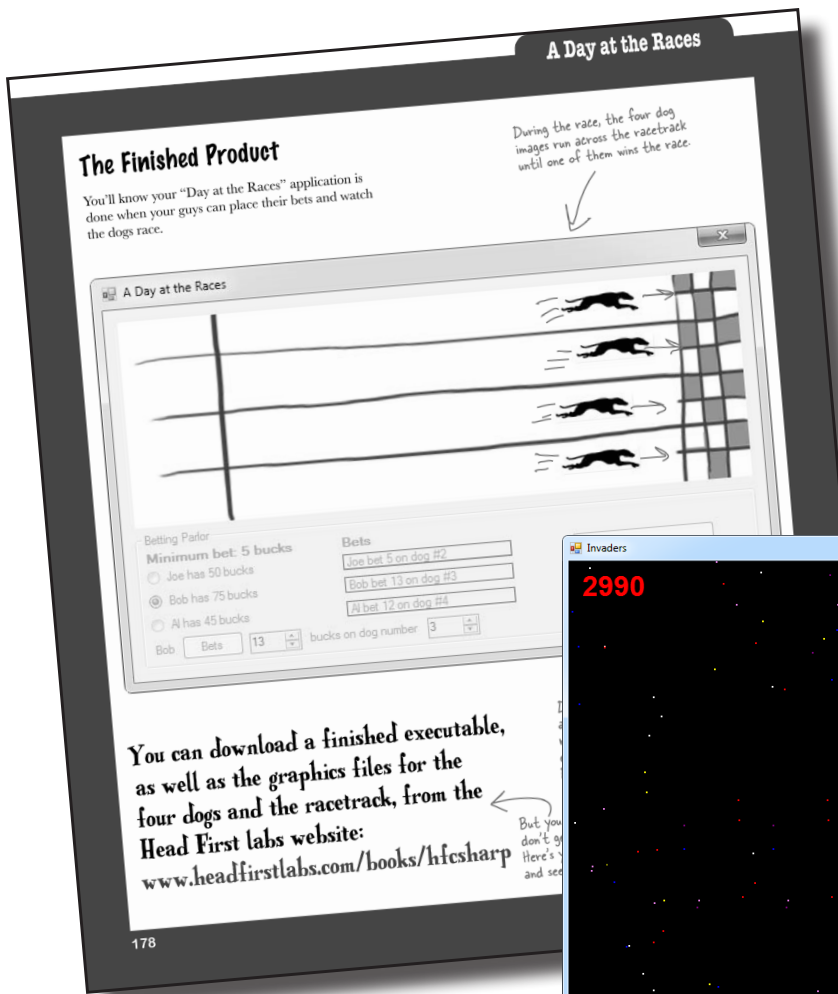
If there's a bee available to do the job, the program assigns the job to the bee and lets the queen know it's taken care of.



When the jobs are all assigned, it's time to work

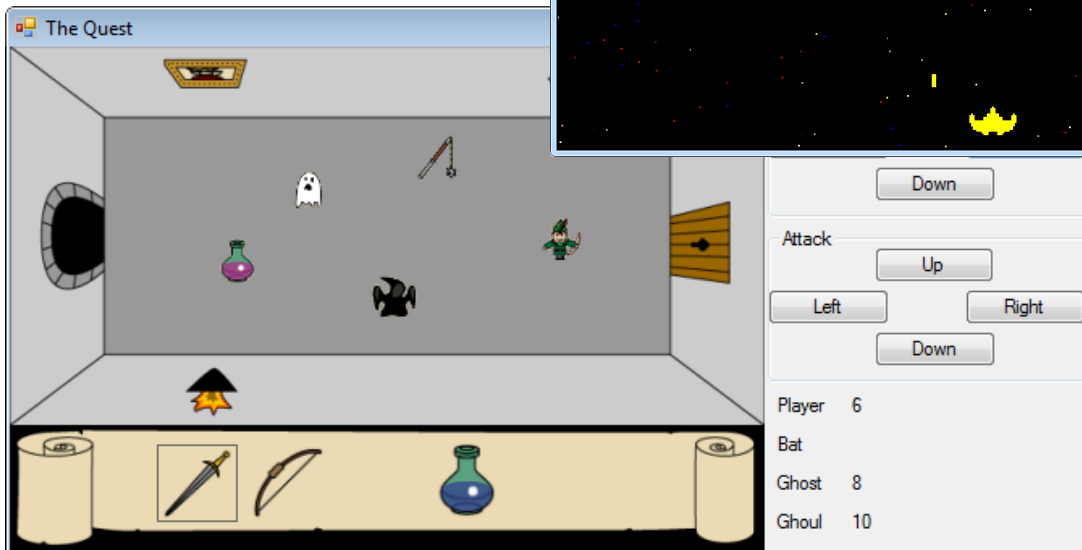
Once the queen's done assigning the work, she'll tell the bees to work the next shift by clicking the "Work the next shift" button. The program then generates a shift report that tells her which bees worked that shift, what jobs they did, and how many more shifts they'll be working each job.





The best way to learn C# is to write lots of code

Every few chapters you will come across a lab that lets you apply what you've learned up to that point. Each lab is designed to simulate a professional programming task, increasing in complexity until-at last-you build a working Invaders game, complete with shooting ships, aliens descending while firing, and an animated death sequence for unlucky starfighters. **This remarkably engaging book will have you going from zero to 60 with C# in no time flat.**



The screenshot shows the O'Reilly website page for the book "Head First C#, Second Edition". The page includes a navigation menu, the O'Reilly logo, and a product listing with a book cover image. Below the listing are promotional offers like "Buy 2 Get 1 FREE" and "Free Shipping". There is also a "Description" section and a "Got a Question?" section with a form.

Order yours today!

Available at fine bookstores,
and wherever
O'REILLY®
books are sold.

The fun's just beginning!
Learn more at the Head
First C# website.



<http://www.headfirstlabs.com/books/hfsharp/>