LEARNING MADE EASY

6th Edition

Beginning Programming with Java®

for dummies®
A Wiley Brand

```
com.project.hello.w
blic class HelloWorld {
public static void main(St
System.out.println("H
}
```

Learn basic programming concepts and methods

Build a foundation of code before writing your own program

Explore the new features in Java 17

Barry Burd, PhD
Java Champion and author of
Java For Dummies

Beginning Programming with Java®

for dummies®
A Wiley Brand

Beginning Programming with Java®

6th Edition

by Barry Burd

for dummies®
A Wiley Brand

Beginning Programming with Java® For Dummies®, 6th Edition

Published by: **John Wiley & Sons, Inc.,** 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

Copyright © 2021 by John Wiley & Sons, Inc., Hoboken, New Jersey

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Trademarks: Wiley, For Dummies, the Dummies Man logo, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and may not be used without written permission. Java is a registered trademark of Oracle America, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For technical support, please visit https://hub.wiley.com/community/support/dummies.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2021944241

ISBN: 978-1-119-80691-2; 978-1-119-80692-9 (ebk); 978-1-119-80693-6 (ebk)

Contents at a Glance

Table of Contents

Introduction

What's your story?

- Are you a working stiff, interested in knowing more about the way your company's computers work?
- Are you a student who needs to complete some extra reading in order to survive a beginning computer course?
- Are you a typical computer user — you've done lots of word processing and you want to do something more interesting with your computer?
- Are you a job seeker with an interest in entering the fast-paced, glamorous, high-profile world of computer programming (or, at least, the decent-paying world of computer programming)?

Well, if you want to write computer programs, this book is for you. This book avoids the snobby of-course-you-already-know assumptions and describes computer programming from scratch.

About This Book

The book uses Java — a powerful, general-purpose computer programming language. But Java's subtleties and eccentricities aren't the book's main focus. Instead, this book emphasizes a process — the process of creating instructions for a computer to follow. Many highfalutin books describe the mechanics of this process — the rules, the conventions, and the formalisms. But those other books aren't written for real people. Those books don't take you from where you are to where you want to be.

In this book, I assume very little about your experience with computers. As you read each section, you get to see inside my mind. You see the problems that I face, the things that I think, and the solutions that I find. Some problems are the kind that I remember facing when I was a novice; other problems are the kind that I face as an expert. I help you understand, I help you visualize, and I help you create solutions on your own. I even get to tell a few funny stories.

Foolish Assumptions

In this book, I make a few assumptions about you, the reader. If one of these assumptions is incorrect, you're probably okay. If all these assumptions are incorrect . . . well, buy the book anyway.

- **I assume that you have access to a computer.** Here's good news. You can run the code in this book on almost any computer. The only computers you can't use to run this code are ancient things that are more than eight years old (give or take a few years). You can run the latest version of Java on Windows, Macintosh, and Linux computers.
- **I assume that you can navigate your computer's common menus and dialog boxes.** You don't have to be a Windows, Linux, or Macintosh power user, but you should be able to start a program, find a file, put a file into a certain directory — that sort of thing. Most of the time, when you practice the stuff in this book, you're typing code on your keyboard, not pointing and clicking the mouse.

  On those rare occasions when you need to drag and drop, cut and paste, or plug and play, I guide you carefully through the steps. But your computer may be configured in any of several billion ways, and my instructions may not quite fit your special situation. So, when you reach one of these platform-specific tasks, try following the steps in this book. If the steps don't quite fit, send me an email message or consult a book with instructions tailored to your system.
- **I assume that you can think logically.** That's all there is to computer programming — thinking logically. If you can think logically, you have it made. If you don't believe that you can think logically, read on. You may be pleasantly surprised.
- **I assume that you know little or nothing about computer programming.** This isn't one of those all-things-to-all-people books. I don't please the novice while I tease the expert. I aim this book specifically toward the novice — the person who has never programmed a computer or has never felt comfortable programming a computer. If you're one of these people, you're reading the right book.

Icons Used in This Book

If you could watch me write this book, you'd see me sitting at my computer, talking to myself. I say each sentence several times in my head. When I have an extra thought, a side comment, or something that doesn't belong in the regular stream, I twist my head a little bit. That way, whoever's listening to me (usually nobody) knows that I'm off on a momentary tangent.

Of course, in print, you can't see me twisting my head. I need some other way of setting a side thought in a corner by itself. I do it with icons. When you see a Tip icon or a Remember icon, you know that I'm taking a quick detour.

Here's a list of icons that I use in this book:

TIP

A tip is an extra piece of information — something helpful that the other books may forget to tell you.

WARNING

Everyone makes mistakes. Heaven knows that I've made a few in my time. Anyway, when I think of a mistake that people are especially prone to make, I write about the mistake in a Warning icon.

REMEMBER

Sometimes I want to hire a skywriting airplane crew. "Barry," says the white smoky cloud, "if you want to compare two numbers, use the double equal sign. Please don't forget to do this." Because I can't afford skywriting, I have to settle for something more modest. I create a paragraph with the Remember icon.

TRY IT OUT

Writing computer code is an activity, and the best way to learn an activity is to practice it. That's why I've created things for you to try in order to reinforce your knowledge. Many of these are confidence-builders, but some are a bit more challenging. When you first start putting things into practice, you discover all kinds of issues, quandaries, and roadblocks that didn't occur to you when you started reading about the material. But that's a good thing. Keep at it! Don't become frustrated. Or, if you do become frustrated, visit this book's website (`http://beginprog.allmycode.com`) for hints and solutions.

TECHNICAL STUFF

Occasionally, I run across a technical tidbit. The tidbit may help you understand what the people behind the scenes (the people who developed Java) were thinking. You don't have to read it, but you may find it useful. You may also find the tidbit helpful if you plan to read other (more geeky) books about Java.

ON THE WEB

This icon calls attention to useful material that you can find online. (You don't have to wait long to see one of these icons. I use one at the end of this introduction!)

CROSS REFERENCE

"If you don't remember what such-and-such means, see blah-blah-blah," or "For more information, read blahbity-blah-blah."

Beyond the Book

In addition to what you're reading right now, this book comes with a free access-anywhere Cheat Sheet containing code that you can copy and paste into your own Java program. To get this Cheat Sheet, simply go to `www.dummies.com` and type **Beginning Programming with Java For Dummies Cheat Sheet** in the Search box.

Where to Go from Here

If you've gotten this far, you're ready to start reading about computer programming. Think of me (the author) as your guide, your host, your personal assistant. I do everything I can to keep things interesting and, most importantly, help you understand.

ON THE WEB

If you like what you read, send me an email, post on my Facebook wall, or give me a tweet. My email address, which I created just for comments and questions about this book, is `BeginProg@allmycode.com`. My Facebook page is `/allmycode`, and my Twitter handle is `@allmycode`. And don't forget: To get the latest information, visit this book's support website: `http://beginprog.allmycode.com`.

1 Getting Started with Java Programming

IN THIS PART . . .

Getting psyched up to be a Java developer

Installing the software

Running some sample programs

IN THIS CHAPTER

- **Recognizing what computer programming is all about**
- **Understanding the software that enables you to write programs**
- **Revving up to use an integrated development environment**

Chapter 1

The Big Picture

Computer programming? What's that? Is it technical? Does it hurt? Is it politically correct? Does Google control it? Why would anyone want to do it? And what about me? Can I learn to do it?

What's It All About?

You've probably used a computer to do word processing: Type a letter, print it, and then send the printout to someone you love. If you have easy access to a computer, you've probably surfed the web: Visit a page, click a link, and see another page. It's easy, right?

Well, it's easy only because someone told the computer exactly what to do. If you take a computer fresh from the factory and give no instructions to it, it can't do word processing, it can't surf the web, and it can't do anything. All a computer can do is follow the instructions that people give to it.

Now imagine that you're using Microsoft Word to write the great American novel and you come to the end of a line. (You're not at the end of a sentence; just the end of a line.) As you type the next word, the computer's cursor jumps automatically to the next line of type. What's going on here?

Well, someone wrote a *computer program* — a set of instructions telling the computer what to do. Another name for a program (or part of a program) is *code.* Listing 1-1 shows you what some of Microsoft Word's code may look like.

LISTING 1-1: A Few Lines in a Computer Program

```
if (columnNumber> 60) {
    wrapToNextLine();
} else {
    continueSameLine();
}
```

If you translate Listing 1-1 into plain English, you get something like this:

```
If the column number is greater than 60,
    then go to the next line.
Otherwise (if the column number isn't greater than 60),
    then stay on the same line.
```

Somebody has to write code of the kind shown in Listing 1-1. This code, along with millions of other lines of code, makes up the program called Microsoft Word.

And what about web surfing? You click a link that's supposed to take you directly to Facebook. Behind the scenes, someone has written code of the following kind:

```
Go to <a href="http://www.facebook.com">Facebook</a>.
```

One way or another, someone has to write a program. That someone is called a *programmer.*

Telling a computer what to do

Everything you do with a computer involves gobs and gobs of code. For example, every computer game is really a big (make that "very big"!) bunch of computer code. At some point, someone had to write the game program:

```
if (person.touches(goldenRing)) {
    person.getPoints(10);
}
```

Without a doubt, the people who write programs have valuable skills. These people have two important qualities:

- They know how to break big problems into smaller, step-by-step procedures.
- They can express these steps in a precise language.

A language for writing steps is called a *programming language*, and Java is just one of several thousand useful programming languages. The stuff in Listing 1-1 is written in the Java programming language.

James Gosling and others at Sun Microsystems created Java in the early to mid-1990s. In 2010, Java became part of Oracle Corporation as part of Oracle's acquiring Sun Microsystems.

Pick your poison

This book isn't about the differences among programming languages, but you should see code in some other languages so that you understand the bigger picture. For example, there's another language, Visual Basic, whose code looks a bit different from code written in Java. An excerpt from a Visual Basic program may look like this:

```
If columnNumber > 60 Then
    Call wrapToNextLine
Else
    Call continueSameLine
End If
```

The Visual Basic code looks more like ordinary English than the Java code in Listing 1-1. But, if you think that Visual Basic is like English, then just look at some code written in COBOL:

```
IF COLUMN-NUMBER IS GREATER THAN 60 THEN
    PERFORM WRAP-TO-NEXT-LINE
ELSE
    PERFORM CONTINUE-SAME-LINE
END-IF.
```

At the other end of the spectrum, you find languages like Forth. Here's a snippet of code written in Forth:

```
: WRAP? 60 > IF WRAP_TO_NEXT_LINE? ELSE CONTINUE_SAME_LINE? THEN ;
```

Computer languages can be very different from one another, but in some ways, they're all the same. When you get used to writing `IF COLUMN-NUMBER IS GREATER THAN 60`, you can also become comfortable writing `if (columnNumber> 60)`.

It's just a mental substitution of one set of symbols for another. Eventually, writing things like `if (columnNumber> 60)` becomes second nature.

From Your Mind to the Computer's Processor

When you create a new computer program, you complete a multistep process. The process involves three important tools:

- **Compiler:** A compiler translates your code into computer-friendly (human-unfriendly) instructions.
- **Virtual machine:** A virtual machine steps through the computer-friendly instructions.
- **Application programming interface:** An application programming interface contains useful prewritten code.

The next three sections describe each of the three tools.

Translating your code

You may have heard that computers deal with zeros and ones. That's certainly true, but what does it mean? Well, for starters, computer circuits don't deal directly with letters of the alphabet. When you see the word *Start* on your computer screen, the computer stores the word internally as `01010011 01110100 01100001 01110010 01110100`. That feeling you get of seeing a friendly-looking, five-letter word is your interpretation of the computer screen's pixels and nothing more. Computers break everything down into very low-level, unfriendly sequences of zeros and ones and then put things back together so that humans can deal with the results.

So, what happens when you write a computer program? Well, the program has to get translated into zeros and ones. The official name for the translation process is *compilation.* Without compilation, the computer can't run your program.

I compiled the code in Listing 1-1. Then I did some harmless hacking to help me see the resulting zeros and ones. What I saw was the mishmash in Figure 1-1.

```
11001010 11111110 10111010 10111110 00000000 00000000
00000000 00101110 00000000 00010101 00001010 00000000
00000101 00000000 00010000 00001010 00000000 00000100
00000000 00010001 00001010 00000000 00000100 00000000
00010010 00000111 00000000 00010011 00000111 00000000
00010100 00000001 00000000 00000110 00111100 01101001
01101110 01101001 01110100 00111110 00000001 00000000
00000011 00101000 00101001 01010110 00000001 00000000
00000100 01000011 01101111 01100100 01100101 00000001
00000000 00001111 01001100 01101001 01101110 01100101
01001110 01110101 01101101 01100010 01100101 01110010
01010100 01100001 01100010 01101100 01100101 00000001
00000000 00001011 01100100 01101001 01110011 01110000
01101100 01100001 01111001 01010111 01101111 01110010
01100100 00000001 00000000 00000100 00101000 01001001
00101001 01010110 00000001 00000000 00001110 01110111
01110010 01100001 01110000 01010100 01101111 01001110
01100101 01111000 01110100 01001100 01101001 01101110
01100101 00000001 00000000 00010000 01100011 01101111
01101110 01110100 01101001 01101110 01110101 01100101
01010011 01100001 01101101 01100101 01001100 01101001
01101110 01100101 00000001 00000000 00001010 01010011
01101111 01110101 01110010 01100011 01100101 01000110
```

FIGURE 1-1: My computer understands these zeros and ones, but I don't.

The compiled mumbo jumbo in Figure 1-1 goes by many different names:

- Most Java programmers call it *bytecode.*
- I often call it a *`.class` file.* That's because, in Java, the bytecode gets stored in files named *`SomethingOrOther`*`.class`.
- To emphasize the difference, Java programmers call Listing 1-1 the *source code* and refer to the zeros and ones in Figure 1-1 as *object code.*

To visualize the relationship between source code and object code, see Figure 1-2. You can write source code and then get the computer to create object code from your source code. To create object code, the computer uses a special software tool called a *compiler.*

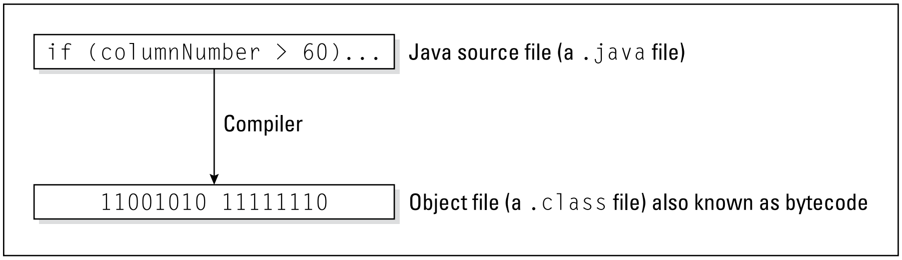
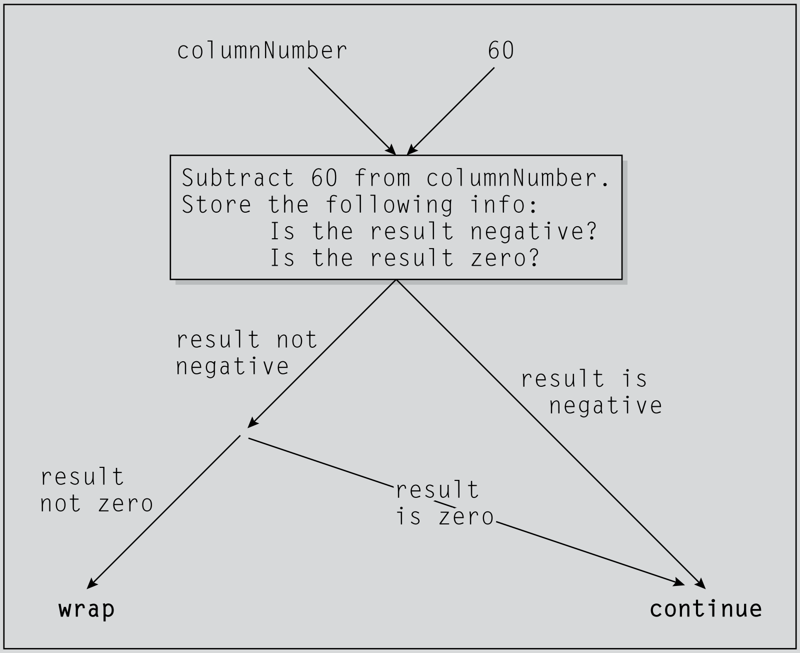

if (columnNumber > 60)...
Java source file (a .java file)
Compiler
11001010 11111110
Object file (a .class file) also known as bytecode

FIGURE 1-2: The computer compiles source code to create object code.

WHAT IS BYTECODE, ANYWAY?

Look at Listing 1-1 and at the listing's translation into bytecode in Figure 1-1. You may be tempted to think that a bytecode file is just a cryptogram — substituting zeros and ones for the letters in words like `if` and `else`. But it doesn't work that way at all. In fact, the most important part of a bytecode file is the encoding of a program's logic.

The zeros and ones in Figure 1-1 describe the flow of data from one part of your computer to another. I illustrate this flow in the following figure. But remember: This figure is just an illustration. Your computer doesn't look at this particular figure, or at anything like it. Instead, your computer reads a bunch of zeros and ones to decide what to do next.



Don't bother to absorb the details in my attempt at graphical representation in the figure. It's not worth your time. The thing you should glean from my mix of text, boxes, and arrows is that bytecode (the stuff in a `.class` file) contains a complete description of the operations that the computer is to perform. When you write a computer program, your source code describes an overall strategy — a big picture. The compiled bytecode turns the overall strategy into hundreds of tiny, step-by-step details. When the computer "runs your program," the computer examines this bytecode and carries out each of the little step-by-step details.

TECHNICAL STUFF

Your computer's hard drive may have a file named `javac` or `javac.exe`. This file contains that special software tool — the compiler. (Hey, how about that? The word *javac* stands for "Java compiler!") As a Java programmer, you often tell your computer to build some new object code. Your computer fulfills this wish by going behind the scenes and running the instructions in the `javac` file.

Running code

Several years ago, I spent a week in Copenhagen. I hung out with a friend who spoke both Danish and English fluently. As we chatted in the public park, I vaguely noticed some kids orbiting around us. I don't speak a word of Danish, so I assumed that the kids were talking about ordinary kid stuff.

Then my friend told me that the kids weren't speaking Danish. "What language are they speaking?" I asked.

"They're talking gibberish," she said. "It's just nonsense syllables. They don't understand English, so they're imitating you."

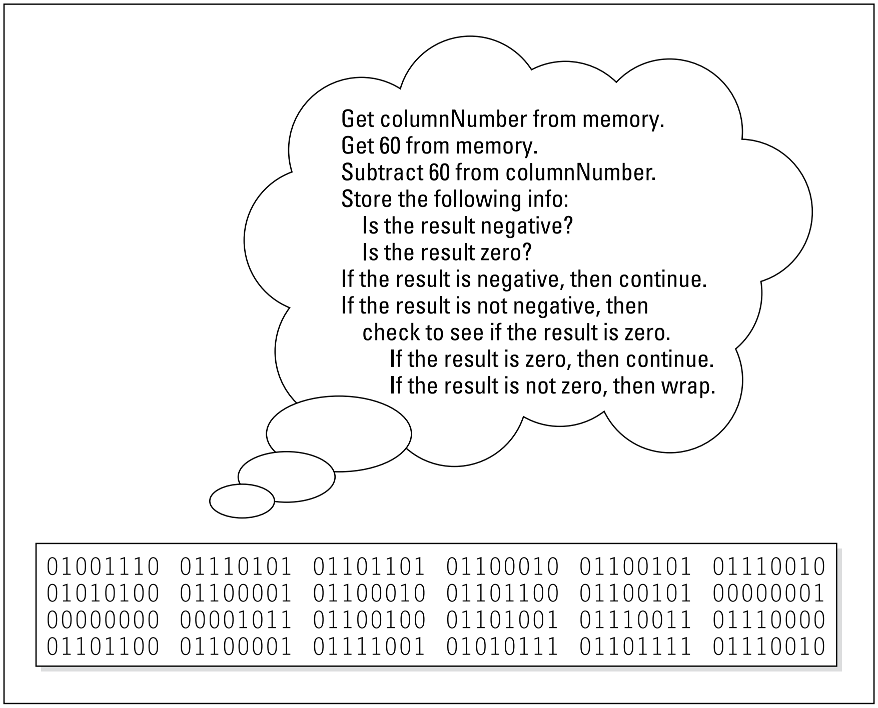
Now to return to present-day matters. I look at the stuff in Figure 1-1, and I'm tempted to make fun of the way my computer talks. But then I'd be just like the kids in Copenhagen. What's meaningless to me can make perfect sense to my computer. When the zeros and ones in Figure 1-1 percolate through my computer's circuits, the computer "thinks" the thoughts shown in Figure 1-3.

Everyone knows that computers don't think, but a computer can carry out the instructions depicted in Figure 1-3. With many programming languages (languages like C++ and COBOL, for example), a computer does exactly what I'm describing. A computer gobbles up some object code and does whatever the object code says to do.

That's how it works in many programming languages, but that's not how it works in Java. With Java, the computer executes a different set of instructions. The computer executes instructions like the ones in Figure 1-4.

The instructions in Figure 1-4 tell the computer how to follow other instructions. Instead of starting with `Get columnNumber from memory`, the computer's first instruction is, "Do what it says to do in the bytecode file." (Of course, in the bytecode file, the first instruction happens to be `Get columnNumber from memory`.)

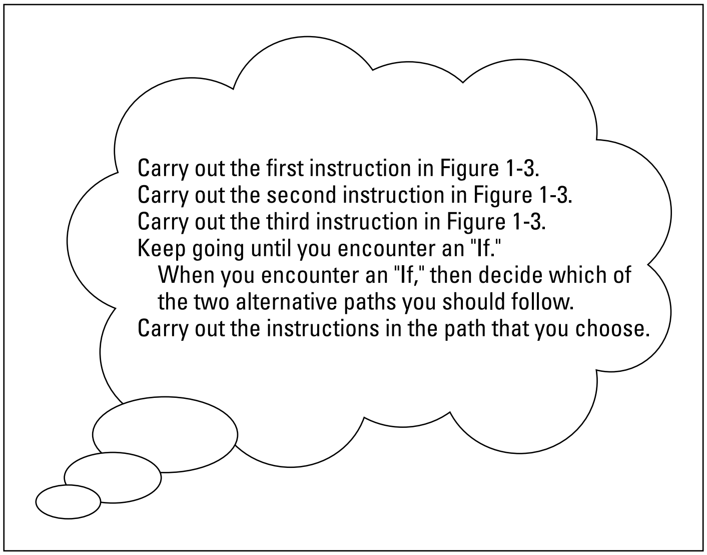
A special piece of software carries out the instructions in Figure 1-4. That special piece of software is called the *Java Virtual Machine (JVM).* The JVM walks your computer through the execution of some bytecode instructions. When you run a Java program, your computer is really running the JVM. That JVM examines your bytecode, zero by zero, one by one, and carries out the instructions described in the bytecode.



```
Get columnNumber from memory.
Get 60 from memory.
Subtract 60 from columnNumber.
Store the following info:
   Is the result negative?
   Is the result zero?
If the result is negative, then continue.
If the result is not negative, then
   check to see if the result is zero.
       If the result is zero, then continue.
       If the result is not zero, then wrap.
```

```
01001110  01110101  01101101  01100010  01100101  01110010
01010100  01100001  01100010  01101100  01100101  00000001
00000000  00001011  01100100  01101001  01110011  01110000
01101100  01100001  01111001  01010111  01101111  01110010
```

FIGURE 1-3: What the computer gleans from a bytecode file.



```
Carry out the first instruction in Figure 1-3.
Carry out the second instruction in Figure 1-3.
Carry out the third instruction in Figure 1-3.
Keep going until you encounter an "If."
   When you encounter an "If," then decide which of
   the two alternative paths you should follow.
Carry out the instructions in the path that you choose.
```

FIGURE 1-4: How a computer runs a Java program.

Many good metaphors can describe the JVM. Think of the JVM as a proxy, an errand boy, a go-between. One way or another, you have the situation shown in Figure 1-5. On the (a) side is the story you get with most programming languages — the computer runs some object code. On the (b) side is the story with Java — the computer runs the JVM, and the JVM follows the bytecode's instructions.

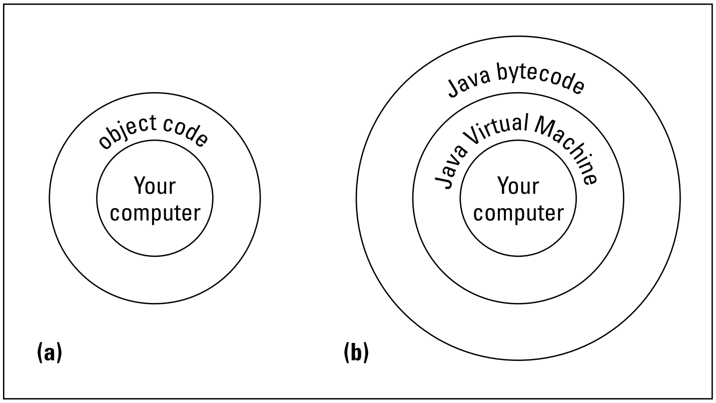


FIGURE 1-5: Two ways to run a computer program.

WRITE ONCE, RUN ANYWHERE

When Java first hit the tech scene in 1995, the language became popular almost immediately. This happened in part because of the JVM. The JVM is like a foreign language interpreter, turning Java bytecode into whatever native language a particular computer understands. So, if you hand my Windows computer a Java bytecode file, the computer's JVM interprets the file for the Windows environment. If you hand the same Java bytecode file to my colleague's Macintosh, the Macintosh JVM interprets that same bytecode for the Mac environment.

Look again at Figure 1-5. Without a virtual machine, you need a different kind of object code for each operating system. But with the JVM, just one piece of bytecode works on Windows machines, Unix boxes, Macs, or whatever. This is called *portability,* and in the computer programming world, portability is a precious commodity. Think about all the people using computers to browse the Internet. These people don't all run Microsoft Windows, but each person's computer can have its own bytecode interpreter — its own JVM.

The marketing folks at Oracle call it the *Write Once, Run Anywhere* model of computing. I call it a great way to create software.

TECHNICAL STUFF

Your computer's hard drive may have files named `javac` and `java` (or `javac.exe` and `java.exe`). A `java` (or `java.exe`) file contains the instructions illustrated previously, in Figure 1-4 — the instructions in the JVM. As a Java programmer, you often tell your computer to run a Java program. Your computer fulfills this wish by going behind the scenes and running the instructions in the `java` file.

Code you can use

During the early 1980s, my cousin-in-law Chris worked for a computer software firm. The firm wrote code for word processing machines. (At the time, if you wanted to compose documents without a typewriter, you bought a "computer" that did nothing but word processing.) Chris complained about being asked to write the same old code over and over again. "First, I write a search-and-replace program. Then I write a spell checker. Then I write another search-and-replace program. Then a different kind of spell checker. And then, a better search-and-replace."

How did Chris manage to stay interested in his work? And how did Chris's employer manage to stay in business? Every few months, Chris had to reinvent the wheel — toss out the old search-and-replace program and write a new program from scratch. That's inefficient. What's worse, it's boring.

For years, computer professionals were seeking the holy grail — a way to write software so that it's easy to reuse. Don't write and rewrite your search-and-replace code. Just break the task into tiny pieces. One piece searches for a single character, another piece looks for blank spaces, and a third piece substitutes one letter for another. When you have all the pieces, just assemble these pieces to form a search-and-replace program. Later on, when you think of a new feature for your word processing software, you reassemble the pieces in a slightly different way. It's sensible, it's cost efficient, and it's much more fun.

The late 1980s saw several advances in software development, and by the early 1990s, many large programming projects were being written from prefab components. Java came along in 1995, so it was natural for the language's founders to create a library of reusable code. The library included about 250 programs, including code for dealing with disk files, code for creating windows, and code for passing information over the Internet. Since 1995, this library has grown to include more than 4,000 programs. This library is called the *Application Programming Interface (API)*.

Every Java program, even the simplest one, calls on code in the Java API. This Java API is both useful and formidable. It's useful because of all the things you can do with the API's programs. It's formidable because the API is extensive. No one memorizes all the features made available by the Java API. Programmers remember the features that they use often, and they look up the features that they need in a pinch. They look up these features in an online document called the *API Specification* (known affectionately to most Java programmers as the *API documentation*, or the *Javadocs*).

The API documentation (see `https://docs.oracle.com/en/java/javase/17/docs/api`) describes the thousands of features in the Java API. As a Java programmer, you consult this API documentation daily. You can bookmark the documentation at the Oracle website and revisit the site whenever you need to look up something, or you can save time by downloading your own copy of the API docs using the links found at `www.oracle.com/technetwork/java/javase/downloads/index.html`.

Your Java Programming Toolset

To write Java programs, you need the tools described previously in this chapter:

- **You need a Java compiler.** Refer to the section "Translating your code."
- **You need a JVM.** Refer to the section "Running code."
- **You need the Java API.** Refer to the section "Code you can use."
- **You need access to the Java API documentation.** Again, refer to the "Code you can use" section.

You also need some less exotic tools:

- **You need an editor to compose your Java programs.** Listing 1-1 contains part of a computer program. When you come right down to it, a computer program is a big bunch of text. So, to write a computer program, you need an *editor* — a tool for creating text documents.

  An editor is a lot like Microsoft Word, or like any other word processing program. The big difference is that an editor adds no formatting to your text — no bold, italic, or distinctions among fonts. Computer programs have no formatting whatsoever. They have nothing except plain old letters, numbers, and other familiar keyboard characters.

  REMEMBER

  When you edit a program, you may see bold text, italic text, and text in several colors. But your program contains none of this formatting. If you see stuff that looks like formatting, it's because the editor you're using does syntax highlighting. With *syntax highlighting,* an editor makes the text appear to be formatted in order to help you understand the structure of your program. Believe me, syntax highlighting is very helpful.

- **You need a way to issue commands.** You need a way to say things like "compile this program" and "run the JVM." Every computer provides ways of issuing commands. (You can double-click icons or type verbose commands in a Run dialog box.) But when you rely on your computer's own facilities, you're forced to jump from one window to another. You open one window to read Java documentation, another window to edit a Java program, and a third window to start up the Java compiler. The process can be *tedious.*

A tool for creating code

In the best of all possible worlds, you do all your program editing, documentation reading, and command issuing through one nice interface. This interface is called an *integrated development environment (IDE)*.

A typical IDE divides your screen's work area into several panes — one pane for editing programs, another pane for listing the names of programs, a third pane for issuing commands, and other panes to help you compose and test programs. You can arrange the panes for quick access. Better yet, if you change the information in one pane, the IDE automatically updates the information in all the other panes.

An IDE helps you move seamlessly from one part of the programming endeavor to another. With an IDE, you don't have to worry about the mechanics of editing, compiling, and running a JVM. Instead, you can worry about the logic of writing programs. (Wouldn't you know it? One way or another, you always have something to worry about!)

In the chapters that follow, I describe basic features of the IntelliJ IDEA IDE (known simply as "IntelliJ" by most Java professionals). IntelliJ has many bells and whistles, but you can ignore most of them and learn to repeat a few routine sequences of steps. After using IntelliJ a few times, your brain automatically performs the routine steps. From then on, you can stop worrying about IntelliJ and concentrate on Java programming.

As you read my paragraphs about IntelliJ, remember that Java and IntelliJ aren't wedded to one another. The programs in this book work with any IDE that can run Java. Instead of using IntelliJ, you can use Eclipse, NetBeans, BlueJ, or any other Java IDE. In fact, if you enjoy roughing it, you can write and run this book's programs without an IDE. You can use Notepad, TextEdit, or vi, along with your operating system's command prompt or Terminal. It's all up to you.

What’s already on your hard drive?

You may already have some of the tools you need for creating Java programs. But, on an older computer, your tools may be obsolete. Many of this book’s examples run on all versions of Java. But some examples don’t run on versions earlier than Java 8. Other examples run only on Java 11, Java 12, Java 13, or later.

The safest bet is to download tools afresh. To find detailed instructions on doing the downloads, see Chapter 2.

IN THIS CHAPTER

- Downloading and installing the IntelliJ IDEA development environment
- Installing Java
- Getting Java documentation

Chapter 2

Setting Up Your Computer

This chapter goes into much more detail than you normally need. If you're like most readers, you'll follow the steps in the "Let's Get Started" section. Then you'll check the table of contents for any other sections you want to read. After reading about a third of this chapter's material and skimming the remaining two-thirds, you'll have everything you need to begin learning about Java programming.

Of course, you may need to read more than just a third of the chapter.

Everyone's situation is unique. One person has an older computer. Another person has some conflicting software. Joe has a PC, and Jane has a Mac. Joe's PC runs Windows 10, and Elaine's runs Windows 8. Joe misreads one of my instructions and, as a result, nothing on his screen matches the steps that I describe. Seventy percent of this chapter describes the things you do in those rare situations in which you must diagnose a problem.

If you find yourself in a real jam, there's always an alternative. You can send an email to me at `BeginProg@allmycode.com`. You can also find me on Facebook at `/allmycode` or on Twitter at `@allmycode`. I'm happy to answer questions and help you figure out what's wrong.

So, by all means, skip anything in this chapter that you don't need to read. You won't break anything by following your instincts. And if you do break anything, there's always a way to fix it.

REMEMBER

The websites that I describe in this chapter are always changing. The software that you download from these sites changes, too. A specific instruction such as "Click the button in the upper right corner" may become obsolete (and even misleading) in no time at all. So in this chapter I provide long lists of steps, but I also describe the ideas behind the steps. When you visit a website, look for ways to get the software that I describe. If a site offers you several options, check the instructions in this chapter for hints on choosing the best one. At one point, you install and run the IntelliJ IDEA application. If the application window doesn't look quite like the window in this chapter's figures, scan your computer screen for whatever options I describe. If, after all that, you can't find what you're looking for, check this book's website (`http://beginprog.allmycode.com`) or send an email to me at `BeginProg@allmycode.com`.

Let's Get Started

To start writing Java programs, you need the software that I describe in Chapter 1: a Java compiler and a Java virtual machine (JVM, for short). You can also use a good integrated development environment (IDE) and some sample code.

Fortunately, all this software is available for free. To get it, just follow these steps:

1. **Visit** `www.jetbrains.com/idea/download` **and download the IntelliJ IDEA Community Edition.**

   IntelliJ IDEA comes in two different editions: Ultimate Edition or Community Edition. The Community Edition is free.

   When you click the web page's Download button, the kind of file you get depends on your computer's operating system:

   - *On Windows,* you choose either an executable installer file (with the `.exe` extension) or a Zip archive (with the `.zip` extension).

     If you want to avoid headaches, choose the executable installer (`.exe`) option.

   - *On a Mac,* you choose between the Intel and Apple Silicon downloads.

     The one you download depends on the kind of processor in your computer. If you're not sure which to download, go to your computer's own menu bar and choose Apple ➪ About This Mac. In the resulting dialog box, look for information about your computer's processor. If you see the word *Intel,* you want the website's Intel download. Otherwise, you want Apple Silicon.

  One way or another, you get a disk image file with the extension `.dmg`.

- *On Linux,* you get a GZip archive file with the extension `.gz`.

CROSS REFERENCE

The file that you download has a name like `ideaIC-2021.exe`, `ideaIC-2021.dmg`, or `ideaIC-2021.tar.gz`. If your computer doesn't display the file extension (such as `.exe` or `.dmg`), check the section entitled "Shining light on filename extensions," later in this chapter.

2. **Check your web browser for its list of downloaded files. In that list, find the most recent download (IntelliJ IDEA) and double-click that download's button or link.**

TIP

Most web browsers save files to a `Downloads` folder on your computer's hard drive, but your browser may be configured a bit differently. One way or another, it helps to make note of the folder containing the downloaded IntelliJ installation file.

What happens when you double-click depends on the kind of operating system you're running:

- *On Windows,* a dialog box offers to guide you through the installation.
- *On a Mac,* a Finder window tells you to drag an icon to your computer's Applications folder.
- *On Linux,* well, . . . (sorry!) . . . you're on your own.

  Linux has too many variations for me to cover all the possibilities here. Besides, if you're a Linux user, you're probably tech-savvy. You know what to do without reading these steps.

3. **Finish installing IntelliJ IDEA.**

If the installation presents you with options, don't think too hard about them. It's usually safe to accept the defaults.

Firing up IntelliJ IDEA

Imagine that you're expecting a delivery from a local confectioner. In the late afternoon, the mail carrier delivers a box containing your favorite chocolate candy. Naturally, you want to open the box as soon as it arrives.

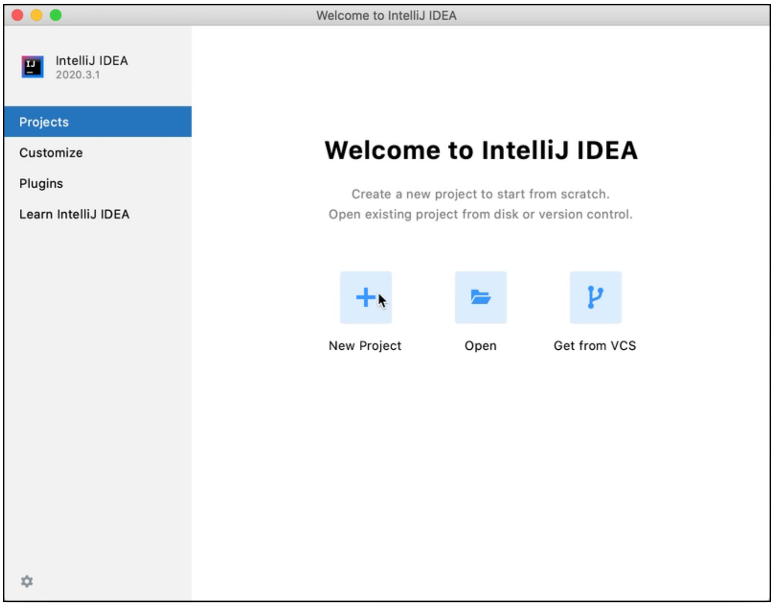
The same is true about this book's software. Here's how you open your newly downloaded IntelliJ IDEA box:

1. **Launch the IntelliJ IDEA application.**

   When you do, you see a flashy banner display. After a few seconds, the banner disappears and you see the Welcome to IntelliJ IDEA dialog box. The box is shown in Figure 2-1.

   TIP

   You can change the IntelliJ color theme by selecting Customize in the Welcome dialog box's side panel.



Welcome to IntelliJ IDEA

Create a new project to start from scratch.
Open existing project from disk or version control.

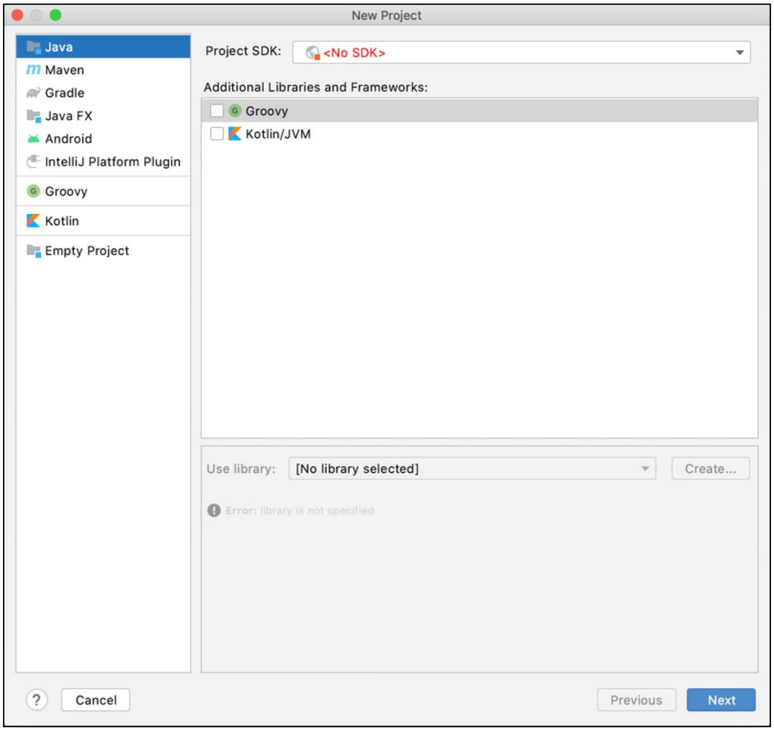
FIGURE 2-1: Willkommen, bienvenue, welcome!

2. **In the Welcome dialog box, choose New Project.**

   A Java application may consist of several files, including code files, image files, data files, installation instructions, and other stuff. With IntelliJ IDEA, you manage an application's files by combining them into a single *project*.

   In this book, a typical project contains only one file. Why bother "collecting" one file into a bigger thing called a "project"? The answer is, IntelliJ IDEA wants each application to be part of a project, just in case the application grows to include dozens or even hundreds of files.

   When you select New Project, a dialog box appears. To no one's surprise, the box's title is New Project. (See Figure 2-2.)



Project SDK: <No SDK>

Use library: [No library selected]

Error: library is not specified

FIGURE 2-2: Everything starts here.

3. **Make sure that the topmost entry (namely, Java) is selected in the dialog box's side panel. (Refer to Figure 2-2.)**

4. **At the top of the dialog box's main body, look for the drop-down list labeled Project SDK. (Refer again to Figure 2-2.)**

TECHNICAL STUFF

The letters SDK stand for *s*oftware *d*evelopment *k*it. During its long history, Java has come in many different shapes and sizes. Even now, the kind of Java you run depends on the kind of Java you need. This drop-down box asks you to select the kind of Java you'll be using in your new project.

CROSS REFERENCE

For more than you'd like to know about Java's many incarnations, see the later section "The Java smorgasbord."

What you do next depends on what you see in that drop-down list.

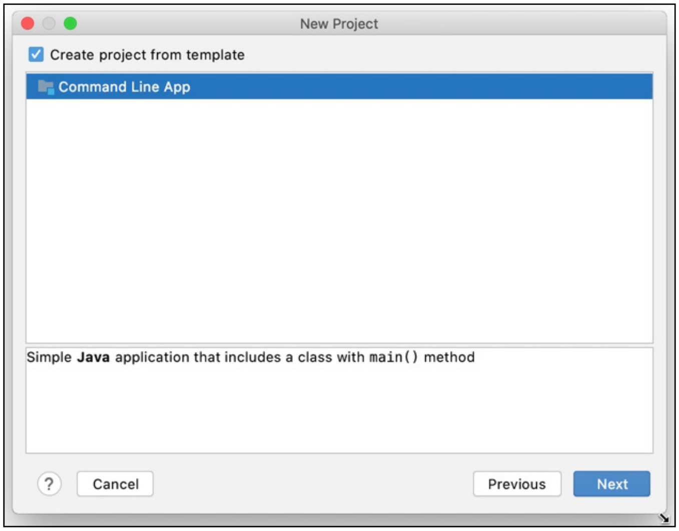
5. **If you see <No SDK> in the drop-down list, jump temporarily to the later section "Installing Java."**

   Likewise, if you see Version 16 or any version lower than 16 (including versions like 1.8.0_241) in the drop-down list, jump temporarily to the later section "Installing Java."

   If you see Version 17 or any version number higher than 17, leave the drop-down list as it is and click Next.

After clicking Next, you see a second New Project dialog box. You can proceed to the next step in this list of instructions.

6. **The second New Project dialog box displays the Create Project from Template check box. Put a check mark in the Create Project from Template check box. (See Figure 2-3.)**



☑ Create project from template

Command Line App

Simple **Java** application that includes a class with `main()` method

FIGURE 2-3: Care to use a project template?

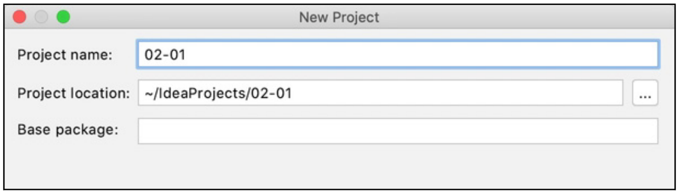
7. **In the same New Project dialog box, make sure that the Command Line App item is selected. (Refer again to Figure 2-3.)**

8. **At the bottom of this New Project dialog box, click Next.**

When you click Next, a third (and, thankfully, final) New Project dialog box asks for a project name, a project location, and a base package. (See Figure 2-4.)

For the project name, almost any sequence of characters will do. In Figure 2-4, I use the name 02-01. After all, this is Chapter 2, and this is the first (and only) project in Chapter 2.

For the project location, I recommend keeping the default.



Project name: 02-01

Project location: ~/IdeaProjects/02-01

Base package:

FIGURE 2-4: What, where, and how.

9. **Delete any text in the Base Package field. (Refer to 2-4.)**

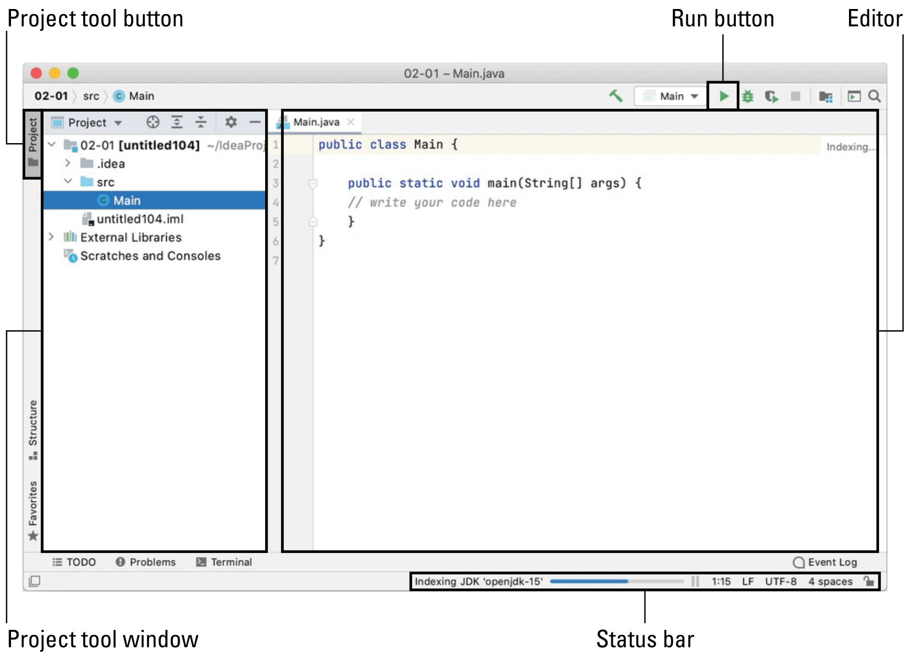
TECHNICAL STUFF

My Java colleagues will rip me to pieces for telling you to leave the Base Package field empty. A *package* is a group of one or more Java code files, and experienced Java professionals never write code without naming a package. The trouble is, having a named base package would make it slightly more difficult for you to run this book's sample programs. The quickest (and dirtiest) solution is to have you clear out the Base Package field. Don't tell anyone. It's our little secret.

10. **In this final New Project dialog box, click Finish.**

At last! IntelliJ's main window appears on your computer screen.

Figure 2-5 contains a screen shot of the main window. In the figure, I've labeled a few of the main window's parts.



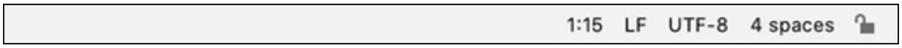
```
public class Main {

    public static void main(String[] args) {
    // write your code here
    }
}
```

FIGURE 2-5: IntelliJ IDEA's main window.

WARNING

It may take a while for IntelliJ to finish creating a new project. You may see only a big gray area in most of the main window. You may see some messages about indexing on the status bar. If so, be patient. Wait for the text on the status bar to stop changing. Figure 2-6 shows what you see when the status bar in Figure 2-5 stops changing.

FIGURE 2-6: Tranquility on the status bar.



With no turmoil on the status bar, you may still not see the Editor or the Project tool window.

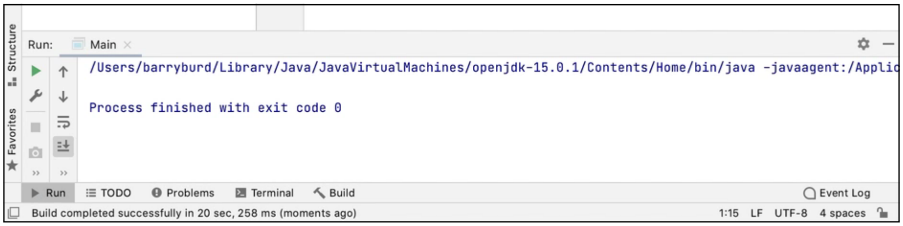
- If you don't see the Project tool window, you can coax it out of hiding: Just click the Project tool button along the window's leftmost edge.
- If you don't see the Editor, expand the Project tool window's tree and double-click the tree's `Main` branch. (Refer to Figure 2-5.)

Enough preliminaries. It's time to run some Java code.

11. **Click the Run button.**

    When you do, you should see some new messages on the IntelliJ status bar. After a number of seconds, you see a new tool window — the IntelliJ Run tool window — along the bottom of the screen. (See Figure 2-7.) Along with some technical gobbledygook, the Run tool window displays the words `Process finished with exit code 0`.

FIGURE 2-7: Not much to see here.



```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk-15.0.1/Contents/Home/bin/java -javaagent:/Applic

Process finished with exit code 0
```

Build completed successfully in 20 sec, 258 ms (moments ago)

That's it! You're done setting up the software. For answers to questions about any of these steps, see the later section "If You Need More Details . . . ."

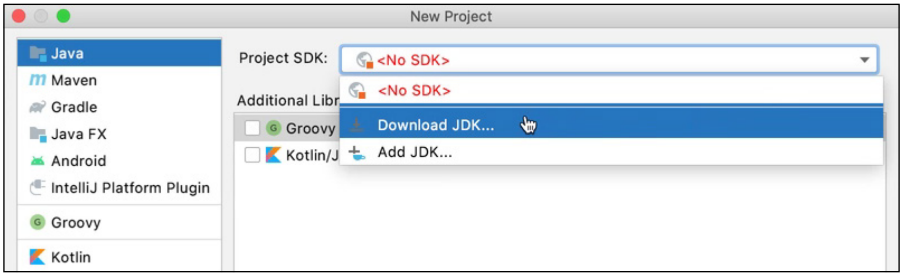
Installing Java

What? You say you were following the previous section's steps and then Step 5 in that section made you take a detour? Well, you've come to the right place!

When you reached Step 5, you were staring at the New Project dialog box. Maybe the text <No SDK> appeared in the dialog box's Project SDK drop-down list. (Refer to Figure 2-2.) Or maybe the drop-down list displayed a version number lower than 17.

In either case, here's what you do next:

1. **In the Project SDK drop-down list, select Download JDK. (See Figure 2-8.)**

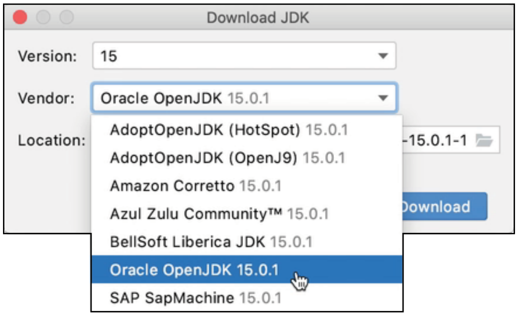


New Project

Project SDK: <No SDK>

FIGURE 2-8: IntelliJ offers to download Java for you.

When you select Download JDK, IntelliJ displays a new Download JDK dialog box. (See Figure 2-9.)



Download JDK

Version: 15

Vendor: Oracle OpenJDK 15.0.1

Location: -15.0.1-1

AdoptOpenJDK (HotSpot) 15.0.1
AdoptOpenJDK (OpenJ9) 15.0.1
Amazon Corretto 15.0.1
Azul Zulu Community™ 15.0.1
BellSoft Liberica JDK 15.0.1
Oracle OpenJDK 15.0.1
SAP SapMachine 15.0.1
Download

FIGURE 2-9: Pick your poison.

2. **In the Download JDK dialog box's Version list, select the highest number available.**

3. **For the Vendor drop-down list, I recommend the Oracle OpenJDK option.**

   I don't have a particularly good reason for recommending this option. Oracle owns the rights to Java, so using Oracle's product means getting Java "from the horse's mouth." And the prefix Open in OpenJDK means that this version of Java is free to use for noncommercial purposes.

   REMEMBER

   No matter which vendor option you pick, you'll be okay. This book's examples run on any newer version of Java.

   The third item in the Download JDK dialog box is a Location field.

4. **In the Location field, your safest move is to accept the default.**

   In other words, leave that field alone.

5. **When you're finished making selections, click the Download button.**

   As if by magic, IntelliJ downloads Java and installs it on your computer.

6. **In the New Project dialog box, click Next.**

   As a result, another dialog box with the same title (New Project) appears. With this second New Project dialog box staring you in the face, you can return to Step 6 in the previous section.

   It was fun guiding you through this section's steps. Please say hello to the "Firing up IntelliJ IDEA" section for me!

If You Need More Details . . .

I just put the finishing touches on a script for a new horror film. Here's an excerpt from the story:

> *The scene is an old country mansion on a dark, windy night. Victoria J. Venom sits alone in her well-appointed office. She's about to install some software on her laptop, but the procedure for installing it isn't straightforward. Victoria must follow many steps (twenty of them, to be precise), and some of the steps involve unfamiliar choices. The software vendor has made changes since the steps were written, so some of the instructions' screen shots don't match what she sees on her computer. Her laptop has some older software that her stepmother once installed. This old software presents options that differ from the ones in the installation instructions.*
>
> *[Cue the foreboding background music.] The instructions tell Victoria what to do but not why to do it. So one false move on her part and the whole installation goes awry. Early on, she ignores a warning message, and things go smoothly in the next few steps. But then, several steps later, Victoria sees nothing but error messages. Unfortunately, the messages give no guidance to help her fix the problems.*
>
> *[Cue the sounds of wolves howling in the distance.] Victoria tries to retrace her steps to correct any bad decisions she made along the way, but this only makes things worse. She tries to roll everything back to Step 1, but that doesn't work. The system remembers some of her bad choices, and any further tinkering gets her in deeper trouble.*

> *In desperation, Victoria picks up her laptop and throws it against the wall. With a scowl on her face and veins throbbing on her neck, she vows to seek revenge.*
>
> *[Cut to the scene with the unsuspecting software vendor.]*

I wonder. Does Wiley plan to start its own video entertainment streaming service? If so, will the service need new movie ideas?

Anyway, installing software can be a pain in the neck. I get plenty of emails from my books' readers, and the number-one question from them is, "What went wrong when I followed the installation instructions?"

The next several sections provide details on the previous section's steps and offer hints to fix any problems that arise.

Shining light on filename extensions

The filenames displayed in Windows File Explorer or macOS Finder can be misleading. You may browse one of your folders and see the name `ideaIC-2021`. The file's real name might be `ideaIC-2021.exe`, `ideaIC-2021.dmg`, `ideaIC-2021.`*`somethingElse`*, or plain old `ideaIC-2021`. Filename endings like `.exe` and `.dmg` are called *filename extensions*.

The ugly truth is that, by default, Windows and Macs hide many filename extensions. This awful feature tends to confuse programmers. So, if you don't want to be confused, change your computer's system-wide settings. Here's how you do it:

- **In Windows 10:** In the taskbar's Search box, type **File Explorer Options** and then press Enter. When the File Explorer Options dialog box appears, click the View tab. Look for the Hide Extensions for Known File Types option. Make sure that this check box isn't selected.
- **In Mac OS X:** On the menu bar for Finder, choose Finder ➪ Preferences. In the resulting dialog box, select the Advanced tab and look for the Show All Filename Extensions option. Make sure that this check box *is* selected.
- **In Linux:** Linux distributions tend not to hide filename extensions. So, if you use Linux, you probably don't have to worry about it. But I haven't checked all Linux distributions. If your IntelliJ download is named `ideaIC` instead of `ideaIC.tar.gz` or `ideaIC.`*`whatever`*, check the documentation specific to your Linux distribution.

Dealing with a Mac's security features

If you have a Mac, you might encounter some speed bumps during your IntelliJ installation:

- **The Safari browser may ask whether you want to allow downloads from** `www.jetbrains.com`**.**

  If so, click Allow.

- **You may see a troubling dialog box warning you that "IntelliJ IDEA is an app downloaded from the Internet" and asking, "Are you sure you want to open it?"**

  If so, your answer should be "Yes. Open it."

- **Your Mac might be set up to install only App Store software.**

  If so, you'll see a message saying, "IntelliJ IDEA CE can't be opened because it was not downloaded from the App Store," or something like that. Here's how you deal with that problem:

  1. *Choose Apple ➪ System Preferences ➪ Security & Privacy.*
  2. *In the Security & Privacy dialog box, select the General tab.*
  3. *Click the lock in the dialog box's lower-left corner and confirm the unlocking by typing your account password.*

     It's the same password you use whenever you log on to the computer.

     In the Security & Privacy dialog box, a group of radio buttons begins with the prompt Allow Apps Downloaded From.
  4. *Select the radio button labeled App Store and Identified Developers.*

     Hooray! The company that makes IntelliJ IDEA is an identified developer.
  5. *Close the Security & Privacy dialog box.*

Using IntelliJ IDEA with finesse

On a cold day one January, while I was rushing to the Milwaukee airport, my car turned itself off. I got out of the car, looked under the hood, and noticed some unattached cables. I had been taking an introductory auto mechanics course, so I didn't have to wait for a repair service to arrive. I reattached the cables and was immediately on my way.

What little I had learned in the auto mechanics course had paid off! Sure, I knew how to drive a car. But, on that day in Milwaukee, getting to the airport on time required knowing a tiny bit more than the absolute minimum.

The same idea is true about working with software. The more you understand, the better off you are. With that in mind, this section goes beyond the minimum knowledge you need for using IntelliJ IDEA.

Stopping and starting work with IntelliJ IDEA

The instructions in this chapter's earlier section "Firing up IntelliJ IDEA" leave you hanging. After the last step, you see the run of a Java program. The run is uneventful because the program that IntelliJ creates isn't meant to do anything. So, what happens after you've finished following these instructions? Here are some possibilities:

- **You leave IntelliJ IDEA open.**

  Without shutting down your computer, you drive to Las Vegas, get married, and go on a honeymoon in Niagara Falls. (Oh, oh! Did someone say "Niagara Falls"? Slowly I turn. . . .) When you return home, your computer is still on. IntelliJ's main window is still showing on the screen. The main window still contains the project in the "Firing up IntelliJ IDEA" section. If you want, you can continue working on that project.

  That's one way to proceed after following the steps in the "Firing up IntelliJ IDEA" section. Here's another:

- **With IntelliJ running and your project in the main window, you close the entire IntelliJ application.**

  Of course, the IntelliJ application stops running. The next time you launch IntelliJ, you see IntelliJ's main window. The project from the earlier section "Firing up IntelliJ IDEA" appears in that window.

  Here's yet another possibility:

- **With IntelliJ running and your project in the main window, you choose File ➪ Close Project.**

  As a result, IntelliJ displays a Welcome window, much like the one shown in Figure 2-1. In addition to the New Project, Open, and Get from VCS buttons, this new window displays a list of your most recent projects. If you click an item in that list, IntelliJ opens the corresponding project in a main window.

  And, finally:

- **With only a Welcome window showing, you close the IntelliJ application.**

  The next time you launch IntelliJ, you see its familiar Welcome window. You can select one of your recent projects or start from scratch by selecting New Project.

Projects by the barrelful

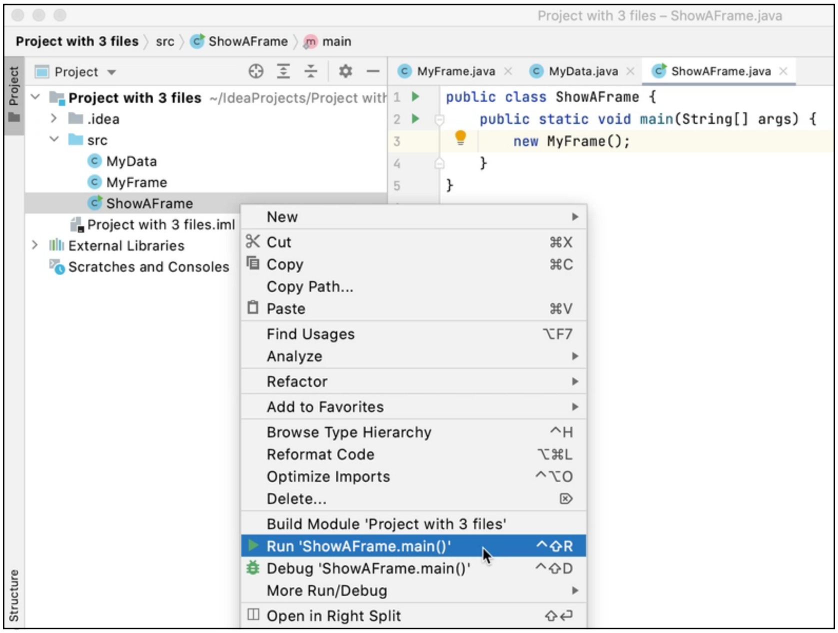
IntelliJ can display several projects at a time, each in its own main window. To make this happen, have at least one main window showing. Then, on the IntelliJ main menu bar, choose File➪New➪Project or File➪Open Recent. In either case, IntelliJ prompts you with a dialog box, asking "Where would you like to open the project?" If you select New Window, IntelliJ leaves all existing windows open and creates an additional window for your newly opened project.

Running code

A single project may contain several Java code files. When you're ready to test your code, you click the Run button, shown in Figure 2-5. That's usually okay. But once in a while, you get some unexpected results. IntelliJ starts running one of your project's Java files. How does IntelliJ decide which file to run?

The safest way around this issue is to avoid clicking the Run button. Figure 2-10 shows part of the main window for a project containing three Java code files. The files' names are `MyData`, `MyFrame`, and `ShowAFrame`.

To run the file named `ShowAFrame`, begin by right-clicking either the `ShowAFrame` branch in the Project tool window's tree or the `ShowAFrame` tab at the top of the Editor. In either case, IntelliJ displays a context menu. On the context menu, select `Run ShowAFrame` or `Run ShowAFrame.main()`.



```
public class ShowAFrame {
    public static void main(String[] args) {
        new MyFrame();
    }
}
```

FIGURE 2-10: A `src` folder with three Java files.

REMEMBER

While IntelliJ churns away in preparation for dealing with your code, you don't see a Run option on either of these context menus. Be patient and wait until there's no activity on the main window's status bar. (Refer to Figure 2-6.) Occasionally, you don't see a Run option even with a status bar like the one shown in Figure 2-6. In that case, the file you're trying to run may not be runnable. For more information, see Chapter 13.

Help! My course instructor doesn't want me to use IntelliJ IDEA!

Most professional Java developers use one of three integrated development environments: IntelliJ IDEA, Eclipse, or NetBeans. Other development environments — such as BlueJ, DrJava, Greenfoot, JCreator, JDeveloper, and jGRASP — are popular among educators. Some people don't use any software development environments. Instead, they type code in NotePad or TextEdit and run code by typing command-line instructions.

However you choose to develop Java programs, the examples in this book will run just fine. If your boss or your instructor wants you to use Eclipse instead of IntelliJ IDEA, that's no problem. You'll have to learn some different editing commands and some different ways to get Java programs to run. But, aside from learning to use Eclipse instead of IntelliJ, your Java coding experience will be exactly the same. All you need is a fairly recent version of Java. Java 17 or higher will work just fine.

Downloading Java without IntelliJ IDEA

IntelliJ IDEA's menus make installing Java a breeze. But IntelliJ and Java are two separate products. At some point, you may have to install Java without running IntelliJ IDEA. If so, visit `https://adoptopenjdk.net` to get the latest available version of the JDK.

Near the top of the AdoptOpenJDK page, you might see links and buttons for Java 8 and Java 11. Those versions of Java are okay, but I recommend a version of Java numbered 17 or higher to get the most from this book's content.

The AdoptOpenJDK page offers you a download of the software that matches your operating system (Windows, Macintosh, or whatever). Beyond that, the website may offer you some choices:

- **Windows users may choose one of two operating system architectures: x86 and x64.**

Oddly enough, *x86* means that your version of Windows uses 32-bit registers, and *x64* means it uses 64-bit registers. These days, most people run 64-bit operating systems. So, if you're in a hurry, choose x64.

For all the gory details, see the later section "A bit of news about bits."

- **Windows users may choose one of two file extensions:** `.msi` **or** `.zip`**.**

  For a fairly painless installation, choose `.msi`. With an `.msi` file, you click a few buttons and accept a few defaults. Installing from a `.zip` file takes more work and is more prone to error.

- **Mac users may choose between an Intel or Apple Silicon download.**

  When you face this choice, follow my advice from Step 1 in the earlier "Let's Get Started" section.

When you feel comfortable with any choices you've made, click the page's Download button and proceed with the installation. (Double-click whatever file you've downloaded and then follow the installation instructions.)

A bit of news about bits

When you get Java from AdoptOpenJDK or some other site, you may be confronted with a choice between x86 and x64 downloads. The Windows operating system comes in two sizes: 32-bit and 64-bit. In the same way, a computer that runs Windows has either a 32- or 64-bit processor. On top of all that, there are two kinds of Java Windows computers — 32-bit Java and 64-bit Java.

For historical reasons, 32-bit processors, operating systems, and applications are often marked with the label *x86*. Less surprisingly, the 64-bit processors, operating systems, and applications are often marked with the label *x64*.

Figure 2-11 shows what you can do with various processors and their operating systems.

According to Figure 2-11, you can run 32-bit Java on a 64-bit processor, and you can do this with either 32-bit Windows or 64-bit Windows. To help you decide where you live in Figure 2-11, type **about your PC** in the taskbar's search box. In the About window that appears, look for the words *System Type.* Alongside those words, you may see something like "64-bit operating system, x64-based processor." That settles it. You're in the bottommost cells of the left and middle columns. You can run 32- or 64-bit Java.

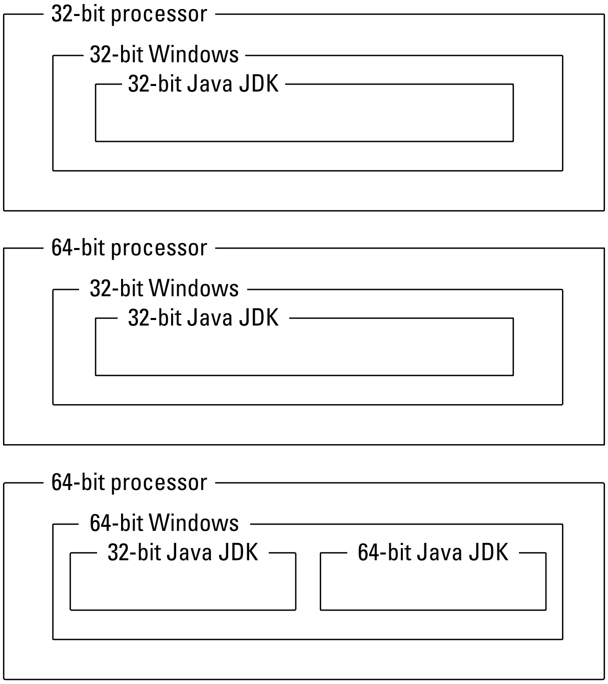


FIGURE 2-11: Mix and match.

Figure 2-11 says you can't run 64-bit Java on 32-bit Windows. If you try downloading 64-bit Java on a 32-bit Windows system, the download proceeds without a hitch. But when the download finishes and you try to run 64-bit Java, Windows complains vigorously. "This installation package is not supported by this product type," says Windows. In that case, revisit `https://adoptopenjdk.net` and choose the x86 option.

Another issue arises if you mix one kind of Java with another kind of IntelliJ IDEA. For example, if you try to run 32-bit Java on 64-bit IntelliJ IDEA, you might see a message of the following kind:

```
Error: LinkageError occurred while loading main class Main

java.lang.UnsupportedClassVersionError: Main has been compiled
by a more recent version of the Java Runtime, this version of
the Java Runtime only recognizes class file versions up to 55.0
```

If you do, look for a Download link or Configure link in the upper right corner of IntelliJ's main window. Click either of those links to get yourself out of hot water.

TIP

Greetings from January 2021! When I check the system requirements for IntelliJ IDEA, the website tells me that I need "64-bit versions of Microsoft Windows 10, 8." In spite of that, I can run IntelliJ IDEA on a 32-bit system. Go figure!

TECHNICAL STUFF

This section's advice about Windows and Java applies to other software as well. For example, you can't run a 64-bit accounting application on 32-bit Linux with a 32-bit processor.

The Java smorgasbord

I could write a whole book about the different makes and models of Java over the years. If I did, the book would be painful to read — like a dictionary, but with no interesting word-origin stories. Anyway, this section explains some of the terminology you might see as you travel through the Java ecosystem.

Medium Java, little Java, and gigantic Java

At some point, you may see mention of Java SE, Java ME, or Java EE. Here's the lowdown on these three kinds of "Java E":

- **Java Standard Edition (Java SE):** This is the only edition you should think about (for now, anyway). Java SE includes all the code you need in order to create general-purpose applications on a typical computer. Nowadays, when you hear the word *Java,* it almost always refers to Java SE.
- **Java Micro Edition (Java ME):** The Micro Edition contains code for programming special-purpose devices such as television sets, printers, and other gadgets. The book that you're reading contains no Java ME examples.
- **Java Enterprise Edition (Java EE):** In 1999, the stewards of Java released an edition that was tailored for the needs of big companies. The starring role in this edition was a framework called Enterprise JavaBeans — a way of managing data storage across connected computers. In 2017, Oracle walked away from Java EE, handing it over to the Eclipse Foundation, which renamed it Jakarta EE.

The rest of this book deals exclusively with Java Standard Edition.

Java for developers and Java for consumers

What comes to mind when you think about a chair? If you're a typical consumer, you think about a horizontal surface — a place to put your bottom. You may also think about a supporting structure for the chair (such as four legs) and some extra parts for resting your arms and your back.

If you're a carpenter, you may think differently about chairs. You think about wood, nails, hammers, screwdrivers, glue, the cost of materials, the marketability of your design, and other things.

You don't need nails or hammers to sit down on a chair. Likewise, you don't need a programmer's software tools to run an existing Java program. To run a Java program that someone else created, the only software you need is a *Java runtime environment (JRE).* But to create new programs, you need more tools. This big bunch of tools is known as a *Java development kit (JDK).*

This book is about creating Java programs. So, if a website makes you choose between a JRE and a JDK, download the JDK.

TECHNICAL STUFF

Other languages have their own development kits. In general, any one of these is called a *software development kit (SDK).*

Java evolves over time

If you want to drive yourself crazy, try making sense of Java's version numbering. It all started in 1996 with JDK 1.0. Next came JDK 1.1. In 1998, some marketing people decided on Java 2 SE 1.2. (The additional number 2 was quite confusing, but that's beside the point.) Things marched along until Java 1.5 was renamed Java 5.0 (but only for external numbering purposes). Java 6 had to do without any *.0* part.

The road from Java 1.0 to Java 9 took 21 years. But, in 2017, Oracle changed to a 6-month cycle for new versions of Java. So, in March 2018, Oracle released Java 10. And, in September 2018, Oracle released Java 11.

Java 11 was declared to be a *long-term support (LTS)* release. Oracle promised assistance with Java 11 for several years after the software's 2018 debut. When you see version numbers like Java 11.0.10, it's because Oracle was posting Java 11 updates as long as three years after the introduction of Java 11.

Other versions, such as Java 10 and Java 12, came with no long-term support. For example, Oracle released Java 12 in March 2019, Java 12.0.1 a month later, and Java 12.0.2 three months later. But in September 2019, Oracle released Java 13 and bid farewell to Java 12. If you wanted help with Java 12, Oracle said, "We're done with Java 12. Go ask somebody else."

Oracle's plan is to create long-term support releases every three years. After Java 11 in September 2018, the next long-term Java release is Java 17 in September 2021. After that, you have Java 23 in September 2024.

DON'T LOOK BACK!

In my travels as a *For Dummies* author, readers sometimes ask me if they should avoid versions of Java beyond the ones I mention in my books. This book refers to Java 17, but when you read the book, IntelliJ IDEA offers to install Java 19. What to do?

In general, the higher the version number, the better your Java experience. The people who manage Java's versions maintain a strict policy of *backward compatibility*, so the code that you and I write doesn't break over time. An app that ran correctly with Java 1.0 is very likely to run correctly with Java 17, and the examples you find in this book are likely to run under Java 18, 19, 20, and beyond.

Backward compatibility is a wonderful thing. Professional programmers work with programs that are thousands (and even millions) of lines long. With backward compatibility, programmers avoid having to fish through old programs to fix bugs caused by new releases.

TECHNICAL STUFF

Over the years, Java's numbering system has taken all kinds of twists and turns. The name Java 1.8.0_241 means something to some people, but it's of no concern to you as a novice programmer.

Everyone gets into the act

Oracle has two versions of the Java JDK. One of them is called *Oracle JDK*; the other is called *Oracle OpenJDK*. The difference between the two is the licensing agreement. Whereas Oracle OpenJDK is free and open-source, Oracle JDK is mainly for commercial use.

In addition, many other companies have their own versions of Java. For example, Amazon publishes an open-source JDK named *Corretto*. The Eclipse Foundation has a Java virtual machine called OpenJ9.

The first time you run IntelliJ IDEA, you may be prompted to choose between Oracle OpenJDK, Amazon Corretto, Azul Zulu, and other flavors of Java. Should you stress for more than a millisecond about this choice? No, you shouldn't. The differences among these products are of concern to big companies running massive applications, but they don't matter at all for the running of this book's examples. Just install a copy of Java and then sit back and enjoy the ride.

Juggling JDKs

Java's versions aren't like indoor cats — they can coexist on the same computer without fighting or hissing at one another. If you have more than one version of Java on your computer, you're okay. You can install Java 8, 11, and 17. You can install Oracle OpenJDK alongside Amazon Corretto. On Windows or Linux, you can mix 32- and 64-bit versions of Java.

The first time you create an IntelliJ project, the New Project dialog box looks for any JDKs that you have on your computer. If you haven't already installed a JDK, you can select the dialog box's Download JDK option. (That option is in the dialog box's Project SDK drop-down list. For more info, refer to the earlier "Installing Java" section.)

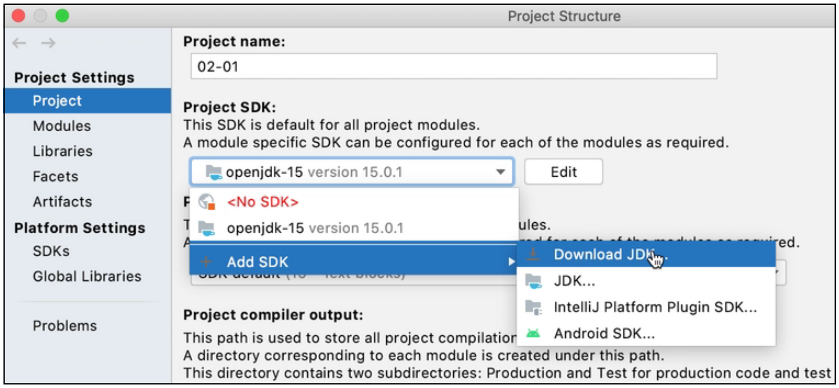
Later, when you start another project from scratch, the New Project dialog box assumes that you want to use the same JDK you used when you created the previous project. In the dialog box's Project SDK drop-down list, you'll find all the JDKs that you've already installed. If you want to use a JDK that you haven't already installed, select Download JDK in the drop-down list.

REMEMBER

In the Project SDK drop-down list, the default is always whatever JDK you chose the last time you created a new project.

Imagine this. You've been struggling for days on a project for work or a course assignment. At the last minute, someone tells you to use a brand-new feature of Java — a feature that's available only in the very newest JDK. How do you change an ongoing project's JDK?

With the project showing in the main IntelliJ window, go to the IntelliJ main menu bar and choose File➪Project Structure. When the Project Structure dialog box appears, look in the left side panel for an item named Project. (It's the first item in the Project Settings section. See Figure 2-12.)



Project Structure

Project name: 02-01

Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.

openjdk-15 version 15.0.1

Project compiler output:
This path is used to store all project compilation
A directory corresponding to each module is created under this path.
This directory contains two subdirectories: Production and Test for production code and test

FIGURE 2-12: Finding your way around the Project Structure dialog box.

After selecting Project, look in the dialog box's main body for a Project SDK drop-down list. In the drop-down list, choose Add SDK ➪ Download JDK, or select a JDK that's already in the list.

Getting the documentation

What follows is a scene from Barry Burd's childhood:

Mom: "What did you do in school today?"

Barry: "We had a substitute who didn't know what to do all day."

Mom: "So what happened?"

Barry: "We just watched a boring movie. I didn't understand any of the big words in the movie."

Mom: "What kinds of big words?"

Barry: "Words like *acnestis.* What does *acnestis* mean?"

Mom: "Look it up!"

Barry: "Awwww, Mom! *That's too much work. I don't want to look it up!*"

In Chapter 1, I refer to a website where you can find Java's API documentation. That website is a life-saver. But sometimes, finding web pages interrupts your workflow. What if you're like young Barry Burd and you don't like looking things up?

IntelliJ IDEA has an *external documentation* feature. Using this feature, you can look things up quickly and easily.

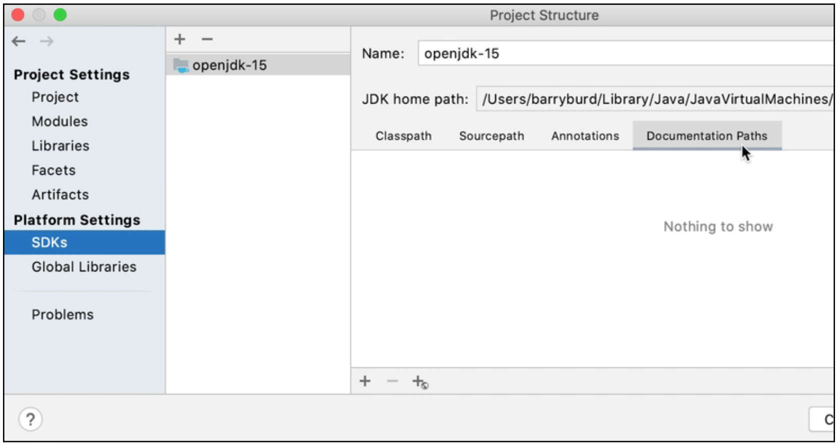
The one-time setup

Before you can use IntelliJ's external documentation feature, you have to tell IntelliJ where the external docs live.

1. **On the IntelliJ main menu bar, choose File ➪ Project Structure.**

   A Project Structure dialog box appears. (See Figure 2-13.)

2. **In the dialog box's left panel, select SDKs. (Refer to Figure 2-13.)**
3. **In the dialog box's main body, select the Documentation Paths tab. (You guessed it! Refer to Figure 2-13.)**



Name: openjdk-15

JDK home path: /Users/barryburd/Library/Java/JavaVirtualMachines/

FIGURE 2-13: More Project Structure stuff.

4. **In the area beneath the tabs, look for an icon with a tiny globe next to a plus sign.**

   In Figure 2-14, I enlarge the icon so that you can find it more easily. In this icon, the plus sign means “Add something,” and the globe means “The thing you’re adding is somewhere else in the world, not on your own computer’s hard drive.”

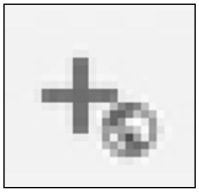
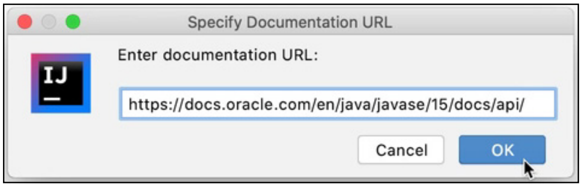


FIGURE 2-14: A hard-to-find icon.

When you click the icon, IntelliJ shows you a small dialog box with the title Specify Documentation URL. (See Figure 2-15.) If you’re lucky, a URL is already in the dialog box’s text field. The URL is something like this:

```
https://docs.oracle.com/en/java/javase/17/docs/api
```



Enter documentation URL:

https://docs.oracle.com/en/java/javase/15/docs/api/

FIGURE 2-15: The URL of Java’s API documentation.

5. **Click OK to accept the URL.**

TIP

If the small dialog box's text field is empty, you can find a URL by searching the web for `Java 17 API docs`, `Java 18 API docs`, or `Java` *`some-number`* `API docs`. In place of *`some-number`*, use whatever number you chose in Step 2 of the earlier "Installing Java" section. For example, if you chose OpenJDK 19.0.2, search for *Java 19 API docs*.

When you dismiss the small dialog box, you're tossed back to the big Project Structure dialog box.

6. **Click OK to dismiss the Project Structure dialog box.**

You're done setting up the IntelliJ external-docs feature. Good work!

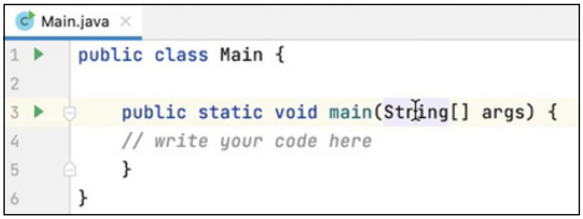
Looking stuff up

After completing the previous section's one-time setup, try these steps:

1. **Open an existing project or create a brand-new project.**

For complete instructions, refer to the earlier sections "Firing up IntelliJ IDEA" and "Stopping and starting work with IntelliJ IDEA."

The code in the Editor contains the word `String`. (See Figure 2-16.) This word has an official meaning in Java programs. (To learn more about words with official meanings, see Chapter 4.)



```
public class Main {

    public static void main(String[] args) {
        // write your code here
    }
}
```

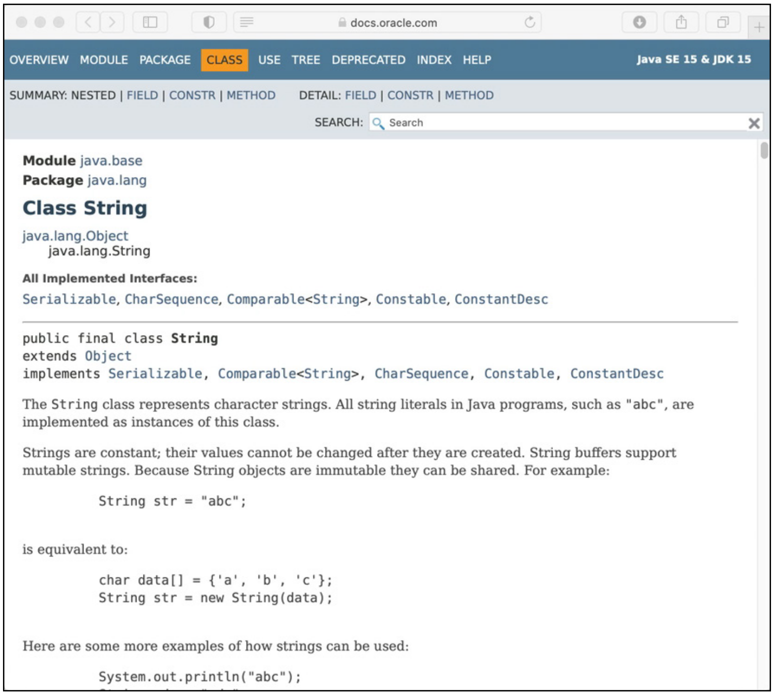
FIGURE 2-16 The cursor, on the word `String`.

2. **In the Editor, click the mouse anywhere on the word `String`. (Refer to Figure 2-16.)**

3. **On the IntelliJ main menu bar, choose View ➪ External Documentation.**

When you do, your web browser displays a brand-new page. For a preview of that page, see Figure 2-17.

A Java documentation page contains loads of useful information, but reading the documentation isn't always easy. For a lesson in making sense of Java's API doc pages, see my helpful little explainer on such documentation at `www.dummies.com/programming/java/making-sense-of-javas-api-documentation/`.


docs.oracle.com
OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java SE 15 & JDK 15
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
SEARCH:
Search

Module java.base
Package java.lang

Class String

java.lang.Object
java.lang.String

All Implemented Interfaces:

`Serializable`, `CharSequence`, `Comparable<String>`, `Constable`, `ConstantDesc`

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
     String str = "abc";
```

is equivalent to:

```
     char data[] = {'a', 'b', 'c'};
     String str = new String(data);
```

Here are some more examples of how strings can be used:

```
     System.out.println("abc");
```

FIGURE 2-17 The topmost part of a Java documentation page.

What's Next?

If you're reading this paragraph, you've probably finished installing IntelliJ IDEA and Java on your computer. In Chapter 3, you start reaping the benefits of your software installation efforts. Don't wait! Turn the page right now!

IN THIS CHAPTER

- » Running a program
- » Editing your own Java code
- » A grand tour of the IntelliJ window

Chapter 3 Running Programs

If you're a programming newbie, for you, running a program probably means clicking a mouse. You want to run Microsoft Word, so you double-click the Microsoft Word icon. That's all there is to it.

When you create your own programs, the situation is a bit different. With a new program, the programmer (or someone from the programmer's company) creates the program's icon. Before that process, a perfectly good program may not even have an icon. So, what do you do with a brand-new Java program? How do you get the program to run? This chapter tells you what you need to know.

Running a Canned Java Program

The best way to get to know Java is to do Java. When you're doing Java, you're writing, testing, and running your own Java programs. This section prepares you by describing how to run and test a program. Rather than write your own program, you run a program that I've already written for you.

Getting the code

To get a copy of all the programs that I've written for you, follow these steps:

1. **Visit this book's website:** `http://beginprog.allmycode.com`**.**

   I'm a world leader in the effort to create plain-looking, no-frills websites. You may even call my site "ugly." If so, that's fine with me.

   The website's front page has a link labeled Download the Code. That link refers to a file named `BeginProgJavaDummies6.zip`.

2. **Click the page's Download the Code link and save the** `BeginProgJavaDummies6.zip` **file on your computer's hard drive.**

3. **In a File Explorer or Finder window, navigate to the folder containing the downloaded** `BeginProgJavaDummies6.zip` **file.**

   Most web browsers save files to the `Downloads` folder on the computer's hard drive, but your browser may be configured a bit differently.

   The `BeginProgJavaDummies6.zip` file is a *compressed archive file*. When you uncompress this file, a new folder, full of smaller files, is created. Your computer's web browser may have already uncompressed the archive file automatically.

4. **If your web browser hasn't already uncompressed the** `BeginProgJavaDummies6.zip` **download, uncompress that file using your computer's File Explorer or Finder.**

   For details, see the nearby "Compressed archive files" sidebar.

5. **Make note of the place on your hard drive where you can find the uncompressed download.**

   As you work your way through this book's examples, you'll return to this place again and again.

Adding the code to IntelliJ IDEA

This chapter's first program calculates your monthly payments on a home mortgage loan. The mortgage-calculating program doesn't open its own window. Instead, the program runs in IntelliJ IDEA's Run tool window. The Run tool window normally appears in the lower portion of the IntelliJ IDEA application. (See Figure 3-1.) A program that operates completely in the Run tool window is called a *command line application* (or *command line program*).

COMPRESSED ARCHIVE FILES

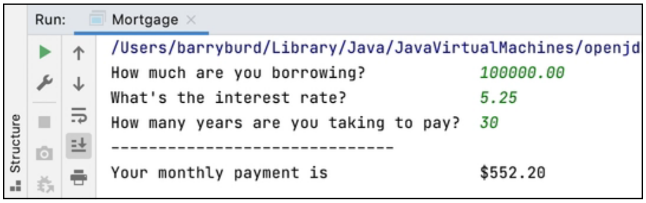
When you visit `http://beginprog.allmycode.com` and you download this book's Java examples, you download a file named `BeginProgJavaDummies6.zip`. A `.zip` file is a single file that encodes a bunch of smaller files and folders. For example, my `BeginProgJavaDummies6.zip` file encodes folders named `Chapter 03`, `Chapter 04`, and so on. The `Chapter 06` folder contains some subfolders, which in turn contain files. (The subfolder folder named `06–02` contains the code in Listing 6-2 — the second listing in Chapter 6.)

A `.zip` file is an example of a compressed archive file. Other examples of compressed archives are `.tar.gz` files, `.rar` files, and `.7x` files. Uncompressing a file means extracting the original files stored inside the big archive file. (For a `.zip` file, another word for *uncompressing* is *unzipping*.) Uncompressing normally re-creates the folder structure encoded in the archive file. So, after uncompressing my `BeginProgJavaDummies6.zip` file, your hard drive has a folder named `BeginProgJavaDummies6` with subfolders named `Chapter 03`, `Chapter 04`, and so on. Each chapter subfolder contains subfolders of its own.

When you download `BeginProgJavaDummies6.zip`, your web browser may uncompress the file automatically for you. If the icon next to `BeginProgJavaDummies6` in File Explorer or Finder looks like a typical folder's icon, then your browser has done the uncompressing. If not, you have to tell your computer to uncompress the `.zip` file.

Don't get me started on listing the dozens of ways to uncompress a `.zip` file! Some ways work on certain systems; other ways work on some other systems. If you're not sure how to uncompress my `.zip` file, try one of these tricks:

- **On Windows:** In File Explorer, navigate to the folder containing the downloaded `.zip` file. Right-click the `.zip` file. In the resulting context menu, select Extract All. In the resulting dialog box, put a check mark in the Show Extracted Files When Complete check box, and then press Extract. When all is said and done, Windows displays the contents of your new uncompressed `BeginProgJavaDummies6` folder.
- **On a Mac:** In a Finder window, navigate to the folder containing the downloaded `.zip` file. When you double-click this `.zip` file, your Mac creates a brand-new folder with the name `BeginProgJavaDummies6`. (The icon for this item looks like a typical folder's icon.) This new folder contains the uncompressed versions of the files that were encoded in the `.zip` file.



```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjd
How much are you borrowing?           100000.00
What's the interest rate?             5.25
How many years are you taking to pay? 30
------------------------------
Your monthly payment is               $552.20
```

FIGURE 3-1: A run of this chapter's mortgage program.

As you run the mortgage program, you see two things in the Run tool window:

- **Messages and results that the mortgage program sends to you:** Messages include things like `How much are you borrowing?`, and results include lines like `Your monthly payment is $552.20`.
- **Responses that you give to the mortgage program while it runs:** If you type **100000.00** in response to the program's question about how much you're borrowing, you see that number echoed in the Run tool window.

TIP

You don't see the Run tool window until you start the run of a Java program. For more information about the Run tool window (and about other IntelliJ IDEA features), see the section entitled "What's All That Stuff in the IntelliJ IDEA Window?" later in this chapter.

CROSS REFERENCE

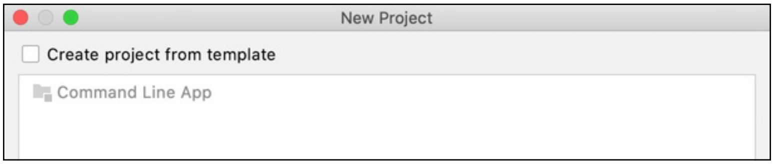
A *graphical user interface* (GUI) program is one that displays nice-looking windows with buttons, text fields, and other such items. Chapter 20 shows you how to start writing GUI programs. For even more GUI program examples, visit this book's website (`http://beginprog.allmycode.com`).

Here's how you prepare to run the mortgage program:

1. **Use IntelliJ IDEA to create a new Java project.**

   Follow the steps in Chapter 2, but make these two tiny changes:

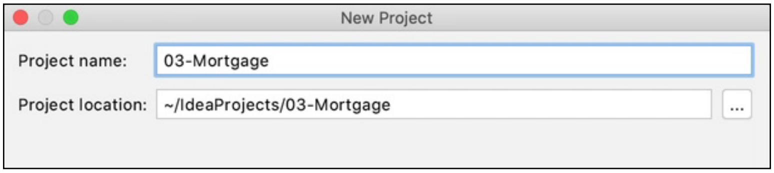
   - When IntelliJ offers to create the project from a template, tell IntelliJ to go jump in the lake. (See Figure 3-2.)
   - When IntelliJ asks for a project name, give it the name `03-Mortgage`. (See Figure 3-3.)



□ Create project from template

Command Line App

FIGURE 3-2: No template and no French fries with my order, please.


New Project

Project name: 03-Mortgage

Project location: ~/IdeaProjects/03-Mortgage

FIGURE 3-3: The last of three New Project dialog boxes.

In truth, the project name that you select can be almost any sequence of characters. It can even be the default name that IntelliJ offers (the name `untitled`). But, if you use the name `03–Mortgage`, you won't have to think much when you reach some of this section's later steps.

2. **Expand the tree in IntelliJ's Project tool window until you see a branch named `src`. (See Figure 3-4.)**

CROSS REFERENCE

The name `src` is short for *source*, so your Java source code lives on this branch. To read more about source code, refer to Chapter 1.

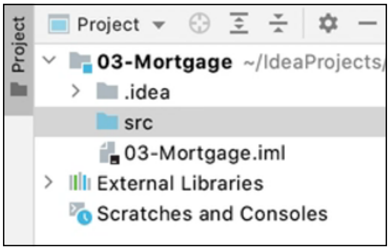


FIGURE 3-4: Finding the `src` folder.

3. **If the `src` branch contains an item named `Main`, delete that item.**

If you unchecked Create Project from Template in Step 1, you shouldn't see an item named `Main`. But if you expand the `src` branch and find a `Main` item, right-click it. On the resulting context menu, choose Delete. After removing all check marks in the confirmation dialog box, click OK.

You're ready to add this book's mortgage example to the newly created IntelliJ project.

4. **In your computer's File Explorer or Finder, navigate to the folder where you uncompressed the `BeginProgJavaDummies6` download.**

Refer to this chapter's earlier section "Getting the code."

Next, you're going to drill down inside the uncompressed `BeginProgJavaDummies6` folder.

5. **In File Explorer or Finder, double-click the `BeginProgJavaDummies6` folder.**

As a result, you see subfolders named `Chapter 03`, `Chapter 04`, `Chapter 05`, and so on.

6. **Double-click the `Chapter 03` folder.**

   As a result, you see a subfolder named `03-Mortgage`.

7. **Double-click the `03-Mortgage` folder.**

   Lo and behold! You see a file named `Mortgage.java`.

8. **Right-click the `Mortgage.java` file, and then choose Copy from the menu that appears.**

9. **Right-click the `src` branch of IntelliJ's Project tool window, and then choose Paste from the resulting context menu.**

   To make absolutely sure that you know what you're doing, IntelliJ displays a confirmation dialog box containing the filename (`Mortgage.java`) and the destination folder (a name ending in `src`).

10. **Click OK to accept the dialog box's suggestions.**

    Good work! It's time to run the `Mortgage.java` program.

Running the code

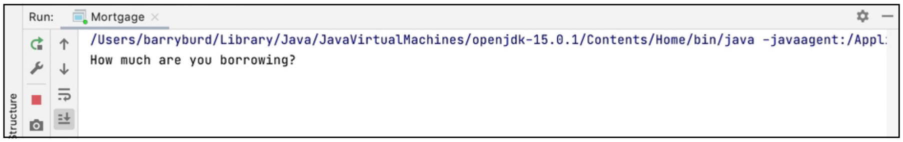
If you had a $100,000 mortgage, how much would you pay each month? To find out, follow these steps:

1. **Follow the steps in the earlier sections entitled "Getting the code" and "Adding the code to IntelliJ IDEA."**

2. **Right-click either the `Mortgage` branch in the Project tool window or the Mortgage tab at the top of the editor.**

3. **On the resulting context menu, choose `Run 'Mortgage.main()'`.**

   After a brief delay, IntelliJ displays its Run tool window. The words `How much are you borrowing?` appear in the window. (See Figure 3-5.)



```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk-15.0.1/Contents/Home/bin/java -javaagent:/Appl:
How much are you borrowing?
```

FIGURE 3-5: Woo-hoo! You're running some Java code!

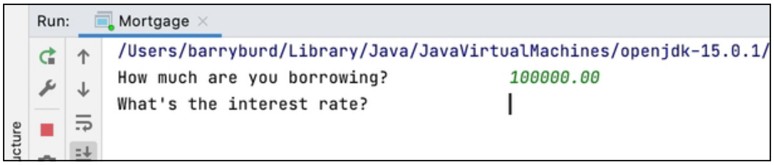
4. **Click anywhere inside the Run tool window, type a number, like** 100000.00**, and then press Enter. (See Figure 3-6.)**

WARNING

When you type a number in Step 4, don't include your country's currency symbol and don't group the digits. (US residents: Don't type a dollar sign and don't use any commas.) Amounts like $100000.00 and 1,000,000.00 cause this mortgage program to crash. You see a `NumberFormatException` message in the Run tool window.

Grouping separators vary from one country to another. The partial run shown in Figure 3-6 is for a computer configured in the United States where *100000.00* (with a dot) means "one hundred thousand." But the run might look different on a computer that's configured in what I call a "comma country" — a country where *100000,00* (with a comma) means "one hundred thousand." If you live in a comma country and you type 100000.00 exactly as it's shown in Figure 3-6, you probably get an error message (an `InputMismatchException`). If so, rerun the program using your country's number format. When you do, you should be okay.

After you press Enter, the Java program displays another message (`What's the interest rate?`) in the Run tool window. (Again, refer to Figure 3-6.)

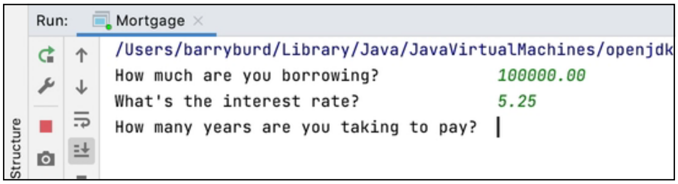


```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk-15.0.1/
How much are you borrowing?          100000.00
What's the interest rate?            |
```

FIGURE 3-6: That's a lot of money!

5. **In response to the interest rate question, type a number, like** 5.25**, and press Enter.**

   After you press Enter, the Java program displays another message (`How many years ... ?`) in Console view. (See Figure 3-7.)



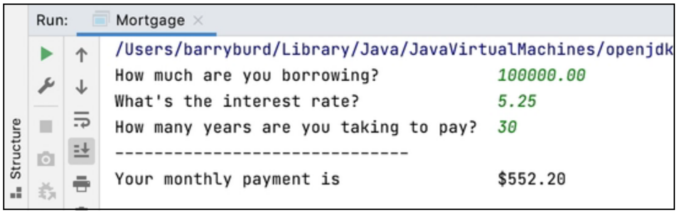
```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk
How much are you borrowing?          100000.00
What's the interest rate?            5.25
How many years are you taking to pay? |
```

FIGURE 3-7: Five-and-a-quarter percent interest.

6. **Type a number, like 30, and press Enter.**

   In response to the numbers that you type, the Java program displays a monthly payment amount. (See Figure 3-8.)

   Disclaimer: Your local mortgage company charges fees of all kinds. To get a mortgage in real life, you pay more than the amount that my Java program calculates. (A lot more.)



```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk
How much are you borrowing?            100000.00
What's the interest rate?              5.25
How many years are you taking to pay?  30
------------------------------
Your monthly payment is                $552.20
```

FIGURE 3-8: Your monthly payment.

WARNING

When you type a number in Step 6, don't include a decimal point. Numbers like 30.0 cause this mortgage program to crash. You see a `NumberFormatException` message in Console view.

TIP

Occasionally, you decide in the middle of a program's run that you've made a mistake of some kind. You want to stop the program's run dead in its tracks. Simply click the Stop button — little red square near the upper right corner of the main IntelliJ window. (See Figure 3-9.)

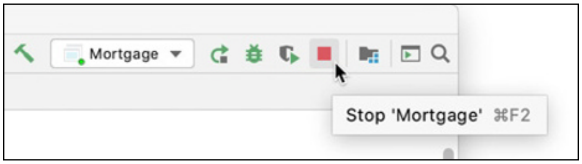


FIGURE 3-9: How to prematurely terminate a program's run.

If you follow this section's instructions and you don't get the results I describe, you can try three things. I list them in order, from best to worst:

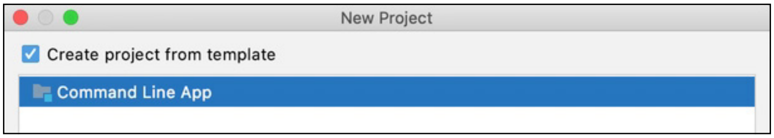
- Check all the steps to make sure that you did everything correctly.
- Send an email to me at `BeginProg@allmycode.com`, post to my Facebook wall (`/allmycode`), or tweet to the Burd (`@allmycode`). If you describe what happened, I can probably figure out what went wrong and tell you how to correct the problem.
- Panic.

Some Programs Don't Come in Cans

The previous section is about running someone else's Java code (code that you download from this book's website — `http://beginprog.allmycode.com`). But eventually, you'll write code on your own. This section shows you how to create code with IntelliJ IDEA. Follow these steps:

1. **Use IntelliJ IDEA to create a new Java project.**

   Follow the steps in Chapter 2. As you march through these steps, put a check mark in the Create Project from Template check box and make sure that the Command Line App item is selected. (See Figure 3-10.)



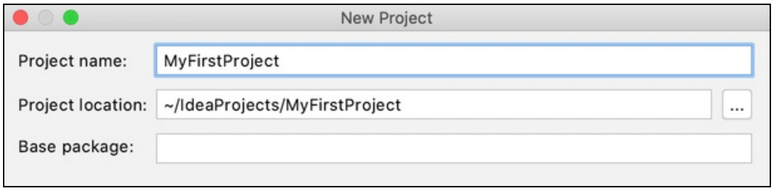
New Project

☑ Create project from template

Command Line App

FIGURE 3-10: Template, please!

In Figure 3-11, I create the name `MyFirstProject`.



New Project

Project name: MyFirstProject

Project location: ~/IdeaProjects/MyFirstProject

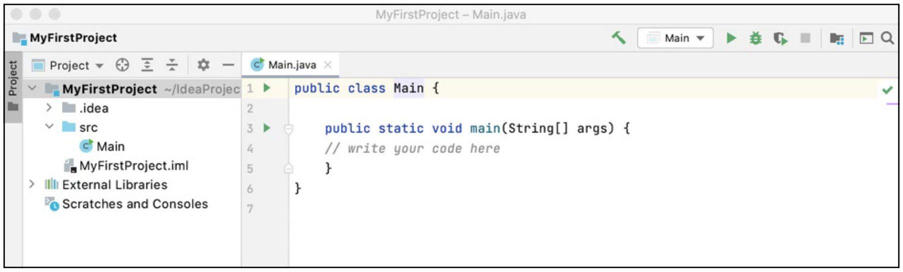
Base package:

FIGURE 3-11: What better name for this project?

REMEMBER

The project name that you select can be almost any sequence of characters. Select descriptive names so that you can find projects easily. Cryptic names, such as `Stuff01` and `Project123`, are useless when you're looking for your work from three weeks ago — work that you only vaguely remember.

When you've finished creating the new project, IntelliJ shows you a window like the one in Figure 3-12. Your new project contains a file named `Main.java`. For your convenience, the `Main.java` file already has some code in it. IntelliJ's editor displays the Java code.



```
public class Main {

    public static void main(String[] args) {
    // write your code here
    }
}
```

FIGURE 3-12: A shiny new project.

2. **Replace an existing line of code in your new Java program.**

   Type a line of code in IntelliJ's editor. Replace the line

   ```
   // write your code here
   ```

   with the line

   ```
   System.out.println("Chocolate, royalties, sleep");
   ```

   See Listing 3-1.

   Here are some tips:

   - Spell each word exactly the way I spell it in Listing 3-1.
   - Capitalize each word exactly the way I do in Listing 3-1.
   - Include all the punctuation symbols — the dots, the quotation marks, the semicolon — everything.
   - Distinguish between the lowercase letter `l` and the digit `1`. The word `println` tells the computer to print a whole line. Each character in the word `println` is a lowercase letter. The word contains no digits.

REMEMBER

LISTING 3-1: A Program to Display the Things I Like

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Chocolate, royalties, sleep");
    }
}
```

WARNING

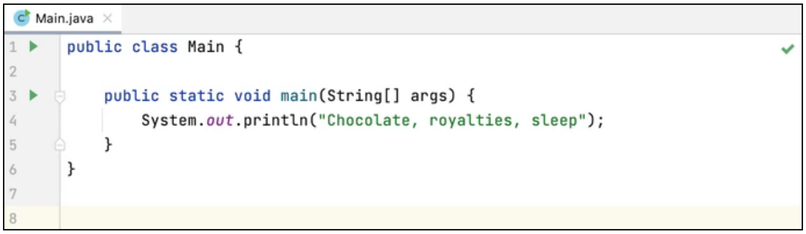
Java is *case-sensitive*, which means that `system.out.printLn` isn't the same as `System.out.println`. If yOu tyPe `system.out.printLn`, your progrAm won't worK. Be sUre to cAPItalize your codE eXactLy as it is shown in LiSTIng 3-1.

WARNING

If you copy-and-paste code from an ebook, make sure that the quotation marks in the code are straight quotation marks (`""`), not curly quotation marks (“ ”). In a Java program, straight quotation marks are good; curly quotation marks are troublesome.

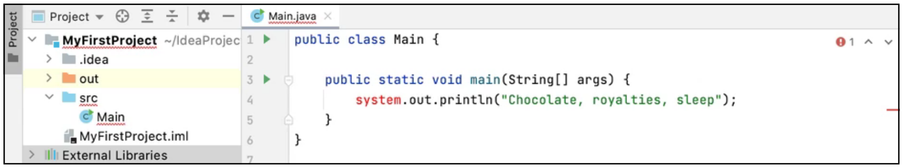
If you typed everything correctly, you see the stuff in Figure 3-13.

If you don't type the code exactly as it's shown in Listing 3-1, you may see jagged red underlines, red-colored words, or other red marks in the editor. In the Project tool window of Figure 3-14, the words `MyFirstProject`, `src`, and `Main` have jagged red underlines. The name `Main.java` on the tab above the editor has a jagged red underline. In the editor itself, a little red mark appears to the far right of Line 4. And, finally, the word `system` on Line 4 is a mean-looking, reddish color. (If your copy of Figure 3-14 has no colors, you'll have to take my word for that last assertion.)



```
1  public class Main {
2
3      public static void main(String[] args) {
4          System.out.println("Chocolate, royalties, sleep");
5      }
6  }
7
8
```

FIGURE 3-13: A Java program, in the IntelliJ editor.



```
1  public class Main {
2
3      public static void main(String[] args) {
4          system.out.println("Chocolate, royalties, sleep");
5      }
6  }
7
```

FIGURE 3-14: A Java program, typed incorrectly.

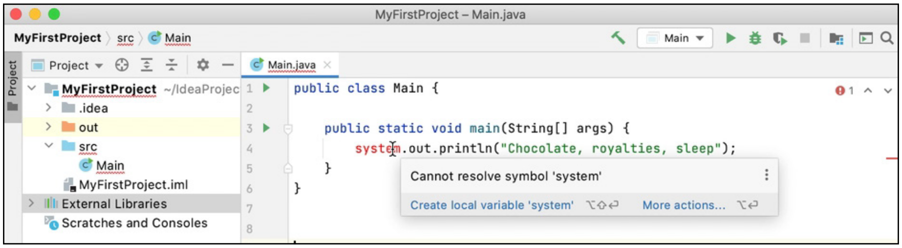
The red marks in IntelliJ's editor refer to compile-time errors in your Java code. A *compile-time error* (also known as a *compiler error*) is an error that prevents the computer from translating your code. (See the talk about code translation in Chapter 1.)

TIP

The error marker in Figure 3-14 appears on Line 4 of the Java program. Line numbers appear in the editor's left margin. You can make IntelliJ's editor start or stop displaying line numbers. To do so, choose File ➪ Settings (in Windows) or IntelliJ IDEA ➪ Preferences (on a Mac). In the resulting dialog box, choose Editor ➪ General ➪ Appearance. When you do all that, you see the Show Line Numbers check box.

To fix compile-time errors, you must become a dedicated detective. You join an elite squad known as *Law & Order: Java Programming Unit*. You seldom find easy answers. Instead, you comb the evidence slowly and carefully for clues. You compare everything you see in the editor, character by character, with my code in Listing 3-1. You don't miss a single detail, including spelling, punctuation, and uppercase-versus-lowercase.

IntelliJ IDEA has a few nice features to help you find the source of a compile-time error. For example, you can hover the mouse pointer over a red word or a jagged red underline. When you do, you see a brief explanation of the error. (See Figure 3-15.)



```
public class Main {

    public static void main(String[] args) {
        system.out.println("Chocolate, royalties, sleep");
    }
}
```

Cannot resolve symbol 'system'

FIGURE 3-15: What the heck does 'system' mean?

In Figure 3-15, a pop-up message tells you that Java doesn't know what the word *system* means — that is, *cannot resolve symbol 'system'*. Where I typed `system` with a lowercase letter `s`, Java wants `System` with an uppercase letter `S`. If I edit the code by replacing `system` with `System`, everything is fine. The red text becomes black, and all the intimidating red marks disappear.

DO I SEE FORMATTING IN MY JAVA PROGRAM?

When you use IntelliJ's editor to write a Java program, you see words in various colors. Certain words are always blue. Other words are always black. You even see some bold and italic phrases. You may think you see formatting, but you don't. Instead, what you see is called *syntax coloring* or *syntax highlighting*.

No matter what you call it, the issue is as follows:

- In Microsoft Word, things like bold formatting are marked inside a document. When you save `MyPersonalDiary.docx`, the instructions to make the words *love* and *hate* bold are recorded inside the `MyPersonalDiary.docx` file.
- In a Java program editor, things like bold and coloring aren't marked inside the Java program file. Instead, the editor displays each word in a way that makes the Java program easy to read.

For example, in a Java program, certain words (words like `class`, `public`, and `void`) have their own, special meanings. So IntelliJ's editor displays `class`, `public`, and `void` in blue letters. When I save my Java program, the computer stores nothing about colored letters in my Java program file. But the editor uses its discretion to highlight special words with blue coloring.

Certain other editors may display the same words in a bold, brown font. Another editor (like Windows Notepad) displays all words in plain old black.

WHAT CAN POSSIBLY GO WRONG?

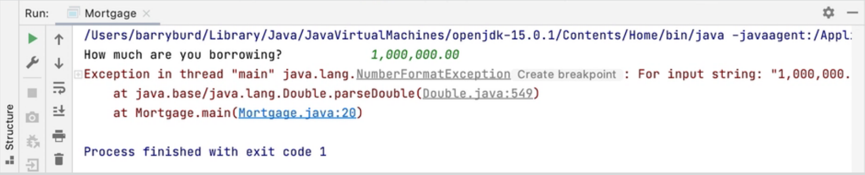
Ridding the editor of red words and jagged underlines is cause for celebration. IntelliJ likes the look of your code, so from that point on, it's smooth sailing. Right?

Well, it ain't necessarily so. In addition to some conspicuous compile-time errors, your code can have other, less obvious errors.

Imagine someone telling you to "go to the intersection and then *rurn tight*." You notice immediately that the speaker made a mistake, and you respond with a polite "Huh?" The nonsensical *rurn tight* phrase is like a compile-time error. Your "Huh?" is like the jagged underlines in IntelliJ's editor. As a listening human being, you may be able to guess what *rurn tight* means, but IntelliJ's editor never dares to fix your code's mistakes.

In addition to compile-time errors, some other kinds of gremlins can hide inside a Java program:

- **Unchecked runtime exceptions:** You have no compile-time errors, but when you run your program, the run ends prematurely. Somewhere in the middle of the run, your instructions tell Java to do something that can't be done. For example, while you're running the Mortgage program in the earlier "Running the code" section, you type 1,000,000.00 instead of 1000000.00. Java doesn't like the commas in the number, so your program crashes and displays a nasty-looking message, as shown in the sidebar figure.



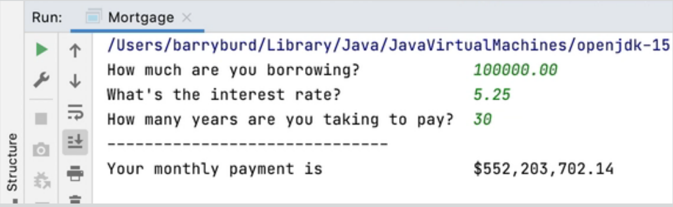
```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk-15.0.1/Contents/Home/bin/java -javaagent:/Appl:
How much are you borrowing?          1,000,000.00
Exception in thread "main" java.lang.NumberFormatException Create breakpoint : For input string: "1,000,000.
    at java.base/java.lang.Double.parseDouble(Double.java:549)
    at Mortgage.main(Mortgage.java:20)

Process finished with exit code 1
```

  This is an example of an *unchecked runtime exception* — the equivalent of someone telling you to turn right at the intersection when the only thing to the right is a big brick wall. IntelliJ's editor doesn't warn you about an unchecked runtime exception, because, until you run the program, the computer can't predict that the exception will occur.

- **Logic errors:** You see no error markers in IntelliJ's editor, and when you run your code, the program runs to completion. But the answer isn't correct. Instead of $552.20 in the figure, the output is $552,203,702.14. (See the next sidebar figure.) The program wrongly tells you to pay thousands of times what your house is worth and tells you to pay this amount each month! It's the equivalent of being told to turn right instead of turning left. You can drive in the wrong direction for a very long time.

(continued)

(continued)

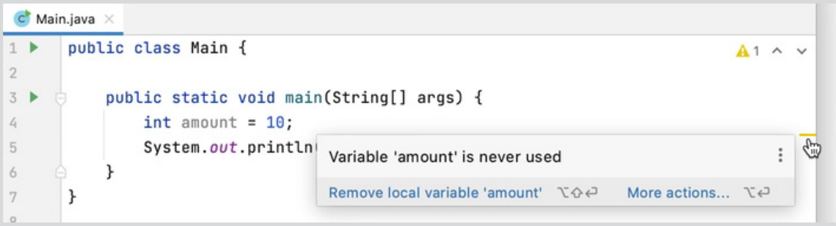


```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk-15
How much are you borrowing?             100000.00
What's the interest rate?               5.25
How many years are you taking to pay?   30
------------------------------
Your monthly payment is                 $552,203,702.14
```

Logic errors are the most challenging errors to find and to fix. And worst of all, logic errors often go unnoticed. In March 1985, I got a monthly home heating bill for $1,328,932.21. Clearly, some computer had printed the incorrect amount. When I called the gas company to complain about it, the telephone service representative said, "Don't be upset. Pay only half that amount."

- **Compile-time warnings:** A warning isn't as severe as an error message. So, when IntelliJ notices something suspicious in your program, the editor displays a non-intrusive clue.

  For example, in the second sidebar figure, I add something about `amount = 10` to the code from Listing 3-1. (It's that bit on Line 4.) The problem is, I never make use of `amount` or of the number 10 anywhere in my program. IntelliJ displays the word `amount` in a light-grey shade and a faint yellow mark appears to the far right of Line 4. When I hover over the word `amount` (or over the faint yellow mark), IntelliJ effectively tells me, "Your `amount = 10` code isn't bad enough to be a showstopper. IntelliJ can still manage to run your program. But are you sure you want `amount = 10` (the stuff that seems to serve no purpose) in your program?"



```
1  public class Main {
2
3      public static void main(String[] args) {
4          int amount = 10;
5          System.out.println
6      }
7  }
```

Variable 'amount' is never used

Imagine being told to "turn when you reach the intersection." The direction may be just fine. But if you're suspicious, you ask, "Which way should I turn? Left or right?"

When you're sure that you know what you're doing, you can ignore warnings and worry about them at some later time. But a warning can be an indicator that something more serious is wrong with your code. My sweeping recommendation is this: Pay attention to warnings. But, if you can't figure out why you're getting a particular warning, don't let the warning prevent you from moving forward.

WARNING

Some of IntelliJ's pop-up messages aren't as helpful as the one in Figure 3-15. If you don't understand a pop-up message, don't be discouraged. Just keep working with your code until you figure out what's wrong.

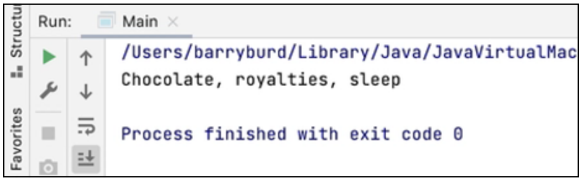
3. **Make any changes or corrections to the code in IntelliJ's editor.**

   When at last you see no red flags in the editor, you're ready to try running the program.

4. **Right-click either the Main branch in the Project tool window or the Main tab at the top of the editor.**

5. **On the resulting context menu, choose Run 'Main'.**

   After a brief delay, IntelliJ displays its Run tool window. The words `Chocolate, royalties, sleep` appear in the window. (See Figure 3-16.)



```
/Users/barryburd/Library/Java/JavaVirtualMac
Chocolate, royalties, sleep

Process finished with exit code 0
```

FIGURE 3-16: The best things in life.

What's All That Stuff in the IntelliJ IDEA Window?

Believe it or not, an editor once rejected one of my book proposals. In the margins, the editor scribbled, "This is not a word" next to things like *can't*, *it's*, and *I've*. To this day, I still do not know what this editor did not like about contractions. My own opinion is that language always needs to expand. Where would we be without new words — words like *crowdfunding*, *livestream*, and *cryptocurrency*?

Even the *Oxford English Dictionary* (the last word in any argument about words) grows by more than 4,000 entries each year. That's an increase of more than 1 percent per year. It's about 11 new words per day!

The fact is, human thought is like a big high-rise building: You can't build the 50th floor until you've built at least part of the 49th. You can't talk about *spam* until you have a word like *email*. With all that goes on these days, you need verbal building blocks. That's why this section contains a bunch of new terms.

In this section, each newly defined term describes an aspect of IntelliJ IDEA. So, before you read all this IntelliJ terminology, I provide the following disclaimers:

- **This section is optional reading.** Refer to this section if you have trouble understanding some of this book's instructions. But if you have no trouble navigating IntelliJ IDEA, don't complicate things by fussing over the terminology in this section.
- **This section provides explanations of terms, not formal definitions of terms.** Yes, my explanations are fairly precise, but no, they're not airtight. Almost every description in this section has hidden exceptions, omissions, exemptions, and exclusions. Take the paragraphs in this section to be friendly reminders, not legal contracts.
- **IntelliJ is a very useful tool.** But IntelliJ isn't officially part of the Java ecosystem. Although I don't describe details in this book, you can write Java programs without ever using IntelliJ.

In this section, you get an overview of IntelliJ IDEA's main window. I focus on the most useful features that help you create Java programs, but keep in mind that IntelliJ IDEA has hundreds of features and many ways to access each feature.

Starting up

Each Java program belongs to a project. You can have dozens of projects on your computer's hard drive. When you run IntelliJ IDEA, each of your projects is either open or closed. An *open* project appears in a window (its own window) on your computer screen. A *closed* project doesn't appear in a window.

Several of your projects can be open at the same time. You can switch between projects by moving from window to window.

TECHNICAL STUFF

I often refer to an open project's window as IntelliJ IDEA's *main window*. This term can be slightly misleading because, with several projects open at a time, you have several main windows open at a time. In a way, none of these windows is more "main" than the others. When I write *main window*, I'm referring to the window whose Java project you're working on at that moment.

If IntelliJ IDEA is running and no projects are open, IntelliJ displays its Welcome screen. (See Figure 3-17.) The Welcome screen may display some recently closed projects. If so, you can open a project by clicking its name on the Welcome screen. For an existing project that's not on the Recent Projects list, you can click the Welcome screen's Open button.

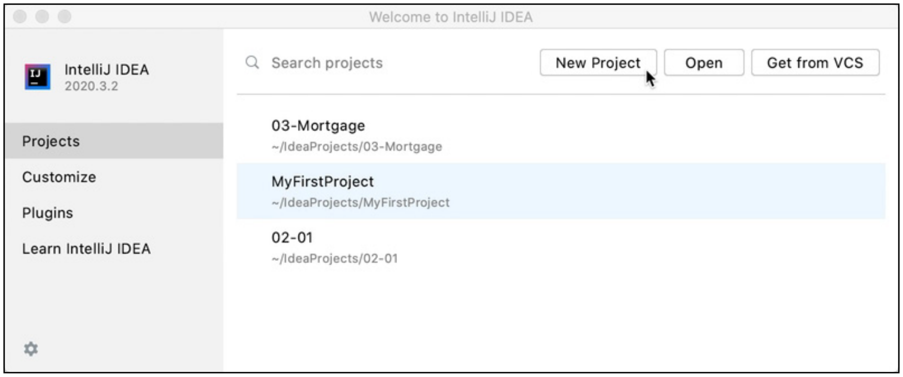


FIGURE 3-17: Look familiar?

If you have any open projects, IntelliJ doesn't display the Welcome screen. In that case, you can open another project by choosing File➪Open or File➪Open Recent in an open project's window. To close a project, you can choose File➪Close Project, or you can do whatever you normally do to close one of the windows on your computer. (On a PC, click the X in the window's upper right corner. On a Mac, click the little red button in the window's upper left corner.)

TIP

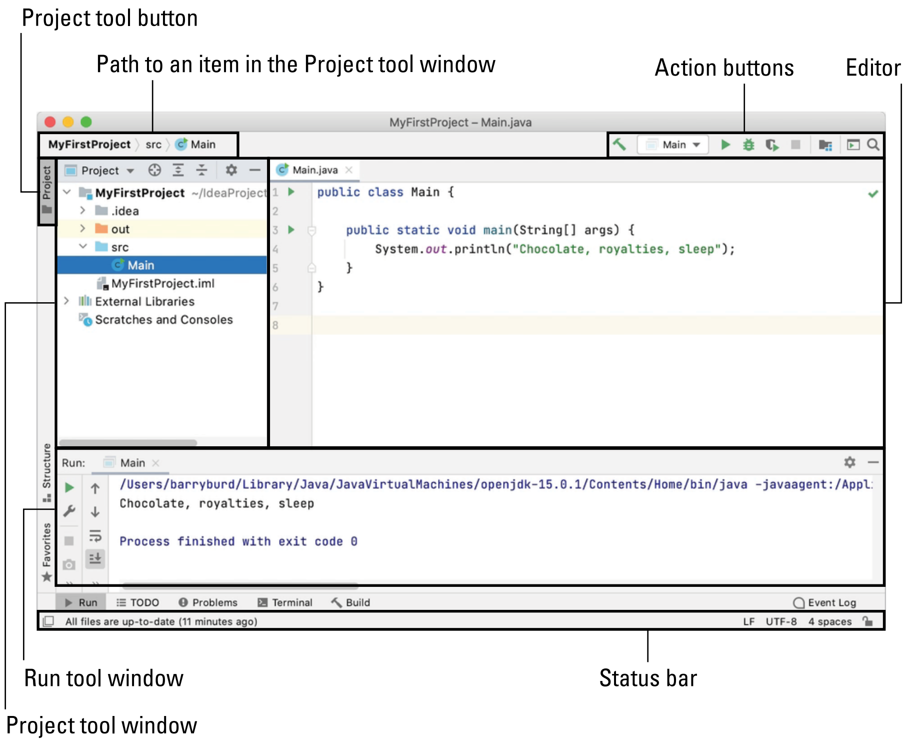
IntelliJ IDEA remembers which projects were open from one run to the next. If any projects are open when you quit IntelliJ, those projects open again (with their main windows showing) the next time you launch IntelliJ. You can override this behavior (so that only the Welcome screen appears each time you launch IntelliJ). With IntelliJ on a Windows computer, start by choosing File➪Settings➪Appearance and Behavior➪System Settings. With IntelliJ on a Mac, choose IntelliJ IDEA➪Preferences➪Appearance and Behavior➪System Settings. In either case, uncheck the Reopen Projects on Startup check box.

The main window

IntelliJ's main window is divided into several areas. Some of these areas can appear and disappear on your command. What comes next is a description of the areas in Figure 3-18, moving from the top of the main window to the bottom.

REMEMBER

The areas you see on your computer screen may differ from the areas in Figure 3-18. Usually, that's okay. You can make areas come and go by choosing certain menu options, including the View option on IntelliJ's main menu bar. You can also click the little tool buttons on the edges of the main window.



```
public class Main {

    public static void main(String[] args) {
        System.out.println("Chocolate, royalties, sleep");
    }
}
```

```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk-15.0.1/Contents/Home/bin/java -javaagent:/Appl:
Chocolate, royalties, sleep

Process finished with exit code 0
```

All files are up-to-date (11 minutes ago)

FIGURE 3-18: The main window has several areas.

The top of the main window

The topmost area contains the navigation bar and the toolbar.

- **The *navigation bar* displays the path to one of the branches in the Project tool window.**

  If you can't see the forest for the Project tool window's tree, check the navigation bar.

- **The *toolbar* contains action buttons, such as Run, Debug, and Stop.**

  When you hover over a button, IntelliJ displays a hint telling you what that button does.

The Project tool window

Below the main menu and the toolbars, you see two different areas. The area on the left contains the *Project tool window*, which you use to navigate from one file to another within your Java project.

At any given moment, the Project tool window displays one of several possible views. For example, back in Figure 3-18, the Project tool window displays its *Project view*. In Figure 3-19, I click the drop-down list and select Packages view (instead of Project view).

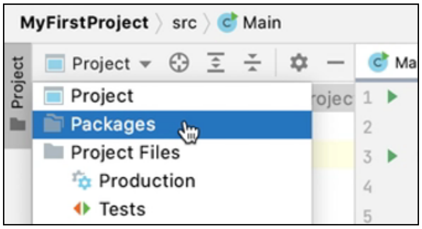


FIGURE 3-19: Selecting Packages view.

Packages view displays many of the same files as Project view, but in Packages view, the files are grouped differently. For most of this book's instructions, I assume that the Project tool window is in its default view — namely, Project view.

REMEMBER

If IntelliJ doesn't display the Project tool window, look for the Project tool button — the little button displaying the word *Project* on the left edge of the main window. Click that Project tool button. (But wait! What if you can't find the little *Project* button? In that case, go to IntelliJ's main menu and choose Window➪Restore Default Layout.)

The editor area

The area to the right of the Project tool window is the *editor area*. When you edit a Java program file, the editor displays the file's text. (Refer to Figure 3-18.) You can type, cut, copy, and paste text as you would in other text editors.

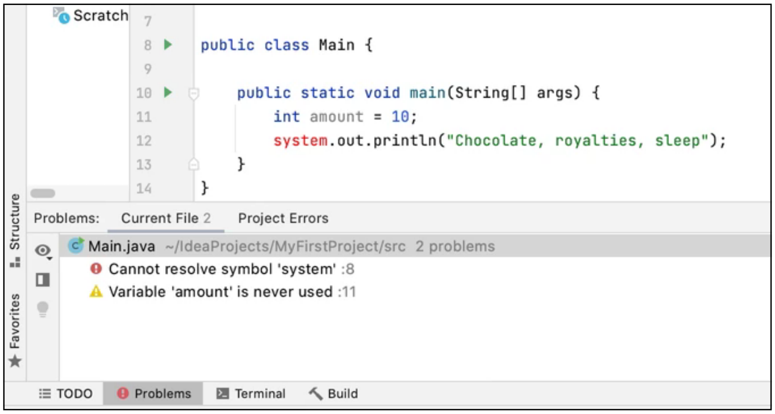
The editor area can have several tabs. Each tab contains a file that's open for editing. To open a file for editing, double-click the file's branch in the Project tool window. To close the file, click the little x next to the file's name on the editor tab.

The lower area

Below the Project tool window and the editor area is another area that contains several tool windows. When you're not using any of these tool windows, you might not see this lower area.

In the lower area, the tool window that I use most often is the Run tool window. (Refer to the lower third of the window in Figure 3-18.) The Run tool window appears automatically when you start running a program. This tool window displays information about the run of a Java program. If your program isn't running correctly, the Run tool window may contain useful diagnostic information.

You can force other tool windows to appear in the lower area by clicking tool buttons near the bottom of the IntelliJ window. For example, when you click the Problems tool button, IntelliJ lists any errors it finds in your code along with any warnings about suspicious-looking aspects of your code. The Problems tool window in Figure 3-20 shows an unusable `Main.java` file and the corresponding Problems tool window.



```
public class Main {

    public static void main(String[] args) {
        int amount = 10;
        system.out.println("Chocolate, royalties, sleep");
    }
}
```

Main.java ~/IdeaProjects/MyFirstProject/src 2 problems

Cannot resolve symbol 'system' :8

Variable 'amount' is never used :11

FIGURE 3-20: The bearer of bad news.

WARNING

A particular tool button might not appear when there's nothing you can do with it. For example, the Run tool button might not appear until you start running a program. Don't worry about that. The tool button shows up whenever you need it.

REMEMBER

In the IntelliJ window, the Run button is different from the Run tool button. The Run button is a green "play" icon — one of the action buttons near the top of the window. The Run tool button is a rectangular region containing a dark grey "play" icon alongside the word "Run." The Run tool button is in the lower left part of the IntelliJ window. You can see both of these buttons in Figure 3-18.

Finishing your tour of the areas in Figure 3-18. . . .

The status bar

The status bar is at the bottom of the IntelliJ window.

The status bar tells you what's happening now. For example, if the cursor is on the 37th character of the 11th line in the editor, you see `11:37` somewhere on the status line. When you tell IntelliJ to run your app, the status bar contains the Run tool window's most recent message.

The kitchen sink

In addition to the areas that I mention in this section, other areas might pop up as the need arises. You can dismiss an area by clicking its Hide button. (See Figure 3-21.)

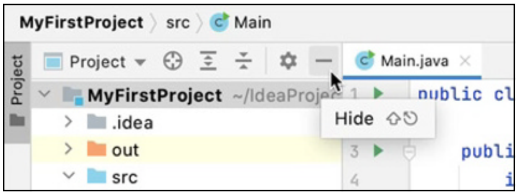


FIGURE 3-21: Hiding the area that contains the Project tool window.

TRY IT OUT

Here are some things for you to try to help you understand the material in this chapter. If trying these things builds your confidence, that's good. If trying these things makes you question what you've read, that's good, too. If trying these things makes you nervous, don't be discouraged. You can find answers and other help at this book's website (`http://beginprog.allmycode.com`). You can also email me with your questions (`BeginProg@allmycode.com`).

INTELLIJ IDEA BASICS

Follow the instructions in this chapter's earlier section "Running a Canned Java Program." Then try the following tasks:

- Make sure you can see the mortgage-calculating program's code in the window on the left side of the IntelliJ window. Expand the `03–Mortgage` branch until you see a branch labeled `Mortgage`. Double-click the `Mortgage` branch.
- In the IntelliJ editor, make any change to the text in the mortgage-calculating program. After making the change, undo the change by choosing Edit ➪ Undo from IntelliJ's main menu.
- Look for IntelliJ's Terminal tab in the lower portion of the IntelliJ window. If you don't see that tab, make the Terminal tab appear by clicking the Terminal tool button at the bottom of the IntelliJ window.
- The IntelliJ window has several areas. Use the mouse to drag the boundaries between the areas (and thus resize each of the areas). To return the areas to the way they were before resizing, choose Window ➪ Restore Default Layout from IntelliJ's main menu.

EXPERIMENTING WITH ERROR MESSAGES

Follow the instructions in this chapter's earlier section "Some Programs Don't Come in Cans" Look for `MyFirstProject` in IntelliJ's Project tool window. As you expand that `MyFirstProject` branch, look for a branch labeled `Main`. When you double-click the `Main` branch, the code for the `Main.java` program appears in the IntelliJ editor.

Try these two experiments:

- In the IntelliJ editor, change the lowercase letter `c` in the word `class` to an uppercase letter `C`. When you do this, notice that lots of red underlines appear. These red underlines indicate that your program has a compile-time error. Java is case-sensitive. So, in a Java program, the word `Class` (with an uppercase letter `C`) doesn't mean the same thing as the word `class` (with a lowercase letter `c`).

  There are a few places in `MyFirstProject` where changing the capitalization doesn't cause errors. But for most of the text, a change in capitalization causes red error warnings to appear in the IntelliJ editor.

- In the IntelliJ editor, change

```
System.out.println("You'll love Java!");
```

- to

```
System.out.println(6/0);
```

  No error markers appear in the IntelliJ editor. But, when you try to run the program, you see `ArithmeticException` in IntelliJ's Run tool window. The red text indicates that an arithmetic exception has occurred. In Java, any attempt to divide by 0 gives you an `ArithmeticException`.

CHANGING A NAME

Follow the instructions in this chapter's earlier section "Some Programs Don't Come in Cans" and look for the places where the word `Main` appears. (That's `Main` with an uppercase `M`, not `main` with a lowercase `m`.) In the IntelliJ editor, change

```
public class Main {
```

to

```
public class ThingsILike {
```

When you make this change, IntelliJ complains with some red underlines, so undo the change before anyone else sees it!

Now, rather than change `Main` in the editor, right-click the word `Main` in the Project tool window. On the resulting context menu, choose Refactor ➪ Rename. When a dialog box appears, type **ThingsILike** in the box's text field.

After dismissing the dialog box, look for two occurrences of the name `ThingsILike` — one in the Project tool window and another in the editor.

2

Writing Your Own Java Programs

IN THIS PART . . .

Dissecting programs and examining the pieces

Working with numbers

Working with things that aren’t numbers

IN THIS CHAPTER

- » Identifying the words in a Java program
- » Using punctuation and indentation
- » Understanding Java statements and methods

Chapter 4

Exploring the Parts of a Program

I work in the science building at a liberal arts college. When I walk past the biology lab, I always say a word of thanks under my breath. I'm thankful for not having to dissect small animals. In my line of work, I dissect computer programs instead. Computer programs smell much better than preserved dead animals. Besides, when I dissect a program, I'm not reminded of my own mortality.

In this chapter, I invite you to dissect a program with me. I have a small program, named `ThingsILike`. I cut apart the program and carefully investigate the program's innards. Get your scalpel ready. Here we go!

Checking Out Java Code for the First Time

I have a confession to make. The first time I look at somebody else's computer program, I feel a bit queasy. The realization that I don't understand something (or many things) in the code makes me nervous. I've written hundreds (maybe thousands) of programs, but I still feel insecure whenever I start reading someone else's code.

The truth is, learning about a computer program is a bootstrapping experience. First, I gawk in awe of the program. Then I run the program to see what it does. Then I stare at the program for a while or read someone's explanation of the program and its parts. Then I gawk a little more and run the program again. Eventually, I come to terms with the program. Don't believe the wise guys who say they never go through these steps. Even experienced programmers approach a new project slowly and carefully.

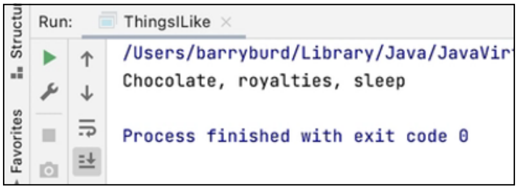
Behold! A program!

In Listing 4-1, you get a blast of Java code. Like all novice programmers, you're expected to gawk humbly at the code. *Don't be intimidated.* When you get the hang of it, programming is pretty easy. Yes, it's fun, too.

LISTING 4-1: A Simple Java Program

```
/*
 * A program to list the good things in life
 * Author: Barry Burd, BeginProg@allmycode.com
 * February 13, 2021
 */

public class ThingsILike {

    public static void main(String[] args) {
        System.out.println("Chocolate, royalties, sleep");
    }
}
```

When I run the program in Listing 4-1, I get the result shown in Figure 4-1: The computer shows the text `Chocolate, royalties, sleep` in IntelliJ's Run tool window. Now, I admit that writing and running a Java program is a lot of work just to display `Chocolate, royalties, sleep`, but every endeavor has to start somewhere.



```
/Users/barryburd/Library/Java/JavaVir
Chocolate, royalties, sleep

Process finished with exit code 0
```

FIGURE 4-1: Running the program in Listing 4-1.

ON THE WEB

Most of the programs in this book are command line programs. When you run one of these programs, the input and output appear in a text-only part of the screen — a place like IntelliJ's Run tool window. In contrast, a GUI (*g*raphical *u*ser *i*nterface) program displays windows, buttons, text fields, and other widgets to interact with the user. You can see GUI versions of the program in Listing 4-1, and in many other examples from this book, by visiting the book's website (`http://beginprog.allmycode.com`).

You can run the code in Listing 4-1 on your computer. Here's how:

1. **Follow the instructions in Chapter 2 for installing IntelliJ IDEA and Java.**
2. **Then follow the instructions in the first half of Chapter 3.**

   Those instructions tell you how to run the project named `03-Mortgage`, which comes in a download from this book's website (`http://beginprog.allmycode.com`). To run the code in Listing 4-1, follow the same instructions for the `04-01` project, which comes in the same download.

What the program's lines say

If the program in Listing 4-1 ever becomes famous, someone will write a *Cliffs Notes* book to summarize the program. The book won't have many pages because you can summarize the action of Listing 4-1 in just one sentence. Here's the sentence:

```
Display Chocolate, royalties, sleep in a text-only area.
```

Now compare the preceding sentence with the bulk in Listing 4-1. Because Listing 4-1 has many more lines, you may guess that it has lots of boilerplate code. Well, your guess is correct. You can't write a Java program without writing the boilerplate stuff, but, fortunately, the boilerplate text doesn't change much from one Java program to another. Here's my best effort at summarizing all the Listing 4-1 text in 66 words or fewer:

```
This program lists the good things in life.
Barry Burd wrote this program on February 13, 2021.
Barry realizes that you may have questions about this
code, so you can reach him at BeginProg@allmycode.com.

This code defines a Java class named ThingsILike.
    Here's the main starting point for the instructions:
        Display Chocolate, royalties, sleep in a text-only area.
```

The rest of this chapter (about 5,000 more words) explains the Listing 4-1 code in more detail.

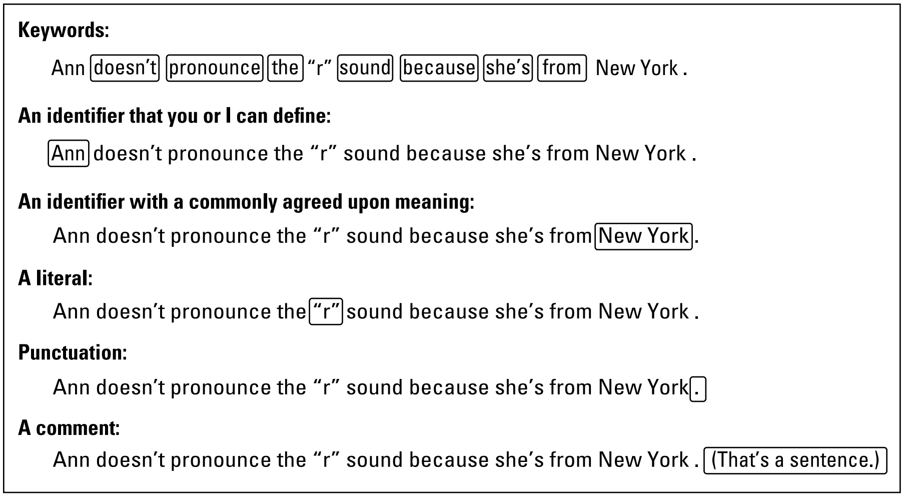
The Elements in a Java Program

That both English and Java are called *languages* is no coincidence. You use a language to express ideas. English expresses ideas to people, and Java expresses ideas to computers. What's more, both English and Java have things like words, names, and punctuation. In fact, the biggest difference between the two languages is that Java is easier to learn than English. (If English were easy, computers would understand English. Unfortunately, they can't.)

Take an ordinary English sentence and compare it with the code in Listing 4-1. Here's the sentence:

> Ann doesn't pronounce the "r" sound because she's from New York.

In your high school grammar class, you worried about verbs, adjectives, and other parts of speech. But in this book, you think in terms of keywords and identifiers, as summarized in Figure 4-2.



Keywords:

Ann doesn't pronounce the "r" sound because she's from New York .

An identifier that you or I can define:

Ann doesn't pronounce the "r" sound because she's from New York .

An identifier with a commonly agreed upon meaning:

Ann doesn't pronounce the "r" sound because she's from New York.

A literal:

Ann doesn't pronounce the "r" sound because she's from New York .

Punctuation:

Ann doesn't pronounce the "r" sound because she's from New York.

A comment:

Ann doesn't pronounce the "r" sound because she's from New York . (That's a sentence.)

FIGURE 4-2: The things you find in a simple sentence.

Ann's sentence has all kinds of things in it. They're the same kinds of things that you find in a computer program. So here's the plan: Compare the elements in Figure 4-2 with similar elements in Listing 4-1. You already understand English, so you can use this understanding to figure out some new things about Java.

But first, here's a friendly reminder: In the next several paragraphs, I draw comparisons between English and Java. As you read these paragraphs, keep an open mind. In comparing Java with English, I may write, "Names of things aren't the same as dictionary words." Sure, you can argue that some dictionaries list proper nouns and that some people have first names like Hope, Prudence, and Spike, but please don't. You'll get more out of the reading if you avoid nitpicking. Okay? Are we still friends?

Keywords and their close cousins

A *keyword* is a dictionary word — a word that's built right into a language.

In Figure 4-2, a word like *from* is a keyword because *from* plays the same role whenever it's used in an English sentence. The other keywords in Ann's sentence are *doesn't, pronounce, the, sound, because,* and *she's.*

Computer programs have keywords, too. In fact, the program in Listing 4-1 uses some of Java's keywords (shown in bold):

```
public class ThingsILike {

    public static void main(String[] args) {
```

Each Java keyword has a specific meaning — a meaning that remains unchanged from one program to another. For example, whenever I write a Java program, the word `public` always signals a part of the program that's accessible to any other piece of code.

WARNING

The java proGRAMMing lanGUage is *case-sensitive*. ThIS MEans that if you change a lowerCASE LETTer in a wORD TO AN UPPercase letter, you chANge the wORD'S MEaning. ChangiNG CASE CAN MakE the enTIRE WORD GO FROM BeiNG MEANINGFul to bEING MEaningless. In Listing 4-1, you can't replace `public` with `Public`. If you do, the WHOLE PROGRAM STOPS WORKING.

This chapter has little or no detail about the meanings of the keywords `class`, `public`, `static`, and `void`. You can peek ahead at the material in other chapters, or you can get along by cheating. When you write a program, just start with

```
public class SomethingOrOther {
```

followed by

```
public static void main(String[] args) {
```

In your first few programs, this strategy serves you well.

Table 4-1 has a complete list of Java keywords.

TABLE 4-1: Java Keywords

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while
_ *(underscore)*				

Here's one thing to remember about keywords: In Java, each keyword has an official, predetermined meaning. The people who have the final say on what constitutes a Java program created all of Java's keywords. You can't make up your own meaning for any of the Java keywords. For example, you can't use the word `public` in a calculation:

```
//This is BAD, BAD CODE:
public = 6;
```

If you try to use a keyword this way, the compiler displays an error message and refuses to translate your source code. It works the same way in English. Have a baby and name it Because:

> "Let's have a special round of applause for tonight's master of ceremonies — Because O. Borel."

You can do it, but the kid will never lead a normal life.

In addition to keywords, Java has *restricted identifiers*, *restricted keywords*, *boolean literals*, and a *null literal*. (See Table 4-2.) The words in these categories are somewhat like keywords, but, for technical reasons, they're not really called keywords. One way or another, I recommend handling these words with kid gloves. Don't use a word in Table 4-2 without considering the word's official meaning.

TABLE 4-2: Treat These Words as If They're Java Keywords

`exports`	`null`	`provides`	`transitive`	`var`
`false`	`open`	`requires`	`true`	`with`
`module`	`opens`	`to`	`uses`	`yield`

TECHNICAL STUFF

Despite my ardent claims in this section, two of Java's keywords have no meaning in a Java program. Those keywords — `const` and `goto` — are reserved for nonuse in Java. If you try to create a variable named `goto`, IntelliJ displays an error message. The creators of Java figure that if you use either of the words `const` or `goto` in your code, you should be told politely to move to the C++ programmers' table.

Identifiers that you or I can define

I like the name Ann, but if you don't like traditional names, make up a brand-new name. You're having a new baby. Call her Deneen or Chrisanta. Name him Belton or Merk.

A *name* is a word that identifies something, so I'll stop calling these things names and start calling them *identifiers*. In computer programming, an *identifier* is a noun of some kind. An identifier refers to a value, a part of a program, a certain kind of structure, or any number of things.

Listing 4-1 has two identifiers that you or I can define on our own. They're the made-up words `ThingsILike` and `args`.

```
public class ThingsILike {

    public static void main(String[] args) {
```

Just as the names Ann and Chrisanta have no special meaning in English, the names `ThingsILike` and `args` have no special meaning in Java. In Listing 4-1, I use `ThingsILike` for the name of my program, but I could also have used a name like `GooseGrease`, `Enzyme`, or `Kalamazoo`. I have to put `(String[] someName)` in my program, but I could use `(String[] args)`, `(String[] commandLineArguments)` or `(String[] cheese)`.

TIP

Make up sensible, informative names to use in your Java programs. Names like GooseGrease are legal, and they're certainly cute, but they don't help you keep track of your program writing strategy.

REMEMBER

When I name my Java program, I can use ThingsILike or GooseGrease, but I can't use the word public. Words like class, public, static, and void are keywords in Java.

The args in (String[] args) holds extra values that you type when you issue the command to run a Java program. For example, there's a way to run Listing 4-1 by typing **java ThingsILike won too 3**. When you do, args refers to the extra values won, too, and 3. As a beginning programmer, you don't need to think about this feature of Java. Just have (String[] args) in each of your programs.

Identifiers with agreed-upon meanings

Many people are named Ann, but only one well-known city is named New York. That's because there's a standard, well-known meaning for the term *New York.* It's the city that never sleeps. If you start your own city, you should avoid naming it New York, because naming it New York would just confuse everyone. (I know, a town in Florida is named New York, but that doesn't count. Remember to ignore exceptions like this one.)

Most programming languages have identifiers with agreed-upon meanings. In Java, almost all these identifiers are defined in the Java API. Listing 4-1 has five such identifiers. They're the words main, String, System, out, and println:

```
public static void main(String[] args) {
    System.out.println("Chocolate, royalties, sleep");
}
```

Here's a quick rundown on the meaning of each of these names (and more detailed descriptions appear throughout this book):

- **main:** The main starting point for execution in every Java program.
- **String:** A bunch of text; a row of characters, one after another.
- **System:** A canned program in the Java API. This program accesses some features of your computer that are outside the direct control of the Java virtual machine (JVM).

- **out:** The place where a command line program displays its text. (For a program running in IntelliJ, the word `out` represents the Run tool window. To read more about command line programs, check the first several paragraphs of Chapter 3.)
- **println:** Displays text on your computer screen.

REMEMBER

The name `println` comes from the words "print a *line*." If you were allowed to write the name in uppercase letters, it would be `PRINTLN`, with a letter `L` near the end of the word. When the computer executes `println`, the computer puts some text in IntelliJ's Run tool window and then immediately *moves to the beginning of the next line*, in preparation for whatever else will appear in the Run tool window.

TECHNICAL STUFF

Strictly speaking, the meanings of the identifiers in the Java API aren't cast in stone. Although you can make up your own meanings for words like `System` or `println`, doing so isn't a good idea — because you'd confuse the dickens out of other programmers, who are used to the standard API meanings for these familiar identifier names.

Literals

A *literal* is a chunk of text that looks like whatever value it represents. In Ann's sentence (refer to Figure 4-2), "r" is a literal because "r" refers to the letter *r*.

Programming languages have literals, too. For example, in Listing 4-1, the stuff in double quotes is a literal:

```
System.out.println("Chocolate, royalties, sleep");
```

When you run the `ThingsILike` program, you see the text `Chocolate, royalties, sleep` in IntelliJ's Run tool window. In Listing 4-1, the text "`Chocolate, royalties, sleep`" refers to this text, exactly as it appears on the screen (minus the quotation marks).

Depending on the context, I may refer to `"Chocolate, royalties, sleep"` as a *string literal*, a *string of characters*, or simply a *string*. One way or another, the word *string* refers to a bunch of characters, one after another.

TECHNICAL STUFF

You're probably not surprised if I tell you that `String` plays an important role in Java. Near the start of each program, the word `String` refers to strings of characters. For more insight on the meaning of `String`, see Chapter 14.

Most of the numbers that you use in computer programs are literals. If you put the statement

```
mySalary = 1000000.00;
```

in a computer program, then 1000000.00 is a literal. It stands for the number 1000000.00 (one million).

If you don't enjoy counting digits, you can put the following statement in your program:

```
mySalary = 1_000_000.00;
```

Starting with Java 7, numbers with underscores are permissible as literals.

TECHNICAL STUFF

Different countries use different number separators and different number formats. For example, in the United States, you write 1,234,567,890.55. In France, you write 1234567890,55. In India, you group digits in sets of two and three. You write 1,23,45,67,890.55. In Switzerland, you may write 1'234'567'890.55 for currency and 1 234 567 890,55 for other numbers. No matter where you live, you can't put a statement like mySalary = 1,000,000.00 in your Java program. Java's numeric literals have no commas in them. But you can write mySalary = 10_00_000.00 for easy-to-read programming in India. And for a program's output, you can display numbers like 1234567890,55 using Java's Locale and NumberFormat classes. (For more on Locale and NumberFormat, check out Chapter 14.)

Punctuation

A typical computer program has lots of punctuation. For example, consider the program in Listing 4-1:

```
public class ThingsILike {

    public static void main(String[] args) {
        System.out.println("Chocolate, royalties, sleep");
    }
}
```

Each bracket, each brace, each squiggle of any kind plays a role in making the program meaningful.

In English, you write all the way across one line and then you wrap the text to the start of the next line. In programming, you seldom work this way. Instead, the code's punctuation guides the indenting of certain lines. The indentation shows which parts of the program are subordinate to which other parts. It's as though, in English, you wrote a sentence like this:

Ann doesn't pronounce the "r" sound because

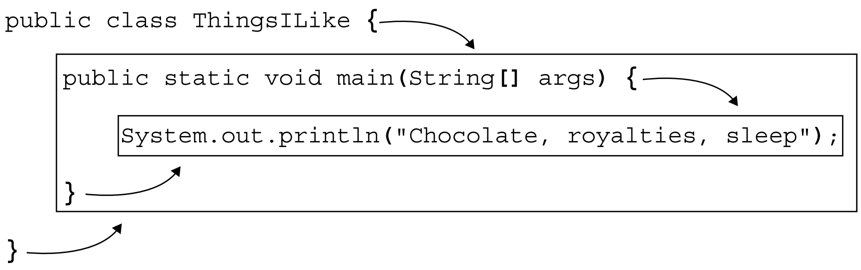
,

as we all know

,

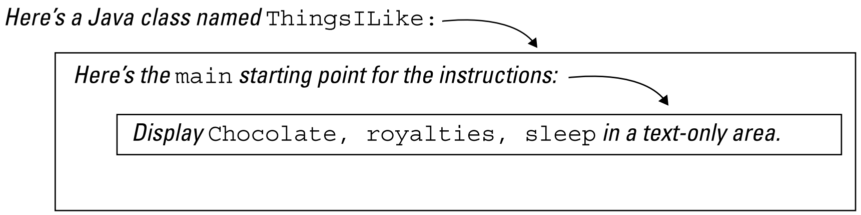
she's from New York.

The diagrams in Figures 4-3 and 4-4 show you how parts of the `ThingsILike` program are contained inside other parts. Notice how a pair of curly braces acts like a box. To make the program's structure visible at a glance, you indent all the stuff inside each box.



```
public class ThingsILike {
    public static void main(String[] args) {
        System.out.println("Chocolate, royalties, sleep");
    }
}
```

FIGURE 4-3: A pair of curly braces acts like a box.



Here's a Java class named `ThingsILike`:

Here's the `main` *starting point for the instructions:*

Display `Chocolate, royalties, sleep` *in a text-only area.*

FIGURE 4-4: The ideas in a computer program are nested inside one another.

REMEMBER

I can't emphasize this point enough: If you don't indent your code or if you indent but you don't do it carefully, your code still compiles and runs correctly. But this successful run gives you a false sense of confidence. The minute you try to update some poorly indented code, you become hopelessly confused. Take my advice:

Keep your code carefully indented at every step in the process. Make its indentation precise, whether you're scratching out a quick test program or writing code for a billionaire customer.

TIP

IntelliJ can indent your code automatically for you. Click the mouse in IntelliJ's editor. (Don't select text. Just click.) Then, on IntelliJ's main menu, choose Code➪Reformat Code. IntelliJ rearranges the lines in the editor, indenting things that should be indented and generally making your code look good.

Comments

A *comment* is text that's outside the normal flow. In Figure 4-2, the words "That's a sentence" aren't part of the Ann sentence. Instead, these words are *about* the Ann sentence.

The same is true of comments in computer programs. The first five lines in Listing 4-1 form one big comment. The computer doesn't act on this comment. There are no instructions for the computer to perform inside this comment. Instead, the comment tells other programmers something about your code.

Comments are for your own benefit, too. Imagine that you set aside your code for a while and work on something else. When you return later to work on the code again, the comments help you remember what you were doing.

The Java programming language has three kinds of comments:

- **Traditional comments:** The comment in Listing 4-1 is a *traditional* comment. The comment begins with /* and ends with */. Everything between the opening /* and the closing */ is for human eyes only. Nothing between /* and */ gets translated by the compiler.

  The second, third, and fourth lines in Listing 4-1 have extra asterisks. I call them "extra" because these asterisks aren't required when you create a comment. They just make the comment look pretty. I include them in Listing 4-1 because, for some reason that I don't entirely understand, most Java programmers add these extra asterisks.

- **End-of-line comments:** Here's some code with end-of-line comments:

```
public class ThingsILike {                   // Two things are missing

    public static void main(String[] args) {
      System.out.println("sleep");           // Missing from here
    }
}
```

An *end-of-line* comment starts with two slashes and extends to the end of a line of type.

TIP

You may hear programmers talk about *commenting out* certain parts of their code. When you're writing a program and something's not working correctly, it often helps to remove some of the code. If nothing else, you find out what happens when that suspicious code is removed. Of course, you may not like what happens when the code is removed, so you don't want to delete the code completely. Instead, you turn your ordinary Java statements into comments. For example, turn

```
System.out.println("Sleep");
```

- into

```
// System.out.println("Sleep");
```

This line keeps the Java compiler from seeing the code while you try to figure out what's wrong with your program.

- **Javadoc comments:** A special *Javadoc* comment is any traditional comment that begins with two asterisks (not just one):

```
/**
 * Print a String and then terminate the line.
 */
```

This is a cool Java feature. The Java software that you download in Chapter 2 includes a little program called `javadoc`. The `javadoc` program looks for these special comments in your code. The program uses these comments to create a brand-new web page — a customized documentation page for your code. To find out more about turning Javadoc comments into web pages, visit this book's website (`http://beginprog.allmycode.com`).

Understanding a Simple Java Program

The following sections present, explain, analyze, dissect, and otherwise demystify the Java program in Listing 4-1.

What is a method?

You're working as an auto mechanic in an upscale garage. Your boss, who's always in a hurry and has a habit of running words together, says, "fixTheAlternator on that junkyOldFord." Mentally, you run through a list of tasks. "Drive the car into the bay, lift the hood, get a wrench, loosen the alternator belt," and so on. Three things are going on here:

- **You have a name for the thing you're supposed to do.** The name is fixTheAlternator.
- **In your mind, you have a list of tasks associated with the name fixTheAlternator.** The list includes "Drive the car into the bay, lift the hood, get a wrench, loosen the alternator belt," and so on.
- **You have a grumpy boss who's telling you to do all this work.** Your boss gets you working by saying, "fixTheAlternator." In other words, your boss gets you working by saying the name of the thing you're supposed to do.

In this scenario, using the word *method* isn't a big stretch. You have a method for doing something with an alternator. Your boss calls that method into action, and you respond by doing all the things in the list of instructions that you've associated with the method.

Java methods

If you believe all that stuff in the preceding section, you're ready to read about Java methods. In Java, a *method* is a list of things to do. Every method has a name, and you tell the computer to do the things in the list by using the method's name in your program.

I've never written a program to get a robot to fix an alternator. But, if I were to, the program might include a method named `fixTheAlternator`. The list of instructions in my `fixTheAlternator` method would look something like the text in Listing 4-2.

LISTING 4-2: A Method Declaration

```
void fixTheAlternator(onACertainCar) {
  driveInto(car, bay);
  lift(hood);
  get(wrench);
  loosen(alternatorBelt);
  ...
}
```

Somewhere else in my Java code (somewhere outside of Listing 4-2), I need an instruction to call my `fixTheAlternator` method into action. The instruction to call the `fixTheAlternator` method into action may look like the line in Listing 4-3.

LISTING 4-3: Calling a Method

```
fixTheAlternator(junkyOldFord);
```

WARNING

Don’t scrutinize Listings 4-2 and 4-3 too carefully. These listings are fakes! I made up these examples so that they look like Java code, but you can’t run the code in these two listings. If you have a grain of salt handy, take Listings 4-2 and 4-3 with it.

TECHNICAL STUFF

Almost every computer programming language has something akin to Java’s methods. If you’ve worked with other languages, you may remember things like subprograms, procedures, functions, subroutines, `Sub` procedures, or `PERFORM` statements. Whatever you call it in your favorite programming language, a *method* is a bunch of instructions collected together and given a new name.

The declaration, the header, and the call

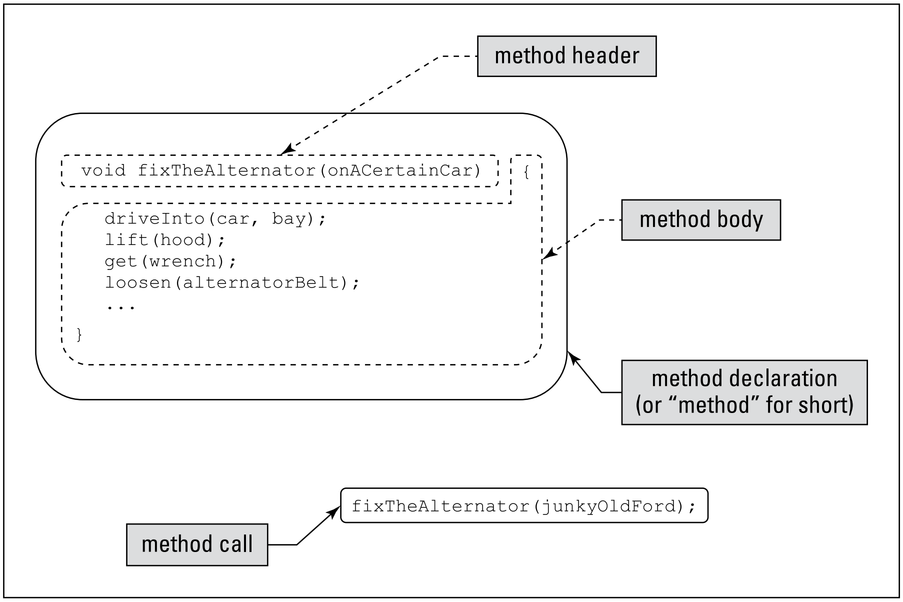
If you have a basic understanding of what a method is and how it works (see the preceding section), you can dig a little deeper into some useful terminology:

- If I’m being lazy, I refer to the code in Listing 4-2 as a *method*. If I’m not being lazy, I refer to this code as a *method declaration*.
- The method declaration in Listing 4-2 has two parts. The first line (the part with the name `fixTheAlternator` in it, up to but not including the open curly brace) is called a *method header*. The rest of Listing 4-2 (the part surrounded by curly braces) is a *method body*.
- The term *method declaration* distinguishes the list of instructions in Listing 4-2 from the instruction in Listing 4-3, which is known as a method call.

For a handy illustration of all the method terminology, see Figure 4-5.

A method’s header and body are like an entry in a dictionary: An entry doesn’t use the word that it defines. Instead, an entry tells you what happens if and when you use the word:

> **chocolate** (choc-o-late) *n.* **1.** The most habit-forming substance on Earth. **2.** Something you pay for with money from royalties. **3.** The most important nutritional element in a person’s diet.
>
> **fixTheAlternator(onACertainCar)** Drive the car into the bay, lift the hood, get the wrench, loosen the alternator belt, and then eat some chocolate.



```
void fixTheAlternator(onACertainCar) {
    driveInto(car, bay);
    lift(hood);
    get(wrench);
    loosen(alternatorBelt);
    ...
}
```

```
fixTheAlternator(junkyOldFord);
```

FIGURE 4-5: The terminology describing methods.

In contrast, a method call is like the use of a word in a sentence. A method call sets some code in motion:

> "I want some chocolate, or I'll throw a fit."
>
> "fixTheAlternator on that junkyOldFord."

REMEMBER

A *method's declaration* tells the computer what will happen if you call the method into action. A *method call* (a separate piece of code) tells the computer to actually call the method into action. A method's declaration and the method's call tend to be in different parts of the Java program.

The main method in a program

In Listing 4-1, the bulk of the code is the declaration of a method named `main`. For now, just look for the word `main` in the code's method header. Don't worry about the other words in the method header — the words `public`, `static`, `void`, `String`, or `args`. I explain these words (on a need-to-know basis) in the next several chapters.

Like any Java method, the `main` method is a recipe:

```
How to make biscuits:
    Preheat the oven.
```

```
      Roll the dough.
      Bake the rolled dough.
```

or

```
How to follow the main instructions in the ThingsILike code:
   Display Chocolate, royalties, sleep in a text-only area.
```

The word `main` plays a special role in Java. In particular, you never write code that explicitly calls a `main` method into action. The word `main` is the name of the method that is called into action automatically when the program begins running.

When the `ThingsILike` program runs, the computer automatically finds the program's `main` method and executes any instructions inside the method's body. In the `ThingsILike` program, the `main` method's body has only one instruction. That instruction tells the computer to print `Chocolate, royalties, sleep` in a text-only area.

REMEMBER

None of the instructions in a method is executed until the method is called into action. But if you give a method the name `main`, that method is called into action automatically.

At last! Tell the computer to do something!

Buried deep in the heart of Listing 4-1 is the single line that actually issues a direct instruction to the computer. The line

```
System.out.println("Chocolate, royalties, sleep");
```

tells the computer to display `Chocolate, royalties, sleep`. (If you use IntelliJ IDEA, the computer displays `Chocolate, royalties, sleep` in the Run tool window.) I can describe this line of code in at least two different ways:

- **It's a statement.** In Java, a direct instruction that tells the computer to do something is called a *statement*. The statement in Listing 4-1 tells the computer to display some text. The statements in other programs may tell the computer to put the number 7 in a certain memory location or make a window appear on the screen. The statements in computer programs do all kinds of things.
- **It's a method call.** Earlier in this chapter, I describe something named a method call. The statement

  ```
  fixTheAlternator(junkyOldFord);
  ```

is an example of a method call, and so is

```
System.out.println("Chocolate, royalties, sleep");
```

Java has many different kinds of statements. A method call is just one kind.

Ending a statement with a semicolon

In Java, each statement ends with a semicolon. The code in Listing 4-1 has only one statement in it, so only one line in Listing 4-1 ends with a semicolon.

Take any other line in Listing 4-1 — the method header, for example. The method header (the line with the word main in it) doesn't directly tell the computer to do anything. Instead, the method header describes some action for future reference. The header announces "Just in case someone ever calls the main method, the next few lines of code tell you what to do in response to that call."

REMEMBER

Every complete Java statement ends with a semicolon. A method call is a statement, so it ends with a semicolon, but neither a method header nor a method declaration is a statement.

The method named System.out.println

The statement in the middle of Listing 4-1 calls a method named System.out.println. This method is defined in the Java API. Whenever you call the System.out.println method, the computer displays text in IntelliJ's Run tool window. (If you're not using IntelliJ IDEA, the display appears in some other text-only area.)

Think about names. Believe it or not, I know two people named Pauline Ott. One of them is a nun; the other is a physicist. Of course, there are plenty of Paulines in the English-speaking world, just as there are several things named println in the Java API. To distinguish the physicist Pauline Ott from the film critic Pauline Kael, I write the full name Pauline Ott. And to distinguish the nun from the physicist, I write "Sister Pauline Ott." In the same way, I write either System.out.println or DriverManager.println. The first (which you use often) writes text on the computer's screen. The second (which you don't use at all in this book) writes to a database log file.

Just as Pauline and Ott are names in their own right, so System, out, and println are names in the Java API. But to use println, you must write the method's full name. You never write println alone. It's always System.out.println or another combination of API names.

WARNING

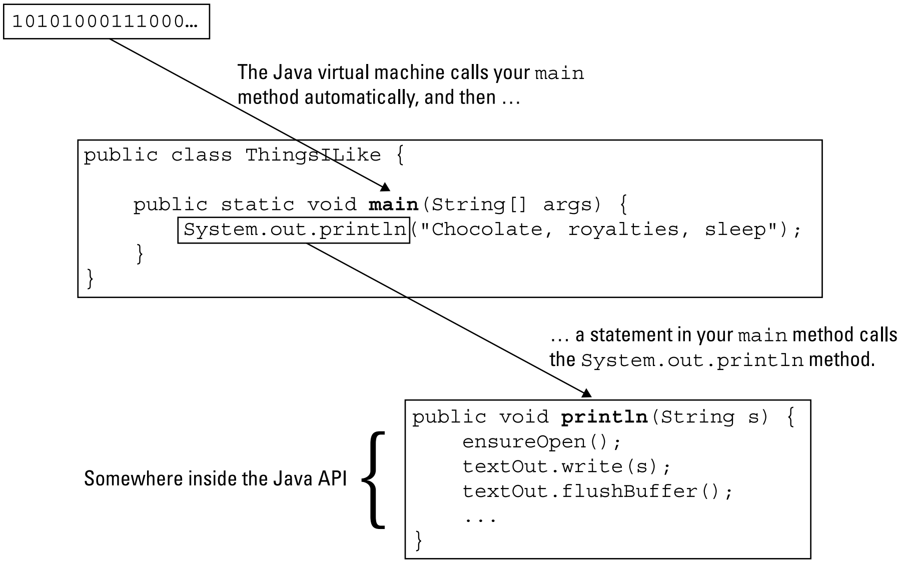
The Java programming language is cAsE-sEnSiTiVe. If you change a lowercase letter to an uppercase letter (or vice versa), you change a word's meaning. You can't replace `System.out.println` with `system.out.Println`. If you do, your program won't work.

Methods, methods everywhere

Two methods play roles in the `ThingsILike` program. Figure 4-6 illustrates the situation, and the next few bullets give you a guided tour:

- **There's a declaration for a `main` method.** I wrote the `main` method myself. This `main` method is called automatically whenever I start running the `ThingsILike` program.
- **There's a call to the `System.out.println` method.** The method call for the `System.out.println` method is the only statement in the body of the `main` method. In other words, calling the `System.out.println` method is the only thing on the `main` method's to-do list.

  The declaration for the `System.out.println` method is buried inside the official Java API. For a refresher on the Java API, refer to Chapter 1.



```
10101000111000...
```

The Java virtual machine calls your `main` method automatically, and then ...

```
public class ThingsILike {

    public static void main(String[] args) {
        System.out.println("Chocolate, royalties, sleep");
    }
}
```

... a statement in your `main` method calls the `System.out.println` method.

Somewhere inside the Java API

```
public void println(String s) {
    ensureOpen();
    textOut.write(s);
    textOut.flushBuffer();
    ...
}
```

FIGURE 4-6: Calling the `System.out.println` method.

TECHNICAL STUFF

When I say things like "`System.out.println` is buried inside the API," I'm not doing justice to the API. True, you can ignore all the nitty-gritty Java code inside the API. All you need to remember is that `System.out.println` is defined somewhere inside that code. But I'm not being fair when I make the API code sound like something magical. The API is just another bunch of Java code. The statements in the API that tell the computer what it means to carry out a call to `System.out.println` look a lot like the Java code in Listing 4-1.

The Java class

Have you heard the term *object-oriented programming* (also known as *OOP*)? *OOP* is a way of thinking about computer programming problems — a way that's supported by several different programming languages. OOP started in the 1960s with a language called Simula. It was reinforced in the 1970s with another language, named Smalltalk. In the 1980s, OOP took off big-time with the language C++.

Some people want to change the acronym and call it COP — class-oriented programming. That's because object-oriented programming begins with something called a *class*. In Java, everything starts with classes, everything is enclosed in classes, and everything is based on classes. You can't do anything in Java until you've created a class of some kind. It's like being on *Jeopardy!* and saying "Let's make it a daily double." Then the host says, "I'm sorry. You must phrase your instruction inside a Java class."

It's important for you to understand what a class is, so I dare not give a haphazard explanation in this chapter. Instead, I devote much of Chapter 13 to the question "What is a class?" Anyway, in Java, your `main` method has to be inside a class. I wrote the code in Listing 4-1, so I got to make up a name for my new class. I chose the name `ThingsILike`, so the code in Listing 4-1 starts with the words `public class ThingsILike`.

Take another look at Listing 4-1 and notice what happens after the line `public class ThingsILike`. The rest of the code is enclosed in curly braces. These braces mark all the stuff inside the class. Without these braces, you'd know where the declaration of the `ThingsILike` class starts, but you wouldn't know where the declaration ends.

It's as though the stuff inside the `ThingsILike` class is in a box. (Refer to Figure 4-3.) To box off a chunk of code, you do two things:

- **You use curly braces.** These curly braces tell the compiler where a chunk of code begins and ends.

- **You indent code.** Indentation tells your human eye (and the eyes of other programmers) where a chunk of code begins and ends.

Don't forget. You have to do both.

TRY IT OUT

THE WORDS IN A PROGRAM

Listing 4-1 contains several kinds of words. Find out what happens when you change some of these words:

- **Change one keyword.** For example, change the word `class` to the word `bologna`. Look for an error message in IntelliJ's editor.
- **Change the word `args` to the word `malarkey`.** After doing so, can your program still run?
- Change the word `ThingsILike` to the word `MyFavorites`. After doing so, can your program still run?

  The words `ThingsILike` and `MyFavorites` are both words that you or I can make up. So why doesn't IntelliJ like `MyFavorites`?

  The answer is, the name in the editor doesn't match the name in the Project tool window. Right-click the word `ThingsILike` in the Project tool window. From the resulting context menu, choose Refactor ➪ Rename. When a dialog box appears, type **`MyFavorites`** in the box's text field. After dismissing the dialog box, the two names match and IntelliJ is happy again.
- Change an identifier that has an agreed-upon meaning. For example, change `println` to `display`. Look for an error message in IntelliJ's editor.
- Change the program's punctuation. For example, remove a pair of curly braces. Look for an error message in IntelliJ's editor.
- Comment out the entire `System.out.println("Chocolate, royalties, sleep");` line. (Use the end-of-line commenting style.) What happens when you run the program?
- Comment out the entire `System.out.println("Chocolate, royalties, sleep");` line. (Use the traditional commenting style.) What happens when you run the program?

VALID IDENTIFIERS

There are limits to the kinds of names you can make up. For example, a person's name might include a dash, but it can't include a question mark. (At least it can't where I come from.) A well-known celebrity can get away with adopting a name that's an unpronounceable symbol. But for most of us, plain old letters, dashes, and hyphens are all we can use.

What kinds of names can you make up as part of a Java program? Find out by changing the word `args` to these other words in IntelliJ's editor. Which of the changes are okay and which are not?

- `helloThere`
- `hello_there`
- `args7`
- `ar7gs`
- `75`
- `7args`
- `hello there`
- `hello-there`
- `public`
- `royalties`
- `@args`
- `#args`
- `/args`

YOUR FAVORITE THINGS

Change the code in Listing 4-1 so that it displays things that *you* like. Run the program to make sure that it displays these things in IntelliJ's Run tool window.

IN THIS CHAPTER

- » Reading input from the keyboard
- » Editing a program
- » Shooting at trouble

Chapter 5
Composing a Program

Just yesterday, I was chatting with my servant, RoboJeeves. (RoboJeeves is an upscale model in the RJ-3000 line of personal robotic life-forms.) Here's how the discussion went:

Me: RoboJeeves, tell me the velocity of an object after it's been falling for three seconds in a vacuum.

RoboJeeves: All right, I will. "The velocity of an object after it's been falling for three seconds in a vacuum." There, I told it to you.

Me: RoboJeeves, don't give me that smart-alecky answer. I want a number. I want the actual velocity.

RoboJeeves: Okay! "A number; the actual velocity."

Me: RJ, these cheap jokes are beneath your dignity. Can you or can't you tell me the answer to my question?

RoboJeeves: Yes.

Me: "Yes" what?

RoboJeeves: Yes, I either can or can't tell you the answer to your question.

Me: Well, which is it? Can you?

RoboJeeves: Yes, I can.

Me: Then do it. Tell me the answer.

> **RoboJeeves:** The velocity is 153,984,792 miles per hour.
>
> **Me (after pausing to think):** RJ, I know you never make a mistake, but that number — 153,984,792 — is much too high.
>
> **RoboJeeves:** Too high? That's impossible. Things fall very quickly on the giant planet Mangorrrrkthongo. Now, if you wanted to know about objects falling to Earth, you should have said so in the first place.

Sometimes that robot rubs me the wrong way. The truth is, RoboJeeves does whatever I tell him to do — nothing more and nothing less. If I say, "Feed the cat," then RJ says, "Feed it to whom? Which of your guests will be having cat for dinner?"

Computers Are Stupid

Handy as they are, all computers do the same darn thing: They do *exactly* what you tell them to do, and that's sometimes very unfortunate. For example, in 1962, a *Mariner* spacecraft to Venus was destroyed just four minutes after its launch. Why? It was destroyed because of a missing keystroke in a FORTRAN program. Around the same time, NASA scientists caught an error that could have trashed the Mercury space flights. (Yup — these were flights with people on board!) The error was a line with a period instead of a comma. (A computer programmer wrote `DO 10 I=1.10` instead of `DO 10 I=1,10`.)

With all due respect to my buddy RoboJeeves, he and his computer cousins are all incredibly stupid. Sometimes they look as though they're second-guessing us humans, but actually they're just doing what other humans told them to do. They can toss virtual coins and use elaborate schemes to mimic creative behavior, but they never really think on their own. If you say, "Jump," they do what they're programmed to do in response to the letters j-u-m-p.

So, when you write a computer program, you have to imagine that a genie has granted you three wishes. Don't ask for eternal love because, if you do, the genie will give you a slobbering, adoring mate — someone you don't like at all. And don't ask for a million dollars unless you want the genie to turn you into a bank robber.

Everything you write in a computer program has to be *precise.* Take a look at an example next.

Building an Echo Chamber

Listing 5-1 contains a small Java program. The program lets you type one line of characters on the keyboard. As soon as you press Enter, the program displays a second line that copies whatever you typed.

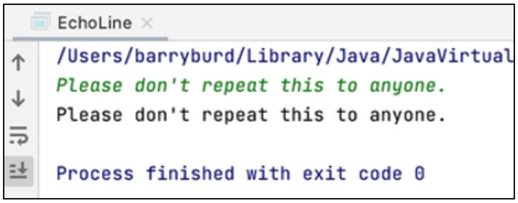
LISTING 5-1: A Java Program

```
import java.util.Scanner;

public class EchoLine {

    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.println(keyboard.nextLine());

        keyboard.close();
    }
}
```

REMEMBER

Most of the programs in this book are command line programs. When you run one of these programs, the input and output appear in IntelliJ's Run tool window. You can see GUI versions of the program in Listing 5-1 — and in many other examples from this book — by visiting the book's website (`http://beginprog.allmycode.com`).

Figure 5-1 shows a run of the `EchoLine` code (the code in Listing 5-1). The text in the figure is a mixture of my own typing and the computer's responses.



```
/Users/barryburd/Library/Java/JavaVirtual
Please don't repeat this to anyone.
Please don't repeat this to anyone.

Process finished with exit code 0
```

FIGURE 5-1: What part of the word *don't* do you not understand?

In Figure 5-1, I type the first line (the first `Please don't repeat this to anyone` line), and the computer displays the second line. Here's what happens when you run the code in Listing 5-1:

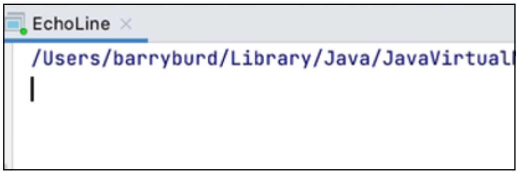
1. **At first, the computer does nothing.**

   The computer is waiting for you to type something.

2. **You click inside the Run tool window.**

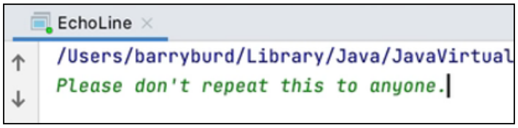
   As a result, you see the cursor on the left edge of the Run tool window, as shown in Figure 5-2.

3. **You type one line of text — any text at all. (See Figure 5-3.)**
4. **You press Enter, and the computer displays another copy of the line that you typed, as shown in Figure 5-4.**



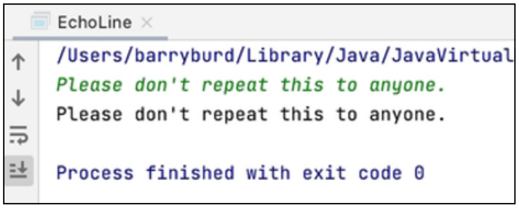
```
/Users/barryburd/Library/Java/JavaVirtual
```

FIGURE 5-2: The computer waits for you to type something.



```
/Users/barryburd/Library/Java/JavaVirtual
Please don't repeat this to anyone.
```

FIGURE 5-3: You type a sentence.



```
/Users/barryburd/Library/Java/JavaVirtual
Please don't repeat this to anyone.
Please don't repeat this to anyone.

Process finished with exit code 0
```

FIGURE 5-4: The computer echoes your input.

After a copy of your input is displayed, the program's run comes to an end.

Typing and running a program

This book's website (`http://beginprog.allmycode.com`) has a link for downloading all the book's Java programs. After you download the programs, you can follow the instructions in Chapter 3 to add a listing's code to IntelliJ IDEA.

But instead of running the ready-made code, I encourage you to start from scratch — to type Listing 5-1 yourself and then to test your newly created code. Just follow these steps:

1. **Use IntelliJ IDEA to create a new Java project.**

   Follow the steps in Chapter 3. As you march through these steps, put a check mark in the Create Project from Template check box and make sure that the Command Line App item is selected.

   When you've finished creating the new project, IntelliJ shows you a main window. The window's editor contains the contents of a newly created `Main.java` file.

2. **Change the name of the `Main.java` file.**

   To do so, right-click the word `Main` in the Project tool window. In the resulting context menu, choose Refactor ➪ Rename. When a dialog box appears, type `EchoLine` in the box's text field.

   This step gives your code the same name as the one in Listing 5-1.

   WARNING

   When you choose Refactor ➪ Rename, IntelliJ changes two things. First, the name of the file on your hard drive changes from `Main.java` to `EchoLine.java`. Also, the second line of code in the file changes from

   ```
   public class Main {
   ```

   to

   ```
   public class EchoLine {
   ```

   Don't replace `public class Main` with `public class EchoLine` by typing in the editor. Doing so wouldn't change the `Main.java` file's name, and that would make Java very angry. (You don't want to make Java angry. Do you?)

   At this point, your code looks like the code in Listing 5-1, but several lines are missing.

3. **In IntelliJ's editor, type the missing lines from Listing 5-1.**

   Copy the code exactly as you see it in Listing 5-1. When you're finished typing, check the spelling, the capitalization, and the punctuation.

   TIP

   IntelliJ IDEA has hundreds of tricks up its virtual sleeve. Many of these tricks make typing a breeze. For more information, see the later section entitled "Make IntelliJ Do All the Work."

   If you typed everything correctly, you don't see any error markers in the editor.

   If you see error markers, go back and compare everything you typed with the stuff in Listing 5-1. Compare every letter, every word, every squiggle, every smudge.

4. **Make any changes or corrections to the code in the editor.**

   When at last you see no error markers, you're ready to run the program.

5. **Right-click either the** `EchoLine` **branch in the Project tool window or the** `EchoLine.java` **tab at the top of the editor.**

6. **In the resulting context menu, select Run 'EchoLine.main()'.**

   Your new Java program runs, but nothing much happens.

7. **Click inside IntelliJ's Run tool window.**

   As a result, the cursor sits on the left edge of the Run tool window. (Refer to Figure 5-2.) The computer is waiting for you to type something.

   WARNING

   If you forget to click inside the Run tool window, IntelliJ may not send your keystrokes to the running Java program. Instead, IntelliJ may send your keystrokes to the editor or to some other part of the project's main window.

8. **Type a line of text and then press Enter.**

   In response, the computer displays a second copy of your line of text. Then the program's run comes to an end. (Refer to Figure 5-4.)

If this list of steps seems a bit sketchy, you can find much more detail in Chapter 3. (Look first at the section in Chapter 3 about typing and running your own code.) For the most part, the steps here are a quick summary of the material in Chapter 3.

So, what's the big deal when you type the program yourself? Well, lots of interesting things can happen when you apply fingers to keyboard. That's why the second half of this chapter is devoted to troubleshooting.

How the EchoLine program works

When you were a tiny newborn, resting comfortably in your mother's arms, she told you how to send characters to the computer screen:

```
System.out.println(whatever text you want displayed);
```

What she didn't tell you was how to fetch characters from the computer keyboard. There are lots of ways to do it, but the one I recommend in this chapter is

```
keyboard.nextLine()
```

Now, here's the fun part. Calling the `nextLine` method doesn't just scoop characters from the keyboard. When the computer runs your program, the computer *substitutes whatever you type on the keyboard* in place of the text `keyboard.nextLine()`.

To understand this, look at the statement in Listing 5-1:

```
System.out.println(keyboard.nextLine());
```

When you run the program, the computer sees your call to `nextLine` and stops dead in its tracks. (Refer to Figure 5-2.) The computer waits for you to type a line of text. So (refer to Figure 5-3) you type this line:

```
Hey, there's an echo in here.
```

The computer substitutes this entire `Hey` line for the `keyboard.nextLine()` call in your program. The process is illustrated in Figure 5-5.

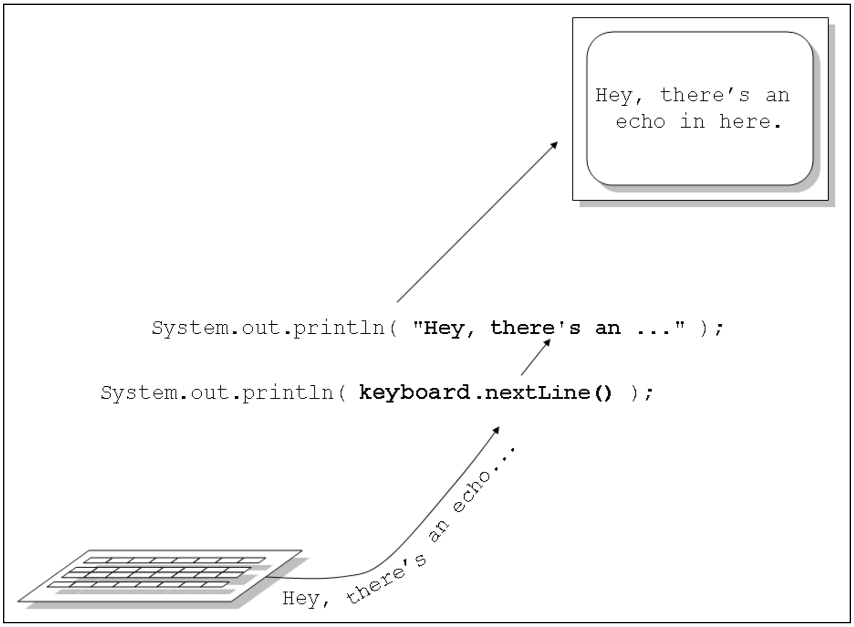

Hey, there’s an echo in here.
System.out.println("Hey, there's an ...");
System.out.println(keyboard.nextLine());
Hey, there's an echo...

FIGURE 5-5: The computer substitutes text in place of the `nextLine` call.

The call to `keyboard.nextLine()` is nestled inside the `System.out.println` call. So, when all is said and done, the computer behaves as though the statement in Listing 5-1 looks like this:

```
System.out.println("Hey, there's an echo in here.");
```

The computer displays another copy of the text `Hey, there's an echo in here.` on the screen. That’s why you see two copies of the `Hey` line in Figure 5-5.

Getting numbers, words, and other things

In Listing 5-1, the words `keyboard.nextLine()` get an entire line of text from the computer keyboard. If you type

```
Testing 1 2 3
```

the program in Listing 5-1 echoes back the entire `Testing 1 2 3` line of text.

Sometimes you don't want a program to get an entire line of text. Instead, you want the program to get a piece of a line. For example, when you type **1 2 3**, you may want the computer to get the number 1. (Maybe the number 1 stands for one customer or something like that.) In such situations, you don't put `keyboard.nextLine()` in your program. Instead, you use `keyboard.`**`nextInt()`**.

Table 5-1 shows you a few variations on the `keyboard.next` business. Unfortunately, the table's entries aren't very predictable. To read a line of input, you call `nextLine`. But to read a word of input, you don't call `nextWord`. (The Java API has no `nextWord` method.) Instead, to read a word, you call `next`.

TABLE 5-1: Some Scanner Methods

To Read This	Make This Method Call
A number with no decimal point in it	`nextInt()`
A number with a decimal point in it	`nextDouble()`
A word (ending in a blank space, for example)	`next()`
A line (or what remains of a line after you've already read some data from the line)	`nextLine()`
A single character (such as a letter, digit, or punctuation character)	`findWithinHorizon(".",0).charAt(0)`

Also, the table's story has a surprise ending. To read a single character, you don't call `next`*`Something`*. Instead, you can call the bizarre `findWithinHorizon(".",0).charAt(0)` combination of methods. (You'll have to excuse the folks who created the `Scanner` class. They created `Scanner` from a specialized point of view.)

To see some of the table's methods in action, check other program listings in this book. Chapters 6, 7, and 8 have some particularly nice examples.

TRY IT OUT

GETTING SINGLE WORDS AND ENTIRE LINES OF TEXT

Follow the instructions in this chapter's earlier section "Typing and running a program," but make these two changes:

- In Step 2, change `Main.java` to `GetInput.java`.
- Rather than type the code in Listing 5-1, type the following program:

  ```
  import java.util.Scanner;

  public class GetInput {

      public static void main(String[] args) {
          Scanner keyboard = new Scanner(System.in);

          System.out.println(keyboard.next());
          System.out.println(keyboard.next());
          System.out.println(keyboard.nextLine());

          keyboard.close();
      }
  }
  ```

When the program runs, type the following text (all on one line) in IntelliJ's Run tool window, and then press Enter. How does the computer respond? Why?

```
I enjoy learning Java.
```

TYPE THREE LINES OF CODE AND DON'T LOOK BACK

Buried innocently inside Listing 5-1 are three extra lines of code. These lines help the computer read input from the keyboard. The three lines are

```
import java.util.Scanner;

    Scanner keyboard = new Scanner(System.in);

    keyboard.close();
```

Concerning these three lines, I have bad news and good news:

- **The bad news is, the reasoning behind these lines is difficult to understand.** That's especially true here in Chapter 5, where I introduce Java's most fundamental concepts.

- **The good news is, you don't have to understand the reasoning behind these three lines.** You can copy-and-paste these lines into any program that gets input from the keyboard. You don't have to change the lines in any way. These lines work with no modifications in all kinds of Java programs.

Just be sure to put these lines in the right places:

- Make the `import java.util.Scanner` line the first line in your program.
- Put the `Scanner keyboard = new Scanner(System.in)` line inside the `main` method immediately after the `public static void main(String[] args) {` line.
- Make the `keyboard.close()` line the last full line in your program. Put it immediately before the two lines with close curly braces (`}`).

At some point in the future, you may have to be more careful about the positioning of these three lines. But for now, the rules I give will serve you well.

A QUICK LOOK AT THE SCANNER

In this chapter, I advise you to ignore any of the meanings behind the lines `import java.util.Scanner` and `Scanner keyboard`. Just paste these two lines mindlessly in your code and then move on.

Of course, you may not want to take my advice. You may not like ignoring things in your code. If you happen to be such a stubborn person, I have a few quick facts for you:

- **The word `Scanner` is defined in the Java API.**

  A `Scanner` is something you can use for getting input.

- **The words `System` and `in` are defined in the Java API.**

  Taken together, the words `System.in` stand for the computer keyboard.

  In later chapters, you see things like `new Scanner(new File("myData.txt"))`. In those chapters, I replace `System.in` with the words `new File("myData.txt")` because I'm not getting input from the keyboard. Instead, I'm getting input from a file on the computer's hard drive.

- **The word `keyboard` doesn't come from the Java API.**

  The word `keyboard` is a Barry Burd creation. Instead of `keyboard`, you can use `readingThingie` (or any other name you want to use as long as you use the name consistently). So, if you want to be creative, you can write

```
Scanner readingThingie = new Scanner(System.in);

System.out.println(readingThingie.nextLine());
```

The revised Listing 5-1 (with `readingThingie` instead of `keyboard`) compiles and runs without a hitch.

- **The line `import java.util.Scanner` is an example of an *import declaration*.**

  The optional import declaration allows you to abbreviate names in the rest of your program. You can remove the import declaration from Listing 5-1. But if you do, you must use the `Scanner` class's *fully qualified name* throughout your code. Here's how:

```
public class EchoLine {

  public static void main(String[] args) {
    java.util.Scanner keyboard = new java.util.Scanner (System.in);

    System.out.println(keyboard.nextLine());

    keyboard.close();
  }
}
```

Make IntelliJ Do All the Work

You may have noticed things popping into view while you type code in IntelliJ's editor. The editor has a boatload of features to ease your typing burden. For a look at some of these features, follow these steps:

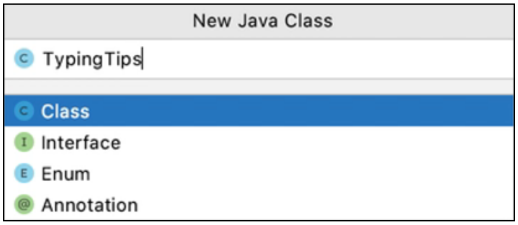
1. **Expand the tree of IntelliJ's Project tool window and look for a branch named `src`.**

   The `src` branch contains your project's Java code files.

2. **Right-click the `src` branch. On the resulting context menu, choose New ➪ Java Class.**

   When you do, a small dialog box asks you for a new class's name.

3. **In Figure 5-6, I name the class `TypingTips`.**

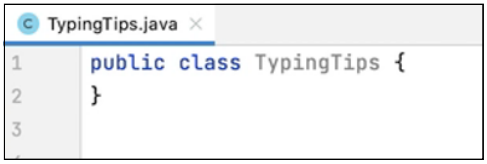


TypingTips

FIGURE 5-6: Naming the class.

4. **To dismiss the dialog box, press Enter.**

   As a result, IntelliJ's editor shows you an empty `TypingTips` class. (See Figure 5-7.)

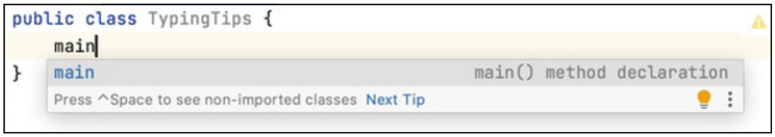


```
public class TypingTips {
}
```

FIGURE 5-7: IntelliJ is happy to oblige.

5. **In the editor, add a blank line between the open and close curly braces.**
6. **On the new blank line, start typing the word `main`.**

   As you type, IntelliJ offers to create a `main()` method declaration. (See Figure 5-8.)



```
public class TypingTips {
    main
}
```

main
main() method declaration
Press ^Space to see non-imported classes Next Tip

FIGURE 5-8: IntelliJ reads your mind.

7. **Accept IntelliJ's offer by pressing Enter.**

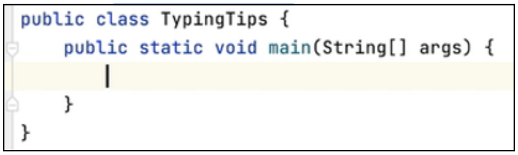
   A `main` method appears in the IntelliJ editor. IntelliJ positions its cursor on a line between the method's open and close curly braces. (See Figure 5-9.)

8. **Start typing the word `Scanner`.**

   IntelliJ offers to finish the typing for you. (See Figure 5-10.)

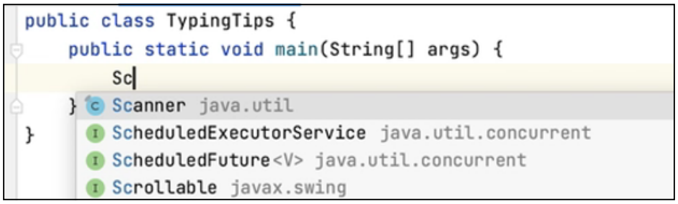
9. **Accept IntelliJ's offer by pressing Enter.**

   IntelliJ completes your typing of the word `Scanner`, but it also adds a `Scanner` import declaration. (For a few words about this import declaration, refer to the earlier sidebar entitled "A quick look at the scanner.")



```
public class TypingTips {
    public static void main(String[] args) {
        |
    }
}
```

FIGURE 5-9: IntelliJ places the cursor exactly where you want it.



```
public class TypingTips {
    public static void main(String[] args) {
        Sc
    } Scanner java.util
}     ScheduledExecutorService java.util.concurrent
      ScheduledFuture<V> java.util.concurrent
      Scrollable javax.swing
```

FIGURE 5-10: It's as if you and IntelliJ are telepathic twins.

10. **Continue to type this line from Listing 5-1:**

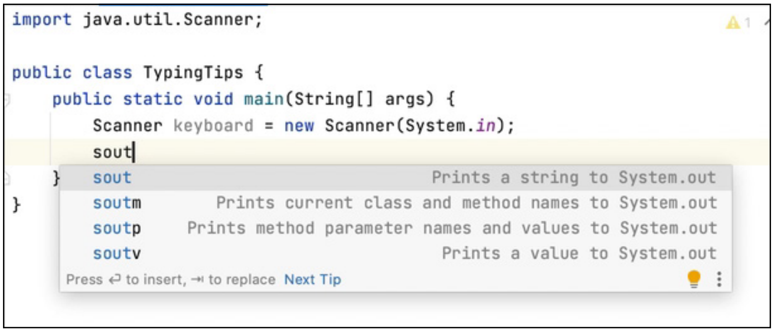
```
Scanner keyboard = new Scanner(System.in);
```

and please type slowly.

As you type, notice the places where IntelliJ offers to complete each of your words. When the whole Scanner *blah–blah* line appears in the editor, you're ready to move on.

11. **On the next line, type** sout **and then press Enter. (See Figure 5-11.)**

The word sout is IntelliJ's own abbreviation for System.out.println(). IntelliJ inserts those characters into your code and positions the cursor between the open and close parentheses.



```
import java.util.Scanner;

public class TypingTips {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        sout
    }   sout                        Prints a string to System.out
}       soutm      Prints current class and method names to System.out
        soutp    Prints method parameter names and values to System.out
        soutv                        Prints a value to System.out
      Press ⏎ to insert, ⇥ to replace  Next Tip
```

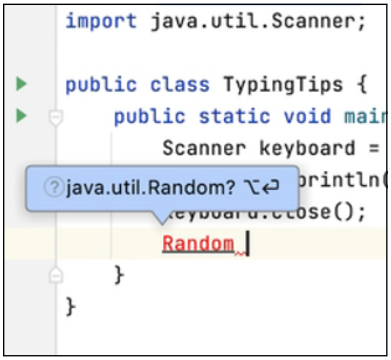
FIGURE 5-11: How to say "something" in Scotland.

12. **Continue typing the code in Listing 5-1. Type slowly so that you can see all the places where IntelliJ offers to write the code for you.**

Are you finished typing the lines in Listing 5-1? If so, try this:

13. **On a brand-new line of code, type the word** Random **followed by a blank space.**

IntelliJ displays a helpful hint because the word Random would benefit from the presence of an import declaration. (See Figure 5-12.)



```
import java.util.Scanner;

public class TypingTips {
    public static void mai
        Scanner keyboard =
                          println(
        keyboard.close();
        Random |
    }
}
```

FIGURE 5-12: "Do you mean `java.util.Random`?" says IntelliJ.

14. **Press Alt+Enter (also known as Option-Enter on a Mac).**

As a result, IntelliJ adds `import java.util.`Random to the top of your program.

15. **Delete the word** Random **that you typed in Step 13.**

With this word deleted, the line `import java.util.`Random no longer serves a purpose, so IntelliJ displays `import java.util.`Random in a light grey shade.

16. **On IntelliJ's main menu bar, choose Code ➪ Optimize Imports.**

When you do, IntelliJ deletes the line `import java.util.`Random. No muss, no fuss.

Expecting the Unexpected

Not long ago, I met an instructor with an interesting policy. He said, "Sometimes when I'm lecturing, I compose a program from scratch on the computer. I do it right in front of my students. If the program compiles and runs correctly on the first try, I expect the students to give me a big round of applause."

At first, you may think this guy has an enormous ego, but you have to put things in perspective. It's unusual for a program to compile and run correctly the first time. There's almost always a typo or another error of some kind.

This section deals with the normal, expected errors that you see when you compile and run a program for the first time. Everyone makes these mistakes, even the most seasoned travelers. The key is keeping a cool head. Here's my general advice:

- **Don't expect a program that you type to compile the first time.**

  Be prepared to return to your editor and fix some mistakes.

- **Don't expect a program that compiles flawlessly to run correctly.**

  Even with no error markers in IntelliJ's editor, your program might still contain flaws. After IntelliJ compiles your program, you still have to run it successfully. That is, your program should finish its run and display the correct output.

  You compile, and then you run. Getting a program to compile without errors is the easier of the two tasks.

- **Read what's in the IntelliJ editor, not what you assume is in the IntelliJ editor.**

  Don't assume that you've typed words correctly, that you've capitalized words correctly, or that you've matched curly braces or parentheses correctly. Compare the code you typed with any sample code that you have. Make sure that every detail is in order.

- **Be patient.**

  Every good programming effort takes a long time to get right. If you don't understand something right away, be persistent. Stick with it (or put it away for a while and come back to it). There's nothing you can't understand if you put in enough time.

- **Don't become frustrated.**

  Don't throw your pie crust. Frustration (not lack of knowledge) is your enemy. If you're frustrated, you can't accomplish anything.

- **Don't think you're the only person who's slow to understand.**

  I'm slow, and I'm proud of it. (Becky, Chapter 6 will be two weeks late.)

- **Don't be timid.**

  If your code isn't working and you can't figure out why it's not working, ask someone. Post a message in an online forum. And don't be afraid of anyone's snide or sarcastic answer. (For a list of gestures you can make in response to peoples' snotty answers, see Appendix Z.)

  To ask me directly, send me an email message, tweet me, or post to me on Facebook. (Send email to `BeginProg@allmycode.com`, tweets to `@allmycode`, or posts to Facebook at `/allmycode`.)

Diagnosing a problem

The "Typing and running a program" section, earlier in this chapter, tells you how to run the `EchoLine` program. If all goes well, your screen ends up looking like the one shown in Figure 5-1. But things don't always go well. Sometimes your finger slips, inserting a typo into your program. Sometimes you ignore one of the details in Listing 5-1 and you get a nasty error message.

Of course, some things in Listing 5-1 are okay to change. Not every word in Listing 5-1 is cast in stone. Here's a nasty wrinkle: I can't tell you that you must always retype Listing 5-1 exactly as it appears. Some changes are okay; others are not. Keep reading for some "f'rinstances."

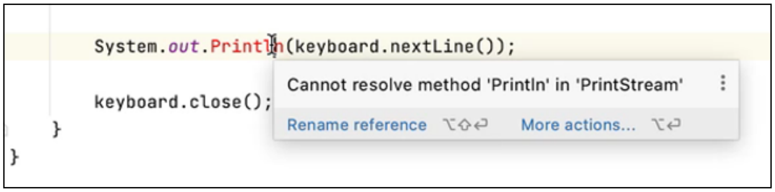
Case sensitivity

Java is case-sensitive. Among other things, *case-sensitive* means that, in a Java program, the letter `P` isn't the same as the letter `p`. If you send me some email and start with "Hi barry" instead of "Hi Barry," I still know what you mean. But Java doesn't work that way.

Change just one character in a Java program and, instead of an uneventful compilation, you get a big headache! Change `p` to `P`, like so:

```
//The following line is incorrect:
System.out.Println(keyboard.nextLine());
```

When you type the program in IntelliJ's editor, you get the ugliness shown in Figure 5-13.



```
        System.out.Println(keyboard.nextLine());

        keyboard.close();
    }
}
```

Cannot resolve method 'Println' in 'PrintStream'

FIGURE 5-13: The Java compiler understands `println`, but not `Println`.

When you see error markers like the ones in Figure 5-13, your best bet is to stay calm and read the messages carefully. Sometimes the messages contain useful hints. (Of course, sometimes they don't.) The message in Figure 5-13 is `Cannot resolve method 'Println' in 'PrintStream'`. In plain English, this means, "The Java compiler can't interpret the word `Println`." (The message stops short of saying, "Don't type the word `Println`, you dummy!" In any case, if the computer says you're one of us dummies, you should take it as a compliment.) Now, for plenty of reasons the compiler may not be able to understand a word like

Println. But, for a beginning programmer, you should check two important things right away:

- **Have you spelled the word correctly?**

  Did you accidentally type print1n with the digit 1 instead of println with the lowercase letter l?

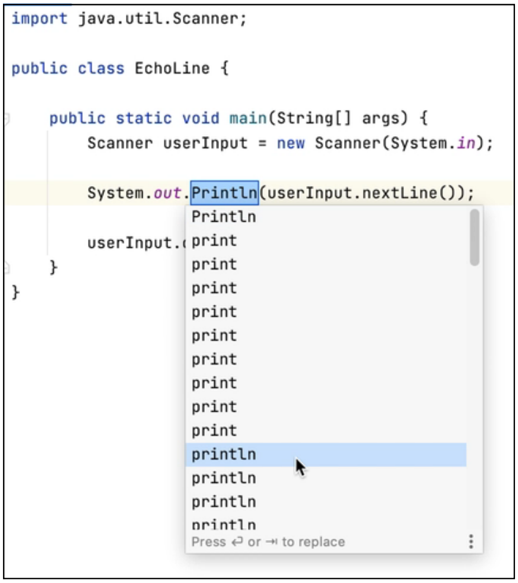
- **Have you capitalized all letters correctly?**

  Did you incorrectly type Println or PrintLn instead of println?

Either of these errors can send the Java compiler into a tailspin. So compare your typing with the approved typing word-for-word (and letter-for-letter). When you find a discrepancy, go back to the editor and fix the problem. Then try compiling the program again.

When in doubt, you can ask IntelliJ for some help. Look again at Figure 5-13 and notice the words Rename reference and More actions in the pop-up box. These are called *quick fixes*. You can hide them or make them reappear by clicking the three-dot icon by the right edge of the pop-up.

If you click the first quick fix (the Rename reference link), IntelliJ offers several suggestions. (See Figure 5-14.) The list of suggestions may contain several println items. That's okay. If you select any of these println items, IntelliJ fixes your code.



```
import java.util.Scanner;

public class EchoLine {

    public static void main(String[] args) {
        Scanner userInput = new Scanner(System.in);

        System.out.Println(userInput.nextLine());

        userInput.(
    }
}
```

FIGURE 5-14: What should we name the baby?

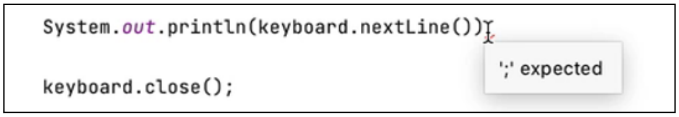
Not enough punctuation

In English and in Java, using the; proper! punctuation is important)

Take, for example, the semicolons in Listing 5-1. What happens if you forget to type a semicolon?

```
//The following code is incorrect:

System.out.println(keyboard.nextLine())

keyboard.close();
```

If you leave off the semicolon, you see the message shown in Figure 5-15.



```
System.out.println(keyboard.nextLine())

keyboard.close();
```

';' expected

FIGURE 5-15: A helpful error message.

A message like the one in Figure 5-15 makes your life much simpler. I don't have to explain the message, and you don't have to puzzle over the message's meaning. Just take the message `';' expected` at its face value. Insert the semicolon between the end of the `System.out.println(keyboard.nextLine())` statement and whatever code comes after the statement. (For code that's easier to read and understand, tack on the semicolon at the end of the `System.out.println(keyboard.nextLine())` statement.)

Too much punctuation

In junior high school, my English teacher said I should use a comma whenever I would normally pause for a breath. This advice doesn't work well during allergy season, when my sentences have two commas for every three or four words. Even as a paid author, I have trouble deciding where the commas should go, so I often add extra commas for good measure. This makes more work for my copy editor, Becky, who has a trash can full of commas by the desk in her office.

It's the same way in a Java program. You can get carried away with punctuation. Consider, for example, the `main` method header in Listing 5-1. This line is a dangerous curve for novice programmers.

For information on the terms *method header* and *method body*, refer to Chapter 4.

WHY CAN'T THE COMPUTER FIX IT?

How often do you get to finish someone else's sentence? "Please," says your supervisor, "go over there and connect the —"

"Wires," you say. "I'll connect the wires." If you know what someone means to say, why wait for them to say it?

This same question comes up in connection with computer error messages. Take a look at the message in Figure 5-15. The computer expects a semicolon after the statement on Line 8. Well, Mr. Computer, if you know where you want a semicolon, just add the semicolon and be done with it. Why are you bothering me about it?

The answer is simple: The computer isn't interested in taking any chances. What if you *don't* really want a semicolon after the statement on Line 8? What if the missing semicolon represents a more profound problem? If the computer added the extra semicolon, it could potentially do more harm than good.

Returning to you and your supervisor:

"Please," says your supervisor, "go over there and connect the —"

"Wires," you say. "I'll connect the wires."

Boom! A big explosion. "Not the wires, you dummy. The dots. I wanted you to connect the dots."

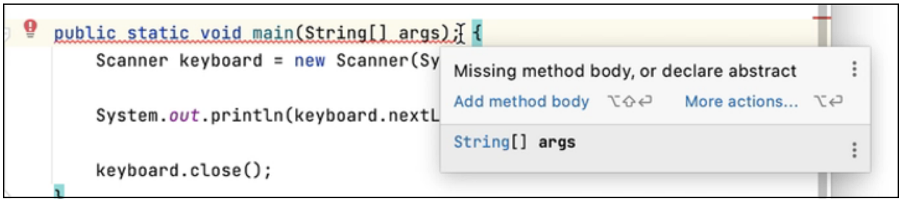
"Sorry," you say.

Normally, you shouldn't end a method header with a semicolon. But people add semicolons anyway. Maybe, in some subtle way, a method header looks like it should end with a semicolon:

```
//The following line is incorrect:
public static void main(String[] args); {
```

If you add this extraneous semicolon to the code in Listing 5-1, you get the message shown in Figure 5-16.

The error message in Figure 5-16 is a bit misleading. The message says `Missing method body, or declare abstract`. But the method has a body. Doesn't it?



```
public static void main(String[] args); {
    Scanner keyboard = new Scanner(Sy

    System.out.println(keyboard.nextL

    keyboard.close();
```

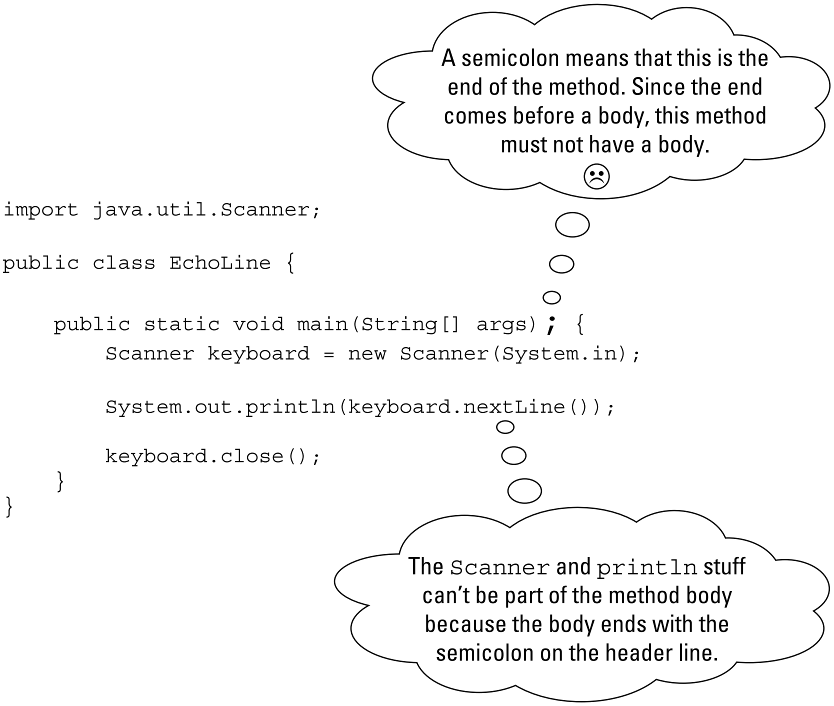
Missing method body, or declare abstract

Add method body ⌥⇧↩ More actions... ⌥↩

String[] args

FIGURE 5-16: An unwanted semicolon messes things up.

When the computer tries to compile `public static void main(String[] args);` (ending with a semicolon), the computer gets confused. I illustrate the confusion in Figure 5-17. Your eye sees an extra semicolon, but the computer's eye interprets this as a method without a body. So that's the error message — the computer says, "This method requires a body instead of a semicolon."



A semicolon means that this is the end of the method. Since the end comes before a body, this method must not have a body. ☹

```
import java.util.Scanner;

public class EchoLine {

    public static void main(String[] args) ; {
        Scanner keyboard = new Scanner(System.in);

        System.out.println(keyboard.nextLine());

        keyboard.close();
    }
}
```

The Scanner and println stuff can't be part of the method body because the body ends with the semicolon on the header line.

FIGURE 5-17: What's on this computer's mind?

If you select the `Add method body` quick fix, IntelliJ creates the following (really horrible) code:

```
import java.util.Scanner;


public class EchoLine {
    public static void main(String[] args) {

    }

    {
        Scanner keyboard = new Scanner(System.in);

        System.out.println(keyboard.nextLine());

        keyboard.close();
    }
}
```

This "fixed" code has no compile-time errors. But when you run this code, nothing happens. The program starts running and then stops running with `Process finished` in the Run tool window.

We all know that a computer is a very patient, very sympathetic machine. That's why it looks at your code and decides to give you one more chance. The computer remembers that Java has an advanced feature in which you write a method header without writing a method body. When you do this, you get what's called an *abstract method* — something that I don't use in this book. Anyway, in Figure 5-17, the computer sees a header with no body. So the computer says to itself, "I know! Maybe the programmer is trying to write an abstract method. The trouble is, an abstract method's header has to have the word `abstract` in it. I should remind the programmer about that." So the computer offers the `declare abstract` quick fix in Figure 5-16.

One way or another, you can't interpret the error message and the quick fixes in Figure 5-9 without reading between the lines. So here are some tips to help you decipher murky messages:

- **Avoid the knee-jerk response.**

  Some people see the `declare abstract` suggestion in Figure 5-16 and wonder how to make the code be `abstract`. Unfortunately, this isn't the right approach. If you don't know what `abstract` means, chances are that you didn't mean to make anything be `abstract` in the first place.

- **Stare at the bad line of code for a long, long time.**

  If you look carefully at the `public static` line in Figure 5-16, eventually you'll notice that it's different from the corresponding line in Listing 5-1. The line in Listing 5-1 has no semicolon, but the line in Figure 5-16 has one.

Of course, you won't always start with some prewritten code like the stuff in Listing 5-1. That's where practice makes perfect. The more code you write, the more sensitive your eyes will become to things like extraneous semicolons and other programming goofs.

Too many curly braces

You're looking for the nearest gas station, so you ask one of the locals. "Go to the first traffic light and make a left," says the local. You go straight for a few streets and see a blinking yellow signal. You turn left at the signal and travel for a mile or so. What? No gas station? Maybe you mistook the blinking signal for a real traffic light.

You come to a fork in the road and say to yourself, "The directions said nothing about a fork. Which way should I go?" You veer right, but a minute later you're forced onto a highway. You see a sign that says Next Exit 24 Miles. Now you're really lost, and the gas gauge points to S. (The *S* stands for Stranded.)

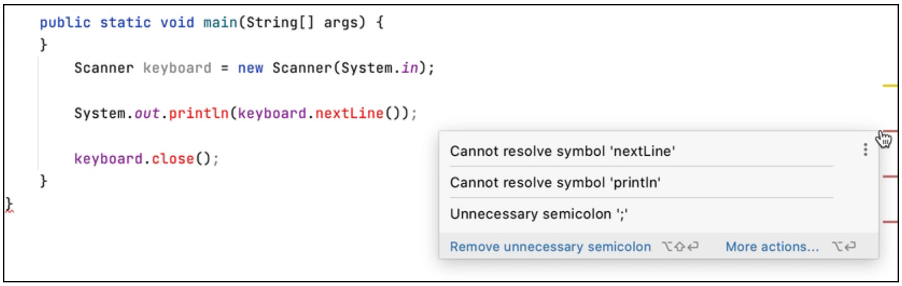
Here's what happened: You made an honest mistake. You shouldn't have turned left at the yellow blinking light. That mistake alone wasn't so terrible. But that first mistake led to more confusion, and eventually, your choices made no sense at all. If you hadn't turned at the blinking light, you'd never have encountered that stinking fork in the road and you'd never have gotten stuck on a highway to nowhere.

Is there a point to this story? Of course there is. A computer can get itself into the same sort of mess. The computer notices an error in your program. Then, metaphorically speaking, the computer takes a fork in the road — a fork based on the original error — a fork for which none of the alternatives leads to good results.

Here's an example. You're retyping the code in Listing 5-1, and you mistakenly type an extra curly brace:

```
//The following code is incorrect:
import java.util.Scanner;

public class EchoLine {

    public static void main(String[] args) {
    }
        Scanner keyboard = new Scanner(System.in);
        System.out.println(keyboard.nextLine());

        keyboard.close();
    }
}
```

You hover over one of the markers on the right side of IntelliJ's editor. You see the messages shown in Figure 5-18.



```
    public static void main(String[] args) {
    }
        Scanner keyboard = new Scanner(System.in);

        System.out.println(keyboard.nextLine());

        keyboard.close();
    }
}
```

Cannot resolve symbol 'nextLine'

Cannot resolve symbol 'println'

Unnecessary semicolon ';'

FIGURE 5-18: Three error messages.

IntelliJ is confused because some of the program's code is completely out of place. IntelliJ displays three messages: `Cannot resolve symbol 'keyboard'`, `Cannot resolve symbol 'println'`, and `Unnecessary semicolon ';'`. None of these messages addresses the cause of the problem. IntelliJ is trying to make the best of a bad situation, but at this point, you shouldn't believe a word that IntelliJ says.

Computers aren't smart animals, and if someone programs IntelliJ to suggest `Remove unnecessary semicolon`, that's exactly what IntelliJ suggests. (Some people say that computers make them feel stupid. For me, it's the opposite. A computer reminds me how dumb a machine can be and how smart a person can be. I like that.)

When you see a bunch of error messages, read each error message carefully. Ask yourself what you can learn from each message. But don't take each message as the authoritative truth. When you've exhausted your efforts with IntelliJ's messages, return to your efforts to stare carefully at the code.

REMEMBER

If you get more than one error message, always look carefully at each message in the bunch. Sometimes a helpful message hides among a bunch of not-so-helpful messages.

Misspelling words (and other missteps)

You've found an old family recipe for deviled eggs (one of my favorites). You follow every step as carefully as you can, but you leave out the salt because of your grandmother's high blood pressure. You hand your grandmother an egg (a finished masterpiece). "Not enough pepper," she says, and she walks away.

The next course is beef bourguignon. You take an unsalted slice to dear old Granny. "Not sweet enough," she groans, and she leaves the room. "But that's impossible," you think. "There's no sugar in beef bourguignon. I left out the salt." Even so, you go back to the kitchen and prepare mashed potatoes. You use unsalted butter, of course. "She'll love it this time," you think.

"Sour potatoes! Yuck!" Granny says, as she goes to the sink to spit it all out. Because you have a strong ego, you're not insulted by your grandmother's behavior. But you're somewhat confused. Why is she saying such different things about three unsalted recipes? Maybe there are some subtle differences that you don't know about.

Well, the same kind of thing happens when you're writing computer programs. You can make the same kind of mistake twice (or at least, make what you think is the same kind of mistake twice) and get different error messages each time.

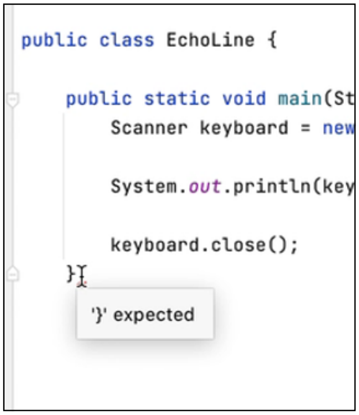
For example, if you change the spelling or capitalization of `println` in Listing 5-1, IntelliJ tells you the method `Cannot resolve method 'Println' in 'PrintStream'`. But if you change `System` to `system`, IntelliJ says that `Cannot resolve symbol 'system'`. And with `System` misspelled, IntelliJ doesn't notice whether `println` is spelled correctly.

In Listing 5-1, if you change the spelling of `args`, nothing goes wrong. The program compiles and runs correctly. But if you change the spelling of `main`, you face some unusual difficulties. (If you don't believe me, read the "Runtime error messages" section, a little later in this chapter.)

Still in Listing 5-1, change the number of close curly braces at the end of the program. With two close braces, everybody's happy. If you accidentally type only one close brace, IntelliJ steers you back on course, telling you `'}' expected`. (See Figure 5-19.) But if you go crazy and type three close braces, IntelliJ misinterprets everything and says `'class' or 'interface' expected`. Unfortunately, adding either word (`class` or `interface`) is no help at all. (See Figure 5-20.)

REMEMBER

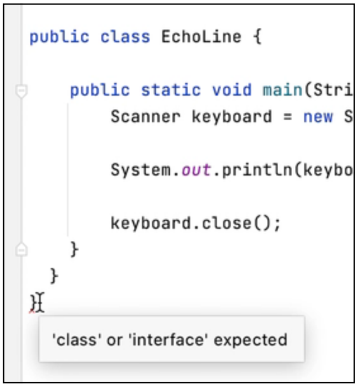
Java responds to errors in many different ways. Two changes in your code might look alike, but similar changes don't always lead to similar results. Each problem in your code requires its own, individualized attention.



```
public class EchoLine {

    public static void main(St
        Scanner keyboard = new

        System.out.println(key

        keyboard.close();
    }}
```

'}' expected

FIGURE 5-19: Remove the second of two close curly braces.



```
public class EchoLine {

    public static void main(Stri
        Scanner keyboard = new S

        System.out.println(keybo

        keyboard.close();
    }
  }
}
```

'class' or 'interface' expected

FIGURE 5-20: You have too many close curly braces, but IntelliJ fails to notice.

Runtime error messages

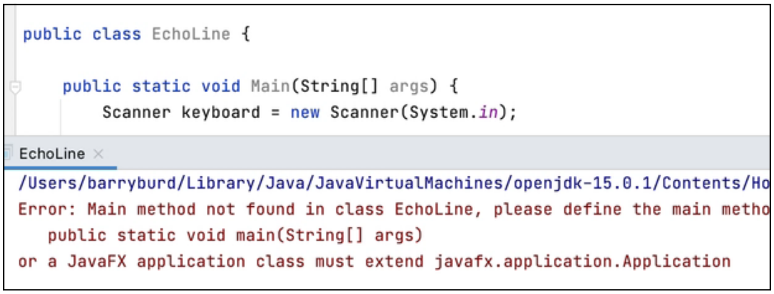
Up to this point in the chapter, I emphasize errors that crop up when you compile a program. Another category of errors hides until you run the program. One example is when you accept the `Add method body` quick fix in Figure 5-16. Another case is when you spell the method name `main` incorrectly.

Assume that, in a moment of wild abandon, you incorrectly spell `main` with a capital `M`:

```
//The following line is incorrect:
public static void Main(String[] args) {
```

When you type the code, everything is hunky-dory. You don't see any error markers.

But then you try to run your program. At this point, the bits hit the fan. The catastrophe is illustrated in Figure 5-21.



```
public class EchoLine {

    public static void Main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
```

```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk-15.0.1/Contents/Ho
Error: Main method not found in class EchoLine, please define the main metho
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

FIGURE 5-21: Whadaya mean "Main method not found"?

Sure, your program has something named `Main`, but does it have anything named `main`? (Yes, I've heard of a famous poet named e. e. cummings, but who the heck is E. E. Cummings?) The computer doesn't presume that your word `Main` means the same thing as the expected word `main`. To make matters worse, the error message in the Run tool window is somewhat misleading. The message starts with `Main` instead of `main`. The English language rule about capitalizing the start of sentence is leading you astray.

One way or another, you need to change `Main` back to `main`. Then everything will be okay.

But in the meantime (or in the "maintime"), how does this improper capitalization make it past the compiler? Why don't you get error messages when you compile the program? And if a capital `M` doesn't upset the compiler, why does this capital `M` mess everything up at runtime?

The answer goes back to the different kinds of words in the Java programming language. As I say in Chapter 4, Java has identifiers, keywords, and a few other kinds of words.

The keywords in Java are cast in stone. If you change `class` to `Class` or `public` to `Public`, you get something new — something that the computer probably can't understand. That's why the compiler chokes on improper keyword capitalizations. It's the compiler's job to make sure that all the keywords are used properly.

On the other hand, the identifiers can bounce all over the place. Sure, there's an identifier named `main`, but you can make up a new identifier named `Main`. (In fact, when you ask IntelliJ to create a command line app, IntelliJ creates a file containing the words `public class Main`.) When the compiler sees a mistyped line, like `public static void Main`, the compiler just assumes that you're making up a brand-new name. So the compiler lets the line pass. You get no complaints from your old friend, the compiler.

But then, when you try to run the code, the computer goes ballistic. The Java virtual machine (JVM) runs your code. (For details, see Chapter 1.) The JVM needs to find a place to start executing statements in your code, so it looks for a starting point named `main`, with a small `m`. If the JVM doesn't see anything named `main`, it gets upset. It screams, "Main method not found in class EchoLine." So at runtime, the JVM, and not the compiler, gives you an error message.

What problem? I don't see a problem

I end this chapter on an upbeat note by showing you some of the things you can change in Listing 5-1 without rocking the boat.

The identifiers that you create

If you create an identifier, that name is up for grabs. For example, in Listing 5-1, you can change `keyboard` to `userInput`:

```
        Scanner userInput = new Scanner(System.in);

        System.out.println(userInput.nextLine());

        userInput.close();
```

A change of this kind is fine as long as you make that change consistently throughout your program. Here's the best way to change a name in your program:

1. **Start with Listing 5-1 in the IntelliJ editor.**

   For details, refer to the earlier "Typing and running a program" section.

2. **In the IntelliJ editor, click on any occurrence of the word** `keyboard`**.**

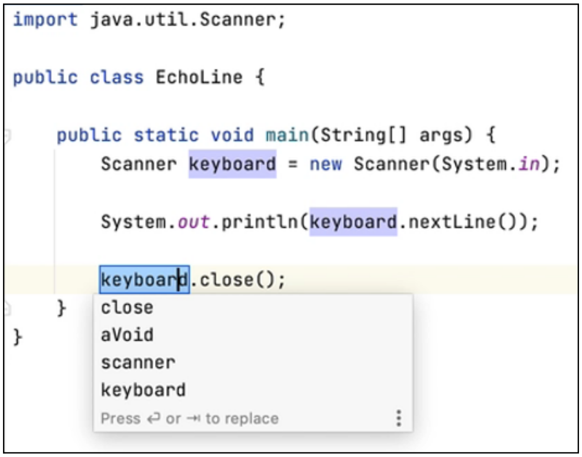
   When you do, IntelliJ highlights all occurrences of the word `keyboard`. That's handy!

3. **Right-click the mouse. On the resulting context menu, choose Refactor ➪ Rename.**

   As a result, IntelliJ suggests some alternatives for the name `keyboard`. (See Figure 5-22.)

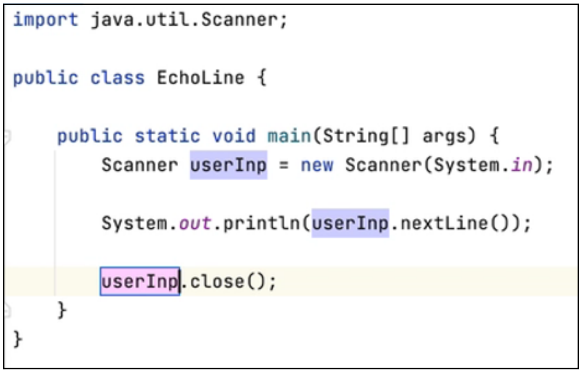
4. **Type** userInput**.**

   As you type, IntelliJ changes all occurrences of the word `keyboard`. (See Figure 5-23.)



```
import java.util.Scanner;

public class EchoLine {

    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.println(keyboard.nextLine());

        keyboard.close();
    }
}
```

FIGURE 5-22: How does `aVoid.close` sound?



```
import java.util.Scanner;

public class EchoLine {

    public static void main(String[] args) {
        Scanner userInp = new Scanner(System.in);

        System.out.println(userInp.nextLine());

        userInp.close();
    }
}
```

FIGURE 5-23: `Inp` without `ut`.

5. **When you're finished typing** userInput**, press Enter.**

   The Enter key tells IntelliJ that you're finished with the renaming. All occurrences of `keyboard` have been replaced by the word `userInput`. For IntelliJ's editor, it's back to business as usual.

IntelliJ's Refactor➪Rename command makes renaming a breeze. More importantly, this feature performs the renaming safely and efficiently. When you're changing `keyboard` to `userInput`, there's no danger of accidentally changing `pianoKeyboard` to `pianouserInput`.

Spaces and indentation

Java isn't fussy about the use of spaces and indentation. All you need to do is keep your program well-organized and readable. Here's an alternative to spacing and indentation of the code in Listing 5-1:

```
import java.util.Scanner;

public class EchoLine
{
    public static void Main(String[] args)
    {
        Scanner userInput =
                new Scanner(System.in);

        System.out.println
                (userInput.nextLine());

        userInput.close();
    }
}
```

If your code isn't formatted in a consistent way, you can get IntelliJ to enforce consistency. Here's how:

1. **Start with Listing 5-1 in the IntelliJ editor.**

   For details, refer to the earlier "Typing and running a program" section.

2. **Change the code's indentation and spacing so that it looks like this:**

   ```
            import java.util.Scanner;

   public class EchoLine {
      public static void main(String[] blah) {
     Scanner keyboard = new Scanner(System.in);
   System.out.println(keyboard.nextLine());
   keyboard.close();
                  } }
   ```

3. **Try to run the newly formatted code.**

   The program runs correctly, but the code is as ugly as red-lipped batfish. Aside from any aesthetic considerations, poorly formatted code is difficult to understand. You want to fix this code as soon as you can.

4. **Click the mouse anywhere inside IntelliJ's editor. Then, on IntelliJ's main menu bar, choose Code ➪ Reformat Code.**

   With that simple act, IntelliJ restores indentation-sanity to your code. IntelliJ reformats your code to look like the code in Listing 5-1. Wow!

How you choose to do things

A program is like a fingerprint. No two programs look much alike. Say that I discuss a programming problem with a colleague. Then we go our separate ways and write our own programs to solve the same problem. Sure, we're duplicating the effort. But will we create the exact same code? Absolutely not. Everyone has their own style, and everyone's style is unique.

I asked fellow Java programmer David Herst to write his own EchoLine program, without showing him my code from Listing 5-1. Here's what he wrote:

```
importjava.io.BufferedReader;
importjava.io.InputStreamReader;
importjava.io.IOException;

public class EchoLine {
  public static void main(String[] args) throws IOException {
    InputStreamReaderisr = new InputStreamReader(System.in);
    BufferedReaderbr = new BufferedReader(isr);
    String input = br.readLine();
    System.out.println(input);
  }
}
```

Don't worry about BufferedReader, InputStreamReader, or things like that. Just notice that, like snowflakes, no two programs are written exactly alike, even if they accomplish the same task. That's nice. It means that your code, however different, can be as good as the next person's. That's encouraging.

TRY IT OUT

COMPILE-TIME ERRORS

The MCV vaccine helps you build an immunity to measles by giving you a mild case of measles. In the same way, you can enhance your immunity against programming errors by making mistakes intentionally in small, throwaway Java programs.

No matter how many years you spend writing code, you'll always have some programming errors in any new code you write. Even the most experienced professional programmers make mistakes. But by practicing with some simple errors, you can discover some errors that beginners make most often, and become accustomed to the "code, test, fix, code again" cycle.

Try these ways of introducing errors in Listing 5-1:

- In the word `println`, change the lowercase letter `l` to a digit `1`.
- Move the entire `System.out.println` line so that it's above the `public static void main` line.
- Delete the parentheses surrounding `System.in`.
- Change `(keyboard.nextLine())` to `(userInput.nextLine)` but don't change any other occurrences of the word `keyboard` in the code.
- Change `System.out.println(keyboard.nextLine());` to the following:

  ```
  System.out.println("I think that I shall never see
                      A poem lovely as a tree.");
  ```

- Experiment by making some other changes. Which of these changes create compile-time errors?

DON'T COPY THIS CODE!

Open your favorite word processing program (Microsoft Word, Apple Pages, or whatever) and create a document containing only the text "Chocolate, royalties, sleep". Most likely, your word processor will automatically use curly quotation marks (“”) instead of straight quotation marks (""), and curly quotation marks aren't good for a Java program. So copy this curly-quoted text from your word processor. In IntelliJ's editor, paste the curly-quoted text into Listing 3-1 (over in Chapter 3). Replace the original `"Chocolate, royalties, sleep"` text in that listing — curly quotation marks and all. See the kind of error messages that IntelliJ displays.

ALLOWABLE CHANGES IN SPACING AND INDENTATION

Change the spacing and indentation in Listing 5-1. Try running this code:

```
import java.util.Scanner;

public class EchoLine
{
        public static void main(String[] args)
        {
                Scanner keyboard = new Scanner(System.in);
                System.out.println(keyboard.nextLine()); keyboard.close();
        }
}
```

IN THIS CHAPTER

- » Declaring variables
- » Assigning values to variables
- » Working with numbers
- » Using Java types

Chapter 6
Using the Building Blocks: Variables, Values, and Types

Back in 1946, John von Neumann wrote a groundbreaking paper about the newly emerging technology of computers and computing. Among other things, he established one fundamental fact: For all their complexity, the main business of computers is to move data from one place to another. Take a number — the balance in a person's bank account. Move this number from the computer's memory to the computer's processing unit. Add a few dollars to the balance and then move it back to the computer's memory. The movement of data . . . that's all there is; there ain't no more.

Good enough! This chapter shows you how to move your data around.

Various Variables and Ways in Which They Vary

Here's an excerpt from a software company's website:

> SnitSoft recognizes its obligation to the information technology community. For that reason, SnitSoft is making its most popular applications available for a nominal charge. For just $5.95 plus shipping and handling, you receive a flash drive containing SnitSoft's premier products.

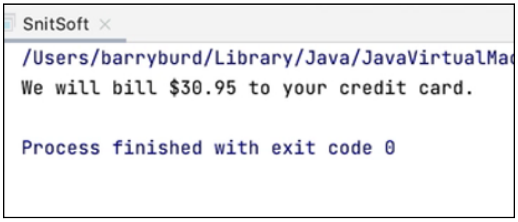
Go ahead. Click the Order Now! link. Just see what happens. You get an order form with two items on it. One item is labeled `$5.95 (Flash drive)`, and the other item reads `$25.00 (Shipping and handling)`. What a rip-off! Thanks to SnitSoft's generosity, you can pay $30.95 for ten cents' worth of software.

Behind the scenes of the SnitSoft web page, a computer program does some scoundrel's arithmetic. The program looks something like the code in Listing 6-1.

LISTING 6-1: SnitSoft's Grand Scam

```
public class SnitSoft {

    public static void main(String[] args) {
        double amount;

        amount = 5.95;
        amount = amount + 25.00;

        System.out.print("We will bill $");
        System.out.print(amount);
        System.out.println(" to your credit card.");
    }
}
```

When I run the code in Listing 6-1 on my own computer (not on the SnitSoft computer), I get the output shown in Figure 6-1.



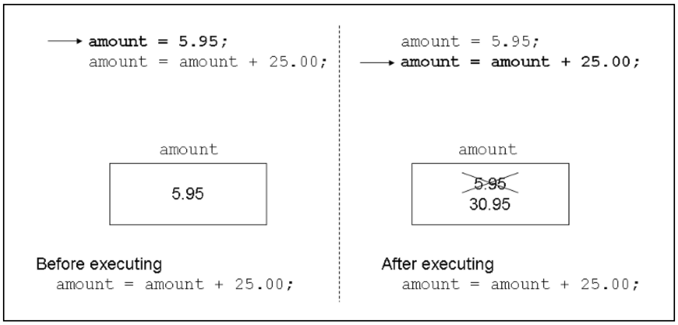
```
/Users/barryburd/Library/Java/JavaVirtualMa
We will bill $30.95 to your credit card.

Process finished with exit code 0
```

FIGURE 6-1: Running the code from Listing 6-1.

Using a variable

The code in Listing 6-1 makes use of a variable named `amount`. A *variable* is a placeholder. You can stick a number like 5.95 into a variable. After you've placed a number in the variable, you can change your mind and put a different number, like 30.95, into the variable. (That's what varies in a variable.) Of course, when you put a new number in a variable, the old number is no longer there. If you didn't save the old number somewhere else, the old number is gone.

Figure 6-2 gives a before-and-after picture of the code in Listing 6-1. When the computer executes `amount = 5.95`, the variable `amount` has the number 5.95 in it. Then, after the `amount = amount + 25.00` statement is executed, the variable `amount` suddenly has 30.95 in it. When you think about a variable, picture a place in the computer's memory where wires and transistors store 5.95, 30.95, or whatever. In Figure 6-2, imagine that each box is surrounded by millions of other such boxes.



```
──→ amount = 5.95;
    amount = amount + 25.00;
```

```
    amount = 5.95;
──→ amount = amount + 25.00;
```

FIGURE 6-2: A variable (before and after).

Now you need some terminology. (You can follow along in Figure 6-3.) The thing stored in a variable is called a *value.* A variable's value can change during the run of a program (when SnitSoft adds the shipping-and-handling cost, for example). The value stored in a variable isn't necessarily a number. (You can, for example, create a variable that always stores a letter.) The kind of value stored in a variable is a variable's *type.* (You can read more about types in the rest of this chapter and in the next two chapters.)

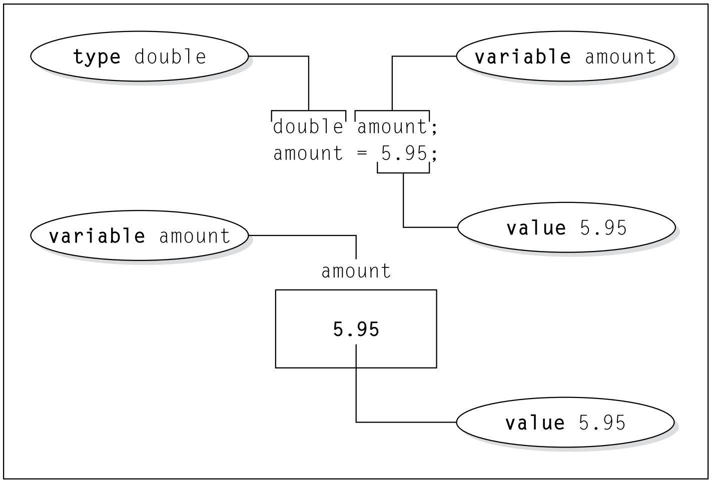


FIGURE 6-3: A variable, its value, and its type.

TECHNICAL STUFF

There's a subtle, almost unnoticeable difference between a variable and a variable's *name.* Even in formal writing, I often use the word *variable* when I mean *variable name.* Strictly speaking, `amount` is the variable name, and all the memory storage associated with `amount` (including the value and type of `amount`) is the variable itself. If you think this distinction between *variable* and *variable name* is too subtle for you to worry about, join the club.

Every variable name is an identifier — a name that you can make up in your own code (for more about this topic, see Chapter 4). In preparing Listing 6-1, I made up the name `amount`.

Understanding assignment statements

The statements with equal signs in Listing 6-1 are called assignment statements. In an *assignment statement,* you assign a value to something. In many cases, this something is a variable.

You should get into the habit of reading assignment statements from right to left. For example, the first assignment statement in Listing 6-1 says, "Assign `5.95` to the `amount` variable." The second assignment statement is just a bit more complicated. Reading the second assignment statement from right to left, you get "Add `25.00` to the value that's already in the `amount` variable and make that number (`30.95`) be the new value of the `amount` variable." For a graphic, hit-you-over-the-head illustration of this concept, see Figure 6-4.

REMEMBER

In an assignment statement, the thing being assigned a value is always on the left side of the equal sign.

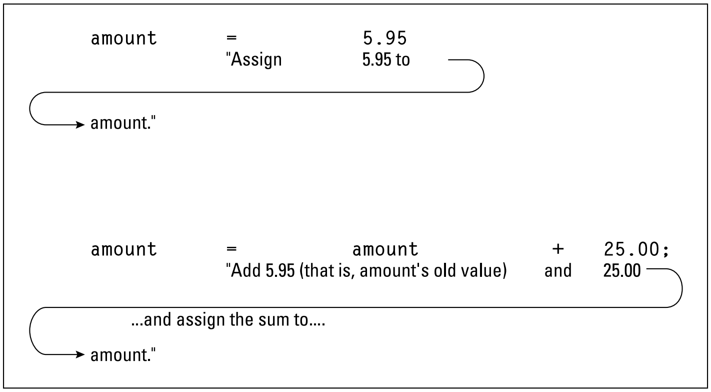

amount = 5.95
"Assign 5.95 to
amount."
amount = amount + 25.00;
"Add 5.95 (that is, amount's old value) and 25.00
...and assign the sum to....
amount."

FIGURE 6-4: Reading an assignment statement from right to left.

FORGET WHAT YOU'VE SEEN

In Listing 6-1, and in other examples throughout this book, I do something that experienced programmers avoid doing: I put actual numbers, such as `5.95` and `25.00`, in my Java code. This is called *hard-coding* the values. I hard-coded values to keep these introductory programming examples as simple as possible.

But in most real-life applications, hard-coding is bad. Imagine a day when SnitSoft raises its shipping-and-handling fee from $25.00 to $35.00. Then the program in Listing 6-1 no longer works correctly. Someone has to launch IntelliJ, look over the code, change the code, test the new code, and distribute the new code to the people who run it. What a pain! For the 10-line program in Listing 6-1, this process takes minutes. For a 10,000-line program in a real-life setting, this process might take days, weeks, or months.

Rather than hard-code values, you should type values on the keyboard during the run of the program. If that's not practical, your program can read values from a computer's hard disk. (Chapter 16 has the scoop on reading from disk files.) One way or another, you should design your program to work with all values, not only with specific values such as `5.95` and `25.00`.

Keep reading this book's examples. When you see my hard-coded values, remember that I use hard-coded values to keep you from being distracted by input and output problems. I keep you focused on whatever new ideas each example has to offer. I don't do hard-coding to convince you that hard-coding is good programming practice. In fact, it's not.

To wrap or not to wrap?

The last three statements in Listing 6-1 use a neat trick. You want the program to display just one line on the screen, but this line contains three different things:

- The line starts with `We will bill $`.
- The line continues with the `amount` variable's value.
- The line ends with `to your credit card`.

These are three separate things, so you put these things in three separate statements. The first two statements are calls to `System.out.print`. The last statement is a call to `System.out.println`.

Calls to `System.out.print` display text on part of a line and then leave the cursor at the end of the current line. After `System.out.print` is executed, the cursor is still at the end of the same line, so the next `System.out.`*`whatever`* can continue printing on that same line. With several calls to `print` capped off by a single call to `println`, the result is just one nice-looking line of output, as Figure 6-5 illustrates.

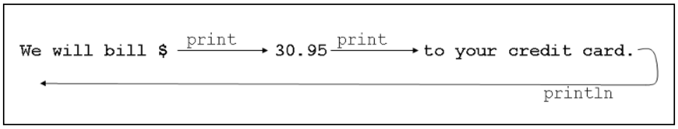


FIGURE 6-5: The roles played by `System.out.print` and `System.out.println`.

REMEMBER

A call to `System.out.print` writes some things and leaves the cursor sitting at the end of the line of output. A call to `System.out.println` writes things and then finishes the job by moving the cursor to the start of a brand-new line of output.

What Do All Those Zeros and Ones Mean?

Here's a word:

> gift

The question for discussion is, what does that word mean? Well, it depends on who looks at the word. For example, an English-speaking reader would say that *gift* stands for something one person bestows upon another in a box covered in brightly colored paper and ribbons:

> Look! I'm giving you a **gift**!

But in German, the word *gift* means "poison:"

> Let me give you some **gift**, my dear.

And in Swedish, *gift* can mean either "married" or "poison:"

> As soon as they got **gift**, she slipped a **gift** into his drink.

Then there's French. In France, there's a candy bar named "Gift":

> He came for the holidays, and all he gave me was a bar of **Gift**.

What do the letters g-i-f-t really mean? Well, they mean nothing until you decide on a way to interpret them. The same is true of the 0's and 1's inside a computer's circuitry.

Take, for example, the sequence 01001010. This sequence can stand for the letter *J*, but it can also stand for the number 74. That same sequence of zeros and ones can stand for $1.0369608636003646 \times 10^{-43}$. And when interpreted as screen pixels, the same sequence can represent the pixels shown in Figure 6-6. The meaning of 01001010 depends entirely on the way the software interprets this sequence.

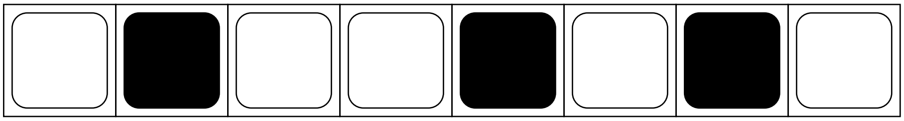


FIGURE 6-6: An extreme close-up of eight black-and-white screen pixels.

Types and declarations

How do you tell the computer what 01001010 stands for? The answer is in the concept called type. The *type* of a variable describes the kinds of values that the variable is permitted to store.

In Listing 6-1, look at the first line in the body of the `main` method:

```
double amount;
```

This line is called a *variable declaration.* Putting this line in your program is like saying, "I'm declaring my intention to have a variable named `amount` in my program." This line reserves the name `amount` for your use in the program.

In this variable declaration, the word *double* is a Java keyword. This word `double` tells the computer which kinds of values you intend to store in `amount`. In particular, the word `double` stands for numbers between -1.8×10^{308} and 1.8×10^{308}. That's an enormous range of numbers. Without the fancy ×10 notation, the second of these numbers is

```
18000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
000000000000000000000000000000000000.0
```

If the folks at SnitSoft ever charge that much for shipping and handling, they can represent the charge with a variable of type `double`.

What's the point?

More important than the humongous range of the `double` keyword's numbers is the fact that a `double` value can have digits to the right of the decimal point. After you declare `amount` to be of type `double`, you can store all sorts of numbers in `amount`. You can store 5.95, 0.02398479, or -3.0. In Listing 6-1, if I hadn't declared `amount` to be of type `double`, I wouldn't have been able to store 5.95. Instead, I would have had to store plain old 5 or dreary old 6, with no digits beyond the decimal point.

For more info on numbers without decimal points, see Chapter 7.

TECHNICAL STUFF

This paragraph deals with a really picky point, so skip it if you're not in the mood. People often use the phrase *decimal number* to describe a number with digits to the right of the decimal point. The problem is, the syllable "dec" stands for the number 10, so the word *decimal* implies a base-10 representation. Because computers store base-2 (not base-10) representations, the word *decimal* to describe such a number is a misnomer. But in this book, I just can't help myself. I'm calling them decimal numbers, whether the techies like it or not.

TRY IT OUT

Here are some things for you to try:

NUMBER CRUNCHING

Change the number values in Listing 6-1 and run the program with the new numbers.

VARYING A VARIABLE

In Listing 6-1, change the variable name `amount` to another name. Change the name consistently throughout the Listing 6-1 code. Then run the program with its new variable name.

USING UNDERSCORES

Modify the code in Listing 6-1 so that shipping-and-handling costs $1 million. Use `1_000_000.00` (with underscores) to represent the million-dollar amount.

MORE INFORMATION, PLEASE

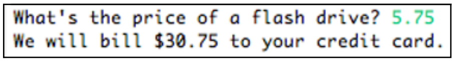
Modify the code in Listing 6-1 so that it displays three values: the original price of the flash drive, the cost of shipping and handling, and the combined cost.

Reading Decimal Numbers from the Keyboard

I don't believe it! SnitSoft is having a sale! For one week only, you can get the SnitSoft flash drive for the low price of just $5.75! Better hurry up and order one.

No, wait! Listing 6-1 has the price fixed at $5.95. I have to revise the program.

I know. I'll make the code more versatile. I'll input the amount from the keyboard. Listing 6-2 has the revised code, and Figure 6-7 shows a run of the new code.



```
What's the price of a flash drive? 5.75
We will bill $30.75 to your credit card.
```

FIGURE 6-7: Getting the value of a `double` variable.

LISTING 6-2: Getting a Double Value from the Keyboard

```
import java.util.Scanner;

public class VersatileSnitSoft {

    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        double amount;

        System.out.print("What's the price of a flash drive? ");
        amount = keyboard.nextDouble();
        amount = amount + 25.00;

        System.out.print("We will bill $");
        System.out.print(amount);
        System.out.println(" to your credit card.");

        keyboard.close();
    }
}
```

WARNING

As I mention over in Chapter 4, grouping separators vary from one country to another. The run shown in Figure 6-7 is for a computer configured in the United States, where 5.75 means "five and seventy-five hundredths." But the run might look different on a computer that's configured in what I call a "comma country" — a country where 5,75 means "five and seventy-five hundredths." If you live in a comma country and you type **5.75** exactly as it's shown in Figure 6-7, you probably get an error message (an `InputMismatchException`). If so, change the number amounts in your file to match your country's number format. When you do, you should be okay.

Though these be methods, yet there is madness in't

Notice the call to the `nextDouble` method in Listing 6-2. Over in Listing 5-1, in Chapter 5, I use `nextLine`; but here in Listing 6-2, I use `nextDouble`.

In Java, each type of input requires its own, special method. If you're getting a line of text, then `nextLine` works just fine. But if you're reading stuff from the keyboard and you want that stuff to be interpreted as a number, you need a method like `nextDouble`.

To go from Listing 6-1 to Listing 6-2, I added an `import` declaration and some stuff about `new Scanner(System.in)`. You can find out more about these concepts by reading the "Getting numbers, words, and other things" section in Chapter 5. (You can find out even more about input and output by visiting Chapter 16.) And more examples (more `keyboard.next`*`Something`* methods) are in Chapters 7 and 8.

Methods and assignments

Note how I use `keyboard.nextDouble` in Listing 6-2. The call to method `keyboard.nextDouble` is part of an assignment statement. If you look in Chapter 5 at the section on how the `EchoLine` program works, you see that the computer can substitute something in place of a method call. The computer does this in Listing 6-2. When you type **5.75** on the keyboard, the computer turns

```
amount = keyboard.nextDouble();
```

into

```
amount = 5.75;
```

(The computer doesn't really rewrite the code in Listing 6-2. This `amount = 5.75` line simply illustrates the effect of the computer's action.) In the second assignment statement in Listing 6-2, the computer adds 25.00 to the 5.75 that's stored in `amount`.

Some method calls have this substitution effect, and others (like `System.out.println`) don't. To find out more about this topic, see Chapter 15.

WHO DOES WHAT, AND HOW?

When you write a program, you're called a *programmer,* but when you run a program, you're called a *user.* So when you test your own code, you're both the programmer and the user.

Suppose that your program contains a `keyboard.next`*`Something`*`()` call, like the calls in Listings 5-1 (in Chapter 5) and 6-2. Then your program gets input from the user. But, when the program runs, how does the user know to type something on the keyboard? If the user and the programmer are the same person, and the program is fairly simple,

(continued)

knowing what to type is no big deal. For example, when you start running the code in Listing 5-1, you have this book in front of you, and the book says "The computer is waiting for you to type something. You type one line of text." So you type the text and press Enter. Everything is fine.

But very few programs come with their own books. In many instances, when a program starts running, the user has to stare at the screen to figure out what to do next. The code in Listing 6-2 works in this stare-at-the-screen scenario. In Listing 6-2, the first call to `print` puts an informative message (`What's the price of a flash drive?`) on the user's screen. A message of this kind is called a *prompt*.

When you start writing programs, you can easily confuse the roles of the prompt and the user's input. *Remember:* No preordained relationship exists between a prompt and the subsequent input. To create a prompt, you call `print` or `println`. Then, to read the user's input, you call `nextLine`, `nextDouble`, or one of the `Scanner` class's other `next`*`Something`* methods. These `print` and `next` calls belong in two separate statements. Java has no commonly used, single statement that does both the prompting and the "`next`-ing."

As the programmer, your job is to combine the prompting and the `next`-ing. You can combine prompting and `next`-ing in all kinds of ways. Some ways are helpful to the user, and some ways aren't, as described in this list:

- **If you don't have a call to `print` or `println`, the user sees no prompt.** A blinking cursor sits quietly and waits for the user to type something. The user has to guess what kind of input to type. Occasionally that's okay, but usually it isn't.
- **If you call `print` or `println` but you don't call a `keyboard.next`*`Something`* method, the computer doesn't wait for the user to type anything.** The program races to execute whatever statement comes immediately after the `print` or `println`.
- **If your prompt displays a misleading message, you mislead the user.** Java has no built-in feature that checks the appropriateness of a prompt. That's not surprising. Most computer languages have no prompt-checking feature.

Be careful with your prompts. Be nice to your user. Remember that you were once a humble computer user, too.

Variations on a Theme

In Listing 6-1, it takes two lines to give the amount variable its first value:

```
double amount;
amount = 5.95;
```

You can do the same thing with just one line:

```
double amount = 5.95;
```

When you do this, you don't say that you're "assigning" a value to the amount variable. The line double amount = 5.95 isn't called an "assignment statement." Instead, this line is called a "declaration with an initialization." You're *initializing* the amount variable. You can do all sorts of things with initializations — even arithmetic:

```
double gasBill  = 174.59;
double elecBill = 84.21;
double H2OBill  = 22.88;
double total    = gasBill + elecBill + H2OBill;
```

If it looks like a double and smells like a double . . .

This chapter's "Types and declarations" section suggests that some variables don't store decimal numbers. Take, for example, the following declaration:

```
char newValue = 'B';
```

Two parts of this declaration indicate that newValue doesn't store a decimal number:

- The word char isn't the same as the word double.
- The letter 'B' isn't a decimal number.

In fact, two parts of the declaration

```
double amount = 5.95;
```

indicate that amount stores a decimal number:

- The word double means "decimal number."
- The value 5.95 is a decimal number.

Wouldn't it be nice if you didn't have to tell Java the same thing more than once? The good news is, you can:

```
var amount = 5.95;
```

In this declaration, the word var indicates that amount stands for a variable. Java decides on the variable's type based on the value after the equal sign. So Java, because it's a very smart programming language, looks at the value 5.95 and decides that amount is a variable of type double. In a way, Java looks at var amount = 5.95 and mentally replaces the word var with the word double. (Okay. Saying that Java does anything "mentally" is a stretch, but you know what I mean.)

Using var, you can remove the redundancy from lines like

```
Scanner keyboard = new Scanner(System.in);
```

(See Listing 6-2.) In this declaration, why do you bother repeating the word Scanner? You convey the same message when you write

```
var keyboard = new Scanner(System.in);
```

This new-and-improved var keyboard declaration is easier to read and understand. How much time do you save? Let's take a wild guess and say that you save one millisecond every time you read var keyboard instead of Scanner keyboard. Over the course of a programmer's career, those milliseconds add up. If you stack those milliseconds end to end, the programmer saves several days' worth of needless effort. (Okay, "several days' worth" is also a wild guess. But you get the idea.)

WARNING

The word var doesn't work in a declaration with no initialization. For example, if you write

```
var amount;
amount = 5.95;
```

on two separate lines, Java tells you to go fly a kite.

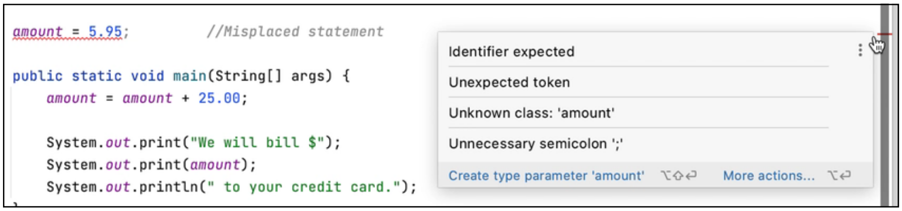
Moving variables from place to place

It helps to remember the difference between initializations and assignments. For one thing, you can drag a declaration with its initialization outside of a method:

```
//This is okay:
public class SnitSoft {
    static double amount = 5.95;

    public static void main(String[] args) {
        amount = amount + 25.00;

        System.out.print("We will bill $");
        System.out.print(amount);
        System.out.println(" to your credit card.");
    }
}
```

You can't do the same thing with assignment statements (see the following code and Figure 6-8):

```
//This does not compile:
public class BadSnitSoftCode {
    static double amount;

    amount = 5.95;         //Misplaced statement

    public static void main(String[] args) {
        amount = amount + 25.00;

        System.out.print("We will bill $");
        System.out.print(amount);
        System.out.println(" to your credit card.");
    }
}
```



```
amount = 5.95;          //Misplaced statement

public static void main(String[] args) {
    amount = amount + 25.00;

    System.out.print("We will bill $");
    System.out.print(amount);
    System.out.println(" to your credit card.");
```

Identifier expected

Unexpected token

Unknown class: 'amount'

Unnecessary semicolon ';'

FIGURE 6-8: A failed attempt to compile BadSnitSoftCode.

You can't drag statements outside of methods. (Even though a variable declaration ends with a semicolon, a variable declaration isn't considered to be a statement. Go figure!)

The advantage of putting a declaration outside of a method is illustrated in Chapter 13. While you wait impatiently to reach that chapter, notice how I added the word `static` to each declaration that I pulled out of the `main` method. I had to do this because the `main` method's header has the word `static` in it. Not all methods are static. In fact, most methods aren't static. But whenever you pull a declaration out of a `static` method, you have to add the word `static` at the beginning of the declaration. All the mystery surrounding the word `static` is resolved in Chapter 14.

WARNING

The previous section introduces `var` to make declaring variables easier, and this section touches on the idea that you can declare a variable outside of a method. Both of these tricks work well on their own, but you can't combine the two of them in the same declaration. For example, the following code gives you nothing but grief:

```
//This is NOT okay:
public class SnitSoft {
    static var amount = 5.95;
```

In this code, the declaration of `amount` isn't inside of a method, so you can't use `var` to declare `amount`. Instead, use the type name `double`.

Combining variable declarations

The code in Listing 6-1 has only one variable (as if variables are in short supply). You can get the same effect with several variables:

```
public class SnitSoftNew {

    public static void main(String[] args) {
        double flashDrivePrice;
        double shippingAndHandling;
        double total;

        flashDrivePrice = 5.95;
        shippingAndHandling = 25.00;
        total = flashDrivePrice + shippingAndHandling;

        System.out.print("We will bill $");
        System.out.print(total);
        System.out.println(" to your credit card.");
    }
}
```

This new code gives you the same output as the code in Listing 6-1. (Refer to Figure 6-1.)

The new code has three declarations — one for each of the program's three variables. Because all three variables have the same type (the type `double`), I can modify the code and declare all three variables in one fell swoop:

```
double flashDrivePrice, shippingAndHandling, total;
```

Which is better — one declaration or three declarations? Neither is better. It's a matter of personal style.

You can even add initializations to a combined declaration. When you do, each initialization applies to only one variable. For example, with the line

```
double flashDrivePrice, shippingAndHandling = 25.00, total;
```

the value of `shippingAndHandling` becomes `25.00`, but the variables `flashDrivePrice` and `total` get no particular value.

TRY IT OUT

Would you like some practice with this section's concepts? You got it!

TIP THE PARKING ATTENDANT

An online blog advises a $2 tip when a parking attendant fetches your car in a New York City garage. Write a program like the one in Listing 6-2. When the program runs, you type the garage's posted price for parking your car. The program tells you how much you'll pay after adding the $2 tip.

DOUBLE PRICE

Modify the code in Listing 6-2 so that whatever a flash drive normally costs, the program charges twice that amount. In other words, the price for a $5 flash drive ends up being $10, and the price for a $100 flash drive becomes $200.

Experimenting with JShell

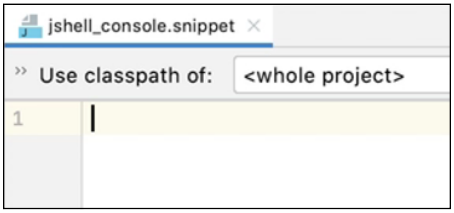
The programs in this book all begin with the same old tiresome refrain:

```
public class SomethingOrOther {

    public static void main(String[] args) {
```

Dealing with this boilerplate code can be annoying, especially when your goal is to test the effect of executing a few simple statements. Fortunately, there's a way avoid all this `public class` and `public static` business. Here's what you do:

1. **On IntelliJ's main menu, choose Tools ➪ JShell Console.**

   As a result, IntelliJ creates an empty Editor tab. The tab's name includes the words `jshell_console`. (See Figure 6-9.)



Use classpath of: <whole project>

FIGURE 6-9: The JShell editor.

   JShell is Java's universal scratch pad. You can test all kinds of ideas on the JShell tab. And the best part is, you don't have to write a complete program. In some cases, you don't even have to write complete Java statements.

TECHNICAL STUFF

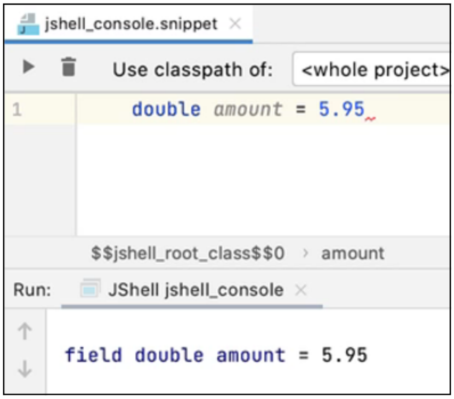
   JShell is only one example of a language's *Read Evaluate Print Loop* (REPL). Many programming languages have REPLs. If you've installed Python on your computer, you can launch Python's REPL by typing `python` in the Command Prompt or Terminal window. With Node.js installed, typing the word `node` starts up a JavaScript REPL.

2. **In the new editor area (the JShell tab), type the following line:**

   ```
   double amount = 5.95
   ```

3. **On Windows, press Ctrl+Enter; on a Mac, press Cmd+Enter.**

   IntelliJ's Run tool window displays the message shown in Figure 6-10.



```
double amount = 5.95
```

```
field double amount = 5.95
```

FIGURE 6-10: JShell acknowledges that you've declared the `amount` variable.

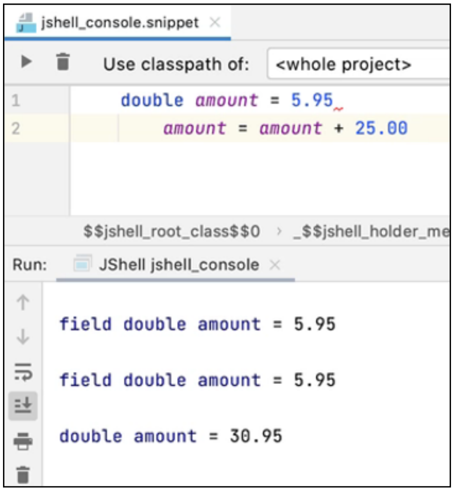
4. **On the next line of the editor, type**

```
amount = amount + 25.00
```

and then press Ctrl+Enter or Cmd+Enter.

IntelliJ's Run tool window adds two new lines, as shown in Figure 6-11.

When you press Ctrl+Enter or Cmd+Enter, IntelliJ runs all statements in the JShell editor, even if it means repeating the run of some of those statements.



```
double amount = 5.95
    amount = amount + 25.00
```

```
field double amount = 5.95

field double amount = 5.95

double amount = 30.95
```

FIGURE 6-11: JShell follows your Java instructions.

5. **Click the Trash Can icon on the left edge of the Run tool window.**

When you do, IntelliJ clears the Run tool window so that you can start afresh.

6. **Add a new line in the JShell editor. The new line looks like this:**

```
7 + 10
```

The line `7 + 10` isn't a complete Java statement — even if you add a semicolon at the end. You won't get away with a line like `7 + 10;` in a regular Java program, but JShell isn't finicky. When you press Ctrl+Enter or Cmd+Enter, the following line shows up in the Run tool window:

```
7 + 10 = 17
```

7. **Add another new line in the JShell editor. The new line looks like this:**

```
amount + 100.00
```

When you press Ctrl+Enter or Cmd+Enter, the following line shows up in the Run tool window:

```
amount + 100.00 = 130.95
```

8. **Click the X icon on the JShell tab at the top of the editor.**

IntelliJ asks whether you want to terminate the JShell process.

9. **To confirm, choose Terminate.**

IntelliJ closes the JShell tab. Your experiment with JShell has come to an end.

When you use JShell, you hardly ever type an entire program. Instead, you type a Java statement, and then JShell responds to your statement, and then you type a second statement, and then JShell responds to your second statement, and then you type a third statement, and so on. A single statement is enough to get a response from JShell.

With JShell, you can test your statements before you put them into a full-blown Java program. That makes JShell a truly useful tool.

TIP

IntelliJ's JShell Console has some quirks. For example, I see a little red line when I type the declaration

```
double amount = 5.95
```

If I hover the mouse over that red line, a little popup says *';' expected*. A red marker normally means "This is an error. IntelliJ refuses to run this code." But, when I press Ctrl+Enter or Cmd+Enter, IntelliJ runs the code and shows me the result in the Run tool window. Go figure!

If the red line makes me nervous, I can add a semicolon. I've heard rumors that IntelliJ sometimes refuses to run a statement that doesn't end with a semicolon, but I've never observed this behavior myself.

TRY IT OUT

FUN WITH JSHELL

On IntelliJ's main menu, choose Tools➪JShell Console. Then type the following code into the JShell editor. When you finish typing, press Ctrl+Enter (on Windows) or Cmd+Enter (on a Mac).

```
double bananaCalories = 100.0
double appleCalories = 95.0
double dietSodaCalories = 0.0
double cheeseburgerCalories = 500.0
bananaCalories + appleCalories +
    dietSodaCalories + cheeseburgerCalories
```

Keep JShell running in preparation for the next experiment.

ADDING AND REMOVING JSHELL CODE

This experiment has several parts. After each part, see what new text appears in the Run tool window. Decide for yourself why that particular text appears.

1. **In the JShell editor, type**

   ```
   height = 5.6
   ```

   and then press Ctrl+Enter or Cmd+Enter.

2. **On the next line in the editor, type**

   ```
   double height
   ```

   and then press Ctrl+Enter or Cmd+Enter.

3. **Without typing anything new in the editor, press Ctrl+Enter or Cmd+Enter again.**

4. **In the JShell editor, erase the line**

   ```
   double height
   ```

   and then press Ctrl+Enter or Cmd+Enter.

5. **Look for a Trash Can icon at the top of the JShell editor.**

   When you hover the cursor over that icon, a tip labeled Drop All Code Snippets appears.

6. **Click the Trash Can icon.**

7. **With the cursor in the IntelliJ editor, press Ctrl+Enter or Cmd+Enter once again.**

IN THIS CHAPTER

- Processing whole numbers
- Making new values from old values
- Understanding Java's more exotic types

Chapter 7 Numbers and Types

Not so long ago, people thought computers did nothing but perform big, number-crunching calculations. Computers solved arithmetic problems, and that was the end of the story.

In the 1980s, with the widespread use of word processing programs, the myth of the big metal math brain went by the wayside. But even then, computers made great calculators. After all, computers are very fast and very accurate. Computers never need to count on their fingers. Best of all, computers don't feel burdened when they do arithmetic. I hate ending a meal in a good restaurant by worrying about the tax and tip, but computers don't mind that stuff at all. (Even so, computers seldom go out to eat.)

Using Whole Numbers

Let me tell you, it's no fun being an adult. Right now I have four little kids in my living room. They're all staring at me because I have a bag full of gumballs in my hand. With 30 gumballs in the bag, the kids are all thinking, "Who's the best? Who gets more gumballs than the others? And who's going to be treated unfairly?" They insist on a complete, official gumball count, with each kid getting exactly the same number of tasty little treats. I must be careful. If I'm not, I'll never hear the end of it.

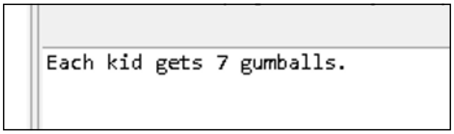
With 30 gumballs and four kids, there's no way to divide the gumballs evenly. Of course, if I get rid of a kid, I can give 10 gumballs to each kid. The trouble is, gumballs are disposable; kids are not. So my only alternative is to divvy up what gumballs I can and dispose of the rest. “Okay, think quickly,” I say to myself. “With 30 gumballs and 4 kids, how many gumballs can I promise to each kid?”

I waste no time in programming my computer to figure out this problem for me. When I’m finished, I have the code in Listing 7-1.

LISTING 7-1: **How to Keep Four Kids from Throwing Tantrums**

```
public class KeepingKidsQuiet {

    public static void main(String[] args) {
        int gumballs;
        int kids;
        int gumballsPerKid;

        gumballs = 30;
        kids = 4;
        gumballsPerKid = gumballs / kids;

        System.out.print("Each kid gets ");
        System.out.print(gumballsPerKid);
        System.out.println(" gumballs.");
    }
}
```

Figure 7-1 shows a run of the `KeepingKidsQuiet` program. If each kid gets seven gumballs, then the kids can’t complain that I’m playing favorites. They’ll have to find something else to squabble about.



```
Each kid gets 7 gumballs.
```

FIGURE 7-1: Fair and square.

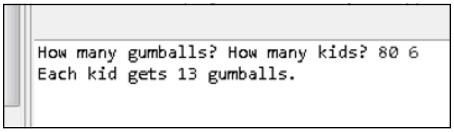
At the core of the gumball problem, I have *whole* numbers — numbers with no digits beyond the decimal point. When I divide 30 by 4, I get 7½, but I can’t take the ½ seriously. No matter how hard I try, I can’t divide a gumball in half, at least not without hearing “my half is bigger than their half.” This fact is reflected nicely in Java. In Listing 7-1, all three variables (`gumballs`, `kids`, and `gumballsPerKid`) are of type `int`. An `int` value is a whole number. When you divide one `int` value by another (as you do with the slash in Listing 7-1), you get another `int`. When you divide 30 by 4, you get 7 — not 7½. You see this in Figure 7-1. Taken together, the statements

```
gumballsPerKid = gumballs/kids;

System.out.print(gumballsPerKid);
```

put the number 7 on the computer screen.

Reading whole numbers from the keyboard

What a life! Yesterday there were four kids in my living room and I had 30 gumballs. Today there are six kids in my house and I have 80 gumballs. How can I cope with all this change? I know! I'll write a program that reads the numbers of gumballs and kids from the keyboard. The program is in Listing 7-2, and a run of the program is shown in Figure 7-2.



```
How many gumballs? How many kids? 80 6
Each kid gets 13 gumballs.
```

FIGURE 7-2: Next thing you know, I'll have 70 kids and 1,000 gumballs.

LISTING 7-2: A More Versatile Program for Kids and Gumballs

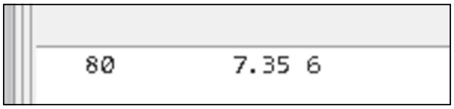
```
import java.util.Scanner;

public class KeepingMoreKidsQuiet {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        int gumballs;
        int kids;
        int gumballsPerKid;

        System.out.print("How many gumballs? How many kids? ");
        gumballs = keyboard.nextInt();
        kids = keyboard.nextInt();

        gumballsPerKid = gumballs / kids;
        System.out.print("Each kid gets ");
        System.out.print(gumballsPerKid);
        System.out.println(" gumballs.");

        keyboard.close();
    }
}
```

You should notice a couple of things about Listing 7-2. First, you can read an int value with the nextInt method. (Refer to the table in Chapter 5.) Second, you can issue successive calls to Scanner methods. In Listing 7-2, I call nextInt twice. All I have to do is separate the numbers I type by blank spaces. In Figure 7-2, I put one blank space between my 80 and my 6, but more blank spaces would work as well.

This blank-space rule applies to many of the Scanner methods. For example, here's some code that reads three numeric values:

```
gumballs = keyboard.nextInt();
costOfGumballs = keyboard.nextDouble();
kids = keyboard.nextInt();
```

Figure 7-3 shows valid input for these three method calls.



```
80       7.35 6
```

FIGURE 7-3: Three numbers for three Scanner method calls.

By the way, if you're staring at Listing 7-2 and wondering what var keyboard means, refer to Chapter 6.

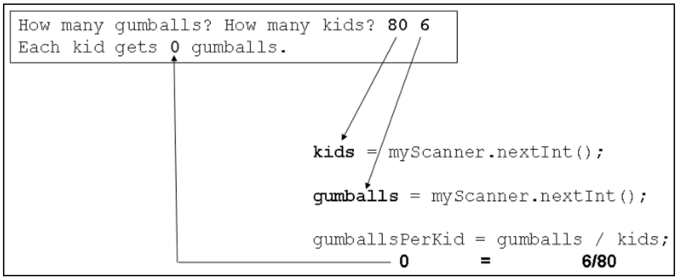
What you read is what you get

When you're writing your own code, you should never take anything for granted. Suppose that you accidentally reverse the order of the gumballs and kids assignment statements in Listing 7-2:

```
//This code is misleading:
System.out.print("How many gumballs? How many kids? ");

kids = keyboard.nextInt();
gumballs = keyboard.nextInt();
```

Here, the line How many gumballs? How many kids? is misleading. Because the kids assignment statement comes before the gumballs assignment statement, the first number you type becomes the value of kids, and the second number you type becomes the value of gumballs. It doesn't matter that your program displays the message How many gumballs? How many kids?. What matters is the order of the assignment statements in the program.

If the `kids` assignment statement accidentally comes first, you can get a strange answer, like the zero answer in Figure 7-4. That's how `int` division works. It just cuts off any remainder. Divide a small number (like 6) by a big number (like 80) and you get 0.



```
How many gumballs? How many kids? 80 6
Each kid gets 0 gumballs.
```

```
kids = myScanner.nextInt();

gumballs = myScanner.nextInt();

gumballsPerKid = gumballs / kids;
```

FIGURE 7-4: How to make six kids very unhappy.

TRY IT OUT

Like the mad scientist in an old horror movie, try these fascinating experiments!

MAKE IT AND BREAK IT

Run the program in Listing 7-2. When the program asks `How many gumballs? How many kids?`, type **80.5 6**. (Actually, if you live in a country where 80,5 represents eighty-and-a-half, type 80,5 instead of 80.5.)

What unpleasant message do you see during this run of the program? Why do you see this message?

BREAK IT AGAIN

Run the program in Listing 7-2. When the program asks `How many gumballs? How many kids?`, type **"80" "6"** (quotation marks and all).

What unpleasant message do you see during this run of the program? Why do you see this message?

A TINY ADDING MACHINE

Write a program that gets two numbers from the keyboard and displays the sum of the two numbers.

Creating New Values by Applying Operators

What could be more comforting than your old friend the plus sign? It was the first thing you learned about in elementary school math. Almost everybody knows how to add two and two. In fact, in English usage, adding two and two is a metaphor for something that's easy to do. Whenever you see a plus sign, one of your brain cells says, "Thank goodness, it could be something much more complicated."

So Java has a plus sign. You can use the plus sign to add two numbers:

```
int apples, oranges, fruit;
apples = 5;
oranges = 16;
fruit = apples + oranges;
```

Of course, the old minus sign is available, too:

```
apples = fruit - oranges;
```

Use an asterisk for multiplication and a forward slash for division:

```
double rate, pay, withholding;
int hours;

rate = 6.25;
hours = 35;
pay = rate * hours;
withholding = pay / 3.0;
```

TIP

When you divide an int value by another int value, you get an int value. The computer doesn't round. Instead, the computer chops off any remainder. If you put System.out.println(11 / 4) in your program, the computer prints 2, not 2.75. If you need a decimal answer, make either (or both) of the numbers you're dividing double values. For example, if you put System.out.println(11.0 / 4) in your program, the computer divides a double value, 11.0, by an int value, 4. Because at least one of the two values is double, the computer prints 2.75.

Finding a remainder

There's a useful arithmetic operator called the *remainder* operator. The symbol for the remainder operator is the percent sign (%). When you put System.out.println(11 % 4) in your program, the computer prints 3. It does so because 4 goes into 11 who-cares-how-many times, with a remainder of 3.

TECHNICAL STUFF

Another name for the remainder operator is the *modulus* operator.

The remainder operator turns out to be fairly useful. After all, a *remainder* is the amount you have left over after you divide two numbers. What if you're making change for $1.38? After dividing 138 by 25, you have 13 cents left over, as shown in Figure 7-5.

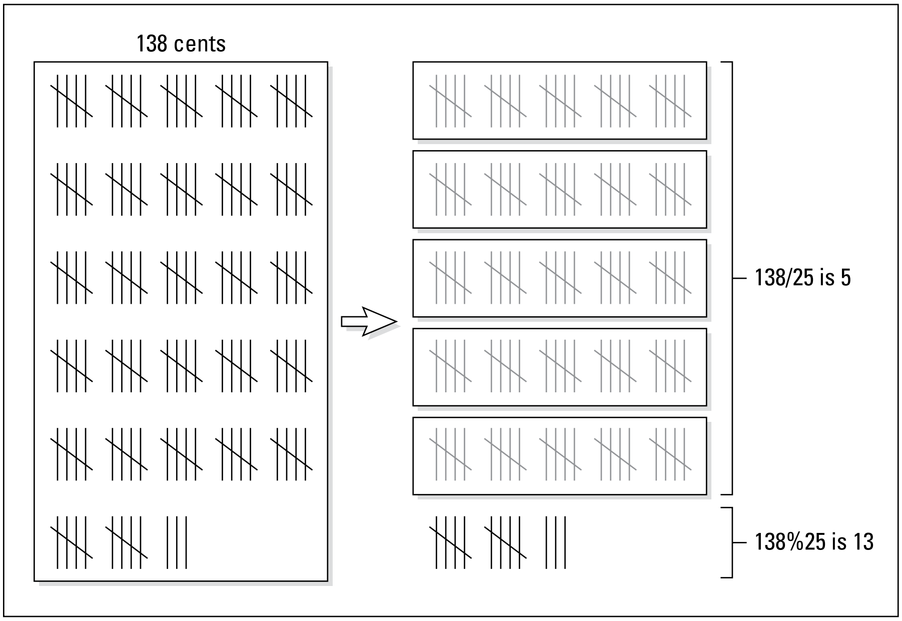


FIGURE 7-5: Hey, bud! Got change for 138 sticks?

The code in Listing 7-3 makes use of this remainder idea.

LISTING 7-3: Making Change

```
import java.util.Scanner;

public class MakeChange {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        int quarters, dimes, nickels, cents;
        int whatsLeft, total;

        System.out.print("How many cents do you have? ");
        total = keyboard.nextInt();
```

(continued)

LISTING 7-3: *(continued)*

```
        quarters = total / 25;
        whatsLeft = total % 25;

        dimes = whatsLeft / 10;
        whatsLeft = whatsLeft % 10;

        nickels = whatsLeft / 5;
        whatsLeft = whatsLeft % 5;

        cents = whatsLeft;

        System.out.println();
        System.out.println("From " + total + " cents you get");
        System.out.println(quarters + " quarters");
        System.out.println(dimes + " dimes");
        System.out.println(nickels + " nickels");
        System.out.println(cents + " cents");

        keyboard.close();
    }
}
```

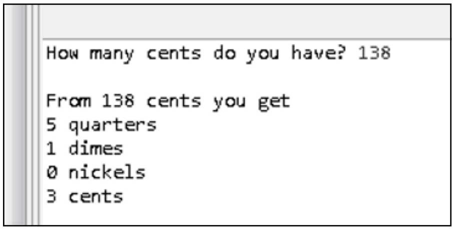
A run of the code in Listing 7-3 is shown in Figure 7-6. You start with a `total` of 138 cents. The statement

```
quarters = total / 25;
```

divides 138 by 25, giving you 5. That means you can make 5 quarters from 138 cents. Next, the statement

```
whatsLeft = total % 25;
```

divides 138 by 25 again and puts only the remainder, 13, into `whatsLeft`. Now you're ready for the next step, which is to take as many dimes as you can out of 13 cents.



```
How many cents do you have? 138

From 138 cents you get
5 quarters
1 dimes
0 nickels
3 cents
```

FIGURE 7-6: Change for $1.38.

You keep going like this until you've divided away all the nickels. At that point, the value of `whatsLeft` is just 3 (meaning 3 cents).

TECHNICAL STUFF

The code in Listing 7-3 makes change in US currency with the following coin denominations: 1 cent, 5 cents (one nickel), 10 cents (one dime), and 25 cents (one quarter). With these denominations, the `MakeChange` program gives you more than simply a set of coins adding up to 138 cents. The `MakeChange` class gives you the *smallest number of coins* that add up to 138 cents. With some minor tweaking, you can make the code work in any country's coinage. You can always get a set of coins adding up to a total. But, for the denominations of coins in some countries, you won't always get the *smallest number of coins* that add up to a total. Before 1970, England's currency included 6 pence, 12 pence, 24 pence, and 30 pence coins. With input 48, a program like the one in Listing 7-3 would have told you to add up three coins: 30 + 12 + 6 = 48. But you could do better by combining only two coins: 24 + 24 = 48.

WARNING

When two or more variables have similar types, you can create the variables with combined declarations. For example, Listing 7-3 has two combined declarations — one for the variables `quarters`, `dimes`, `nickels`, and `cents` (all of type `int`) and another for the variables `whatsLeft` and `total` (both of type `int`). But to create variables of different types, you need separate declarations. For example, to create an `int` variable named `total` and a `double` variable named `amount`, you need one declaration `int total;` and another declaration `double amount;`.

Take control of your program's output

Listing 7-3 has a call to `System.out.println()` with nothing in the parentheses. When the computer executes this statement, the cursor jumps to a new line on the screen. (I often use this statement to put a blank line in a program's output.)

The last five `println` calls in Listing 7-3 use another cute trick. In Java, you can concatenate strings with a plus sign (+). When you *concatenate* strings, you scrunch them together, one right after another. For example, the expression

```
"Barry" + " " + "Burd"
```

scrunches together `Barry`, a blank space, and `Burd`. The new scrunched-up string is (you guessed it) `Barry Burd`.

In the following statement from Listing 7-3

```
System.out.println("From " + total + " cents you get");
```

IF THINE INT OFFENDS THEE, CAST IT OUT

The run in Figure 7-6 seems artificial. Why would you start with 138 cents? Why not use the more familiar $1.38? The reason is that the number 1.38 isn't a whole number, and whole numbers are more accurate than other kinds of numbers. In fact, without whole numbers, the remainder operator isn't very useful. For example, the value of `1.38 % 0.25` is `0.1299999999999999`. All those nines are tough to work with. Imagine reading your credit card statement and seeing that you owe $0.1299999999999999. You'd probably pay $0.13 and let the credit card company keep the change. But after years of rounding numbers, the credit card company would make a fortune! Chapter 8 describes, in a bit more detail, inaccuracies that may come from using `double` values.

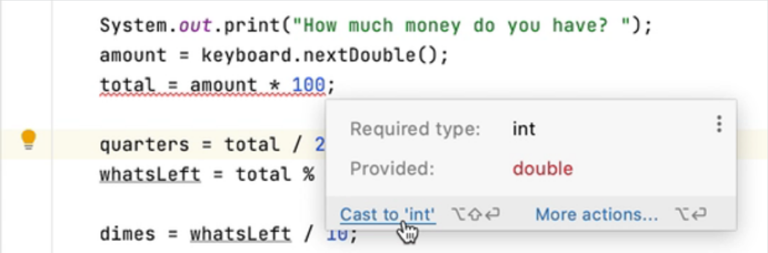
Throughout this book, I illustrate Java's `double` type with programs about money. Many authors do the same thing. But for greater accuracy, avoid using `double` values for money. Instead, you should use `int` values or use the `long` values that I describe in the last section of this chapter. Even better, look up `BigInteger` and `BigDecimal` in Java's API documentation. These `Big`*`SomethingOrOther`* types are cumbersome to use, but they provide industrial-strength numeric range and accuracy.

Now, what if you want to input `1.38` and then have the program take your 1.38 and turn it into 138 cents? How can you get your program to do this?

My first idea is to multiply 1.38 by 100:

```
//This doesn't quite work.
double amount;
int total;
amount = keyboard.nextDouble();
total = amount*100;
```

In everyday arithmetic, multiplying by 100 does the trick. But computers are fussy. With a computer, you have to be very careful when you mix `int` values and `double` values. (See the first figure in this sidebar.)



```
System.out.print("How much money do you have? ");
amount = keyboard.nextDouble();
total = amount * 100;

quarters = total / 2
whatsLeft = total %

dimes = whatsLeft / 10;
```

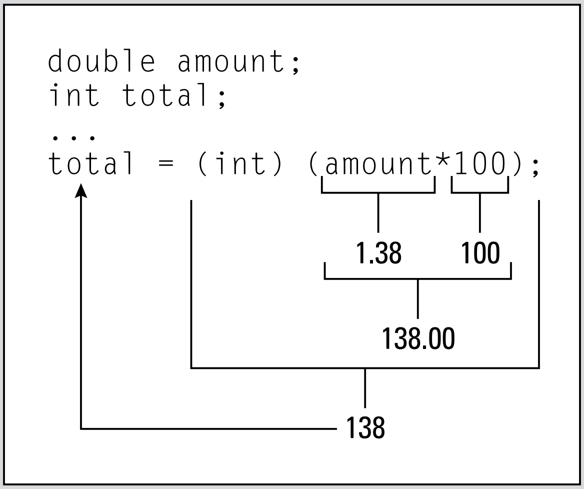
Required type: int
Provided: double

To cram a `double` value into an `int` variable, you need something called *casting*. When you *cast* a value, you essentially say, “I’m aware that I’m trying to squish a `double` value into an `int` variable. It’s a tight fit, but I want to do it anyway.”

To do casting, select IntelliJ’s `Cast to 'int'` quick fix. Casting puts the name of a type in parentheses, as follows:

```
//This works!
total = (int) (amount * 100);
```

The casting notation `(int)` turns the `double` value 138.00 into the `int` value 138, and everybody’s happy. (See the second figure in this sidebar.)



```
double amount;
int total;
...
total = (int) (amount*100);
```

the variable `total` stores an `int` value, while the literals `"From "` and `" cents you get"` are strings. To understand this, remember that the value of `total` isn’t a string of digit characters like `"138"`. Instead, `total` is an amount such as `138`. In everyday life, the difference between `"138"` and `138` is rarely important. But in computer programming, the difference is a big deal.

Fortunately, Java’s plus sign concatenates strings to numbers and numbers to strings. When you put a plus sign between two such values, you get a string, not a number. For example, in Listing 7-3, Java sees

```
"From " + total + " cents you get"
```

With the input shown in Figure 7-6, Java turns the expression into

```
"From " + 138 + " cents you get"
```

Then, Java does you a favor and replaces the number 138 with its most sensible string representation

```
"From " + "138" + " cents you get"
```

Finally, Java concatenates the three strings to give you the output you want.

```
From 138 cents you get
```

When Java does all this work, is Java being fussy about the types of its values? Yes. it is. Is this fussiness necessary? Yes, it is. Do you have to worry about this fussiness whenever you write a Java program? Probably not. Java does most of the fussing for you.

TRY IT OUT

JAVA ARITHMETIC

What's the value of each of the following expressions? Type each expression on a separate line in JShell to find out whether your answers are correct:

```
10 / 3

10 % 3

3 / 10

3 % 10

8 * 3 + 4

4 + 8 * 3

8 * (3 + 4)

34 % 5 - 2 * 2 + 21 / 5
```

VARIABLE VALUES

What's the value of each of the following variables (`a`, `b`, `c`, and so on)? Type each statement on a separate line in JShell to find out whether your answers are correct:

```
int a = 8

int b = 3

int c = b / a

int d = a / b

int e = a % b

int f = 5 + e * d - 2
```

HIRING A PLUMBER

A local plumber charges $75 to come to my house. In addition, for every hour the plumber works at my house, the plumber charges an additional $125. Write a program that inputs the number of hours that a plumber works at my house and outputs the total amount that the plumber charges.

MAKING CHANGE AGAIN

Modify the code in Listing 7-3 so that it starts by getting a number of dollars and a number of cents from the keyboard. For example, rather than type **138** (meaning 138 cents), the user types **1 38** (1 dollar and 38 cents).

HOW TALL AM I?

Where I come from, we don't use metric measurements. Instead, we measure each person's height in feet and inches. A foot is 12 inches, and I'm five-and-a-half feet tall. (My height in feet is the `double` value 5.5.) Write a program to find my height in inches. (That is, from 5.5 feet, calculate 66 inches.)

Modify the program so that it asks for the user's height in feet and then reports the person's height in inches.

Modify the program so that it asks for the user's height in feet and inches. For example, a person who's five-and-a-half feet tall types the number **5** (for five feet) followed by the number **6** (for six more inches). The program reports the person's height in inches.

HOW MANY ANNIVERSARIES?

My wife and I were married on February 29, so we have one anniversary every four years. Write a program with a variable named `years`. Based on the value of the `years` variable, the program displays the number of anniversaries we've had.

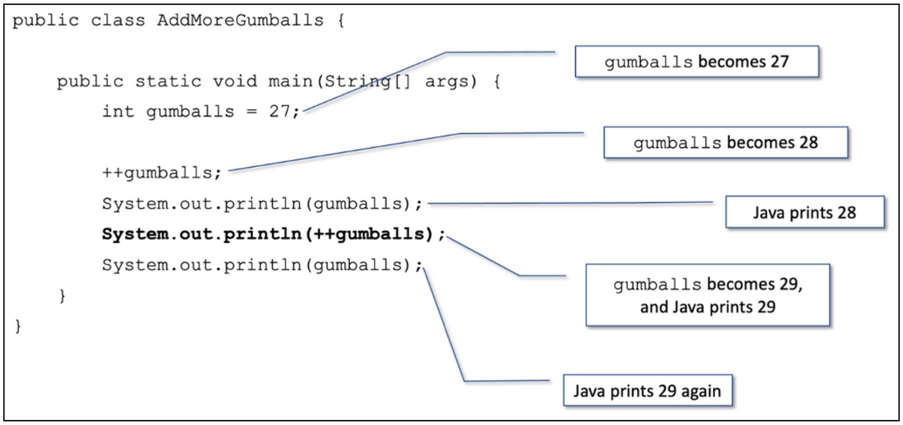
For example, if the value of `years` is `4`, the program displays the sentence `Number of anniversaries: 1`. If the value of `years` is `7`, the program still displays `Number of anniversaries: 1`. But if the value of `years` is `8`, the program displays `Number of anniversaries: 2`.

The increment and decrement operators

Java has some neat little operators that make life easier (for the computer's processor, for your brain, and for your fingers). Altogether, there are four such operators — two increment operators and two decrement operators. The increment operators add one, and the decrement operators subtract one. To see how they work, you need some examples.

Using preincrement

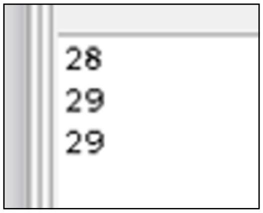
The first example is in Figure 7-7.



```
public class AddMoreGumballs {

    public static void main(String[] args) {
        int gumballs = 27;

        ++gumballs;
        System.out.println(gumballs);
        System.out.println(++gumballs);
        System.out.println(gumballs);
    }
}
```

FIGURE 7-7: Using preincrement.

A run of the program in Figure 7-7 is shown in Figure 7-8. In this horribly uneventful run, the count of gumballs is displayed three times.



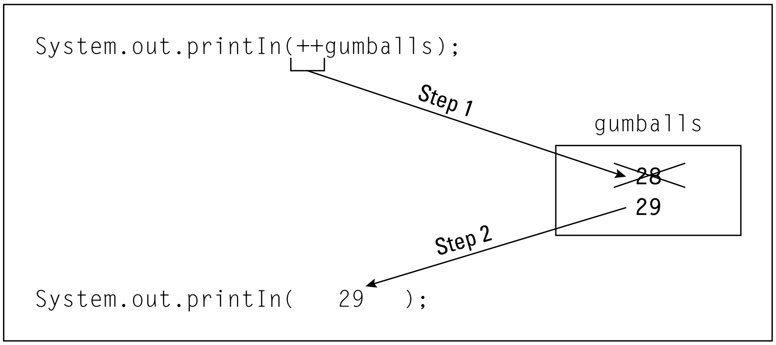
```
28
29
29
```

FIGURE 7-8: A run of the preincrement code (the code in Figure 7-7).

The double plus sign goes under two different names, depending on where you put it. When you put the ++ before a variable, the ++ is called the *preincrement* operator. In the word *preincrement,* the *pre* stands for *before.* In this setting, the word *before* has two different meanings:

- You're putting ++ before the variable.
- The computer adds 1 to the variable's value before the variable is used in any other part of the statement.

Figure 7-9 has a slow-motion, instant replay of the preincrement operator's action. In Figure 7-9, the computer encounters the `System.out.println (++gumballs)` statement. First, the computer adds 1 to `gumballs` (raising the value of `gumballs` to `29`). Then the computer executes `System.out.println`, using the new value of `gumballs (29)`.



```
System.out.printIn(++gumballs);
```

```
System.out.printIn(   29   );
```

FIGURE 7-9: The preincrement operator in action.

Using postincrement

An alternative to preincrement is *postincrement*. With postincrement, the *post* stands for *after.* The word *after* has two different meanings:

- You put ++ after the variable.
- The computer adds 1 to the variable's value after the variable is used in any other part of the statement.

Figure 7-10 shows a close-up view of the postincrement operator's action. In Figure 7-10, the computer encounters the `System.out.println(gumballs++)` statement. First, the computer executes `System.out.println`, using the old value of `gumballs (28)`. Then the computer adds 1 to `gumballs` (raising the value of `gumballs` to `29`).

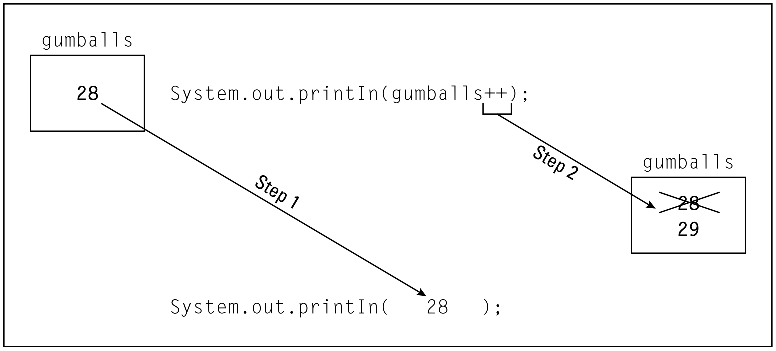
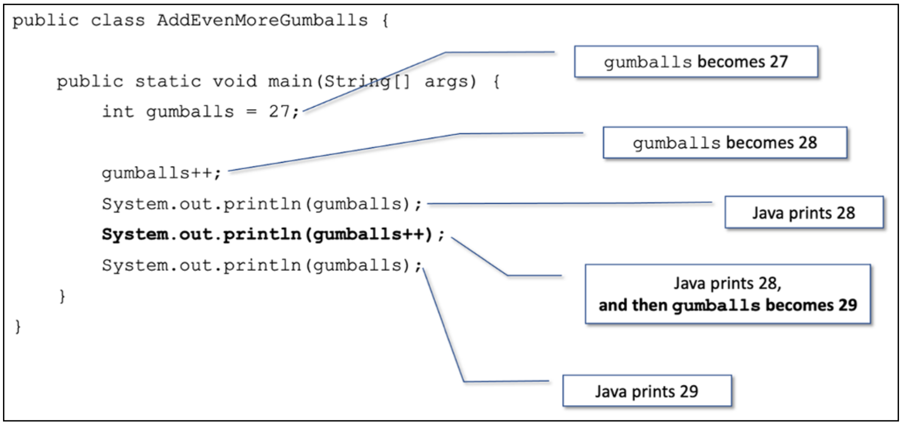

gumballs
28
System.out.println(gumballs++);
Step 1
Step 2
gumballs
~~28~~
29
System.out.println(28   );

FIGURE 7-10: The postincrement operator in action.

Look at the bold line of code in Figure 7-11. The computer prints the old value of `gumballs (28)` on the screen. Only after printing this old value does the computer add 1 to `gumballs` (raising the `gumballs` value from `28` to `29`).



```
public class AddEvenMoreGumballs {

    public static void main(String[] args) {
        int gumballs = 27;

        gumballs++;
        System.out.println(gumballs);
        System.out.println(gumballs++);
        System.out.println(gumballs);
    }
}
```

FIGURE 7-11: Using postincrement.

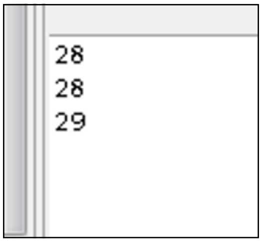
REMEMBER

With `System.out.println(gumballs++)`, the computer adds 1 to `gumballs` *after* printing the old value that `gumballs` already had.

A run of the code in Figure 7-11 is shown in Figure 7-12. Compare Figure 7-12 with the run in Figure 7-8.

- With preincrement in Figure 7-8, the second number that's displayed is `29`.
- With postincrement in Figure 7-12, the second number that's displayed is `28`.

  In Figure 7-12, the number `29` doesn't show up on the screen until the end of the run, when the computer executes one last `System.out.println(gumballs)`.



```
28
28
29
```

FIGURE 7-12: A run of the postincrement code (the code in Figure 7-11).

TIP

Are you trying to decide between using preincrement or postincrement? Ponder no longer. Most programmers use postincrement. In a typical Java program, you often see things like `gumballs++`. You seldom see things like `++gumballs`.

In addition to preincrement and postincrement, Java has two operators that use `--`:

- With *predecrement* (`--gumballs`), the computer subtracts 1 from the variable's value before the variable is used in the rest of the statement.
- With *postdecrement* (`gumballs--`), the computer subtracts 1 from the variable's value after the variable is used in the rest of the statement.

TRY IT OUT

EXPLORE PREINCREMENT AND POSTINCREMENT USING JSHELL

Type the boldface text, one line after another, into JShell, and see how JShell responds:

```
int i = 8

i++

i

i

i++

i

++i

I
```

STATEMENTS AND EXPRESSIONS

Any part of a computer program that has a value is called an *expression*. If you write

```
gumballs = 30;
```

then `30` is an expression (an expression whose value is the quantity 30). If you write

```
amount = 5.95 + 25.00;
```

then `5.95 + 25.00` is an expression (because 5.95 + 25.00 has the value 30.95). If you write

```
gumballsPerKid = gumballs / kids;
```

then `gumballs / kids` is an expression. (The value of the expression `gumballs / kids` depends on whatever values the variables `gumballs` and `kids` have when the statement with the expression in it is executed.)

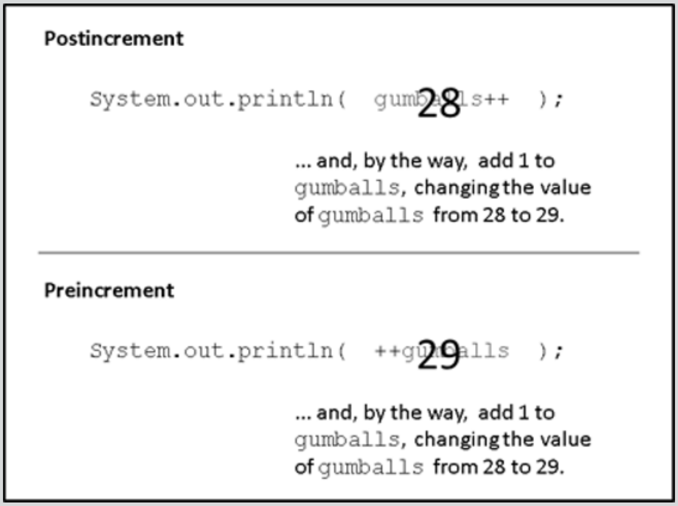
This brings us to the subject of the pre- and postincrement and decrement operators. You can think about these operators in two ways: the way everyone understands it and the right way. The way I explain it in most of this section (in terms of time, with *before* and *after*) is the way everyone understands the concept. Unfortunately, the way everyone understands the concept isn't really the right way. When you see ++ or −−, you can think in terms of time sequence. But occasionally some programmer uses ++ or −− in a convoluted way, and the notions of before and after break down. So if you're ever in a tight spot, you should think about these operators in terms of statements and expressions.

First, remember that a statement tells the computer to do something, and an expression has a value. (Statements are described in Chapter 4, and expressions are described earlier in this sidebar.) Which category does `gumballs++` belong to? The surprising answer is both. The Java code `gumballs++` is both a statement and an expression.

Suppose that, before executing the code `System.out.println(gumballs++)`, the value of `gumballs` is 28:

- As a statement, `gumballs++` tells the computer to add 1 to `gumballs`.
- As an expression, the value of `gumballs++` is 28, not 29.

  So, even though `gumballs` gets 1 added to it, the code `System.out.println(gumballs++)` really means `System.out.println(28)`. (See the figure in this sidebar.)



Postincrement
System.out.println(gumballs++);
28
... and, by the way, add 1 to gumballs, changing the value of gumballs from 28 to 29.
Preincrement
System.out.println(++gumballs);
29
... and, by the way, add 1 to gumballs, changing the value of gumballs from 28 to 29.

Now, almost everything you've just read about gumballs++ is true about ++gumballs. The only difference is that, as an expression, ++gumballs behaves in a more intuitive way. Suppose that before executing the code System.out.println(++gumballs), the value of gumballs is 28:

- As a statement, ++gumballs tells the computer to add 1 to gumballs.
- As an expression, the value of ++gumballs is 29.

So, with System.out.println(++gumballs), the variable gumballs gets 1 added to it, and the code System.out.println(++gumballs) really means System.out.println(29).

EXPLORE PREINCREMENT AND POSTINCREMENT IN A JAVA PROGRAM

Before you run the following code, try to predict what the code's output will be. Then run the code to find out whether your prediction is correct:

```
public class Main {

  public static void main(String[] args) {
    inti = 10;
    System.out.println(i++);
    System.out.println(--i);
    --i;
    i--;
    System.out.println(i);
    System.out.println(++i);
    System.out.println(i--);
    System.out.println(i);
  }
}
```

Assignment operators

If you've read the previous section — the section about operators that add 1 — you may be wondering whether you can manipulate these operators to add 2 or add 5 or add 1000000. Can you write `gumballs++++` and still call yourself a Java programmer? Well, you can't. If you try it, IntelliJ will give you an error message:

```
Variable expected
```

If you don't use IntelliJ, you may see a different error message:

```
error: unexpected type
      gumballs++++;
              ^
required: variable
found:    value
```

IntelliJ or no IntelliJ, the bottom line is the same: Namely, your code contains an error, and you have to fix it.

How can you add values other than 1? As luck would have it, Java has plenty of assignment operators you can use. With an *assignment operator*, you can add, subtract, multiply, or divide by anything you want. You can do other cool operations, too.

For example, you can add 1 to the `kids` variable by writing

```
kids += 1;
```

Is this better than `kids++` or `kids = kids + 1`? No, it's not better. It's just an alternative. But you can add 5 to the `kids` variable by writing

```
kids += 5;
```

You can't easily add 5 with preincrement or postincrement. And what if the kids get stuck in an evil scientist's cloning machine? The statement

```
kids *= 2;
```

multiplies the number of `kids` by 2.

With the assignment operators, you can add, subtract, multiply, or divide a variable by any number. The number doesn't have to be a literal. You can use a number-valued expression on the right side of the equal sign:

```
double amount = 5.95;
double shippingAndHandling = 25.00, discount = 0.15;
amount += shippingAndHandling;
amount -= discount * 2;
```

The preceding code adds 25.00 (`shippingAndHandling`) to the value of `amount`. Then the code subtracts 0.30 (`discount * 2`) from the value of `amount`. How generous!

CROSS REFERENCE

If the word *literal* doesn't ring any bells for you, refer to Chapter 4.

TRY IT OUT

EXPERIMENT WITH ASSIGNMENT OPERATORS

Before you run the following code, try to predict what the code's output will be. Then run the code to find out whether your prediction is correct:

```
public class Main {

  public static void main(String[] args) {
    inti = 10;

    i += 2;
    i -= 5;
    i *= 6;

    System.out.println(i);
    System.out.println(i += 3);
    System.out.println(i /= 2);
  }
}
```

MAKING CHANGE YET AGAIN

In addition to the assignment operators that I describe in this section, Java also has a %= operator. The %= operator does for remainders what the += operator does for addition. Modify the code in Listing 7-3 so that it uses the %= assignment operator wherever possible.

Size Matters

Here are today's new vocabulary words:

> **foregift** (fore-gift) *n.* A premium that a lessee pays to the lessor upon the taking of a lease.
>
> **hereinbefore** (here-in-be-fore) *adv.* In a previous part of this document.

Now imagine yourself scanning some compressed text. In this text, all blanks have been removed to conserve storage space. You come upon the following sequence of letters:

> hereinbeforegiftedit

The question is, what do these letters mean? If you knew each word's length, you could answer the question:

> here in be foregift edit
>
> hereinbefore gifted it
>
> herein before gift Ed it

A computer faces the same kind of problem. When a computer stores several numbers in memory or on a disk, the computer doesn't put blank spaces between the numbers. So imagine that a small chunk of the computer's memory looks like the stuff in Figure 7-13. (The computer works exclusively with zeros and ones, but Figure 7-13 uses ordinary digits. With ordinary digits, it's easier to see what's going on.)

FIGURE 7-13: Storing the digits 4221.

4	2	2	1

What number or numbers are stored in Figure 7-13? Is it two numbers, 42 and 21? Or is it one number, 4,221? And what about storing four numbers, 4, 2, 2, and 1? It all depends on the amount of space each number consumes.

Imagine a variable that stores the number of paydays in a month. This number never exceeds 31. You can represent this small number with just eight zeros and ones. But what about a variable that counts stars in the universe? That number could easily be more than a trillion, and to represent 1 trillion accurately, you need 64 zeros and ones.

At this point, Java comes to the rescue. Java has four types of whole numbers. Just as in Listing 7-1, I declare

```
int gumballsPerKid;
```

I can also declare

```
byte paydaysInAMonth;
short sickDaysDuringYourEmployment;
long numberOfStars;
```

Each of these types (`byte`, `short`, `int`, and `long`) has its own range of possible values. (See Table 7-1.)

TABLE 7-1:

Java's Primitive Numeric Types

Type Name	Range of Values
Whole Number Types	
`byte`	–128 to 127
`short`	–32768 to 32767
`int`	–2147483648 to 2147483647
`long`	–9223372036854775808 to 9223372036854775807
Decimal Number Types	
`float`	-3.4×10^{38} to 3.4×10^{38}
`double`	-1.8×10^{308} to 1.8×10^{308}

Java has two types of *decimal* numbers (numbers with digits to the right of the decimal point). Just as in Listing 6-1 (over in Chapter 6), I declare

```
double amount;
```

I can also declare

```
float monthlySalary;
```

Given the choice between `double` and `float`, I always choose `double`. A variable of type `double` has a greater possible range of values and much greater accuracy. (See Table 7-1.)

Table 7-1 lists six of Java's *primitive* types (also known as *simple* types). Java has only eight primitive types, so only two of Java's primitive types are missing from Table 7-1.

Chapter 8 describes the two remaining primitive types. Chapter 13 introduces types that aren't primitive.

As a beginning programmer, you don't have to choose among the types in Table 7-1. Just use `int` for whole numbers and `double` for decimal numbers. If, in your travels, you see something like `short` or `float` in someone else's program, just remember the following:

- The types `byte`, `short`, `int`, and long represent whole numbers.
- The types `float` and `double` represent decimal numbers.

Most of the time, that's all you need to know.

IN THIS CHAPTER

- » Working with characters
- » Dealing with "true" or "false" values
- » Rounding out your knowledge of Java's primitive types

Chapter **8**

Numbers? Who Needs Numbers?

I don't particularly like fax machines. They're so inefficient. Send a short fax and what do you have? You have two slices of a tree — one at the sending end and another at the receiving end. You also have millions of dots — dots that scan tiny little lines across the printed page. The dots distinguish patches of light from patches of darkness. What a waste!

Compare a fax with an email message. Using email, I can send a 25-word contest entry with just 2,500 zeros and ones, and I don't waste any paper. Best of all, an email message doesn't describe light dots and dark dots. An email message contains codes for each of the letters — a short sequence of zeros and ones for the letter *A*, a different sequence of zeros and ones for the letter *B*, and so on. What could be simpler?

Now imagine sending a one-word fax. The word is *true*, which is understood to mean, "True, I accept your offer to write *Beginning Programming with Java For Dummies*, 6th Edition." A fax with this message sends a picture of the four letters t-r-u-e, with fuzzy lines where dirt gets on the paper and little white dots where the cartridge runs short on toner.

But really, what's the essence of the "true" message? There are just two possibilities, aren't there? The message could be "true" or "false," and to represent those possibilities, I need very little fanfare. How about 0 for "false" and 1 for "true"?

> They ask, "Do you accept our offer to write *Beginning Programming with Java For Dummies,* 6th Edition?"
>
> "1," I reply.

Too bad I didn't think of that a few months ago. Anyway, this chapter deals with letters, truth, falsehood, and other such things.

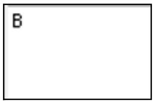
A Brief Character Study

In Chapters 6 and 7, you store numbers in all your variables. That's fine, but there's more to life than numbers. For example, I wrote this book with a computer, and this book contains thousands and thousands of nonnumeric things called *characters.*

The Java type that's used to store characters is *char.* Listing 8-1 has a simple program that uses the `char` type, and a run of the Listing 8-1 program is shown in Figure 8-1.

LISTING 8-1: Using the char Type

```
public class LowerToUpper {

    public static void main(String[] args) {
        char smallLetter, bigLetter;

        smallLetter = 'b';
        bigLetter = Character.toUpperCase(smallLetter);
        System.out.println(bigLetter);
    }
}
```



```
B
```

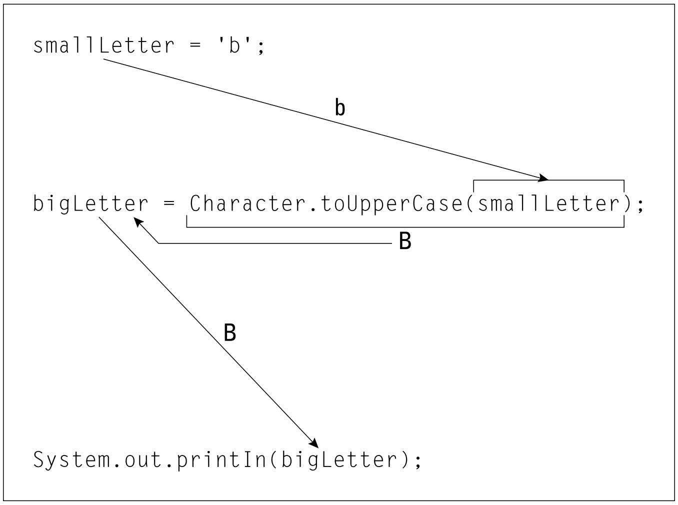
FIGURE 8-1: Exciting program output!

In Listing 8-1, the first assignment statement stores the letter `b` in the `smallLetter` variable. In that statement, notice how `b` is surrounded by single quote marks (`'  '`). In a Java program, every `char` literal starts and ends with a single quote mark.

TECHNICAL STUFF

When you surround a letter with quote marks, you tell the computer that the letter isn't a variable name. For example, in Listing 8-1, the incorrect statement `smallLetter = b` would tell the computer to look for a variable named `b`. Because there's no variable named `b`, IntelliJ would display `b` in an alarming, red color.

In the second assignment statement of Listing 8-1, the program calls an API method whose name is `Character.toUpperCase`. The method `Character.toUpperCase` does what its name suggests — the method produces the uppercase equivalent of a lowercase letter. In Listing 8-1, this uppercase equivalent (the letter `B`) is assigned to the variable `bigLetter`, and the `B` that's in `bigLetter` is printed on the screen, as illustrated in Figure 8-2.



```
smallLetter = 'b';

bigLetter = Character.toUpperCase(smallLetter);

System.out.println(bigLetter);
```

FIGURE 8-2: The action in Listing 8-1.

REMEMBER

When the computer displays a `char` value on the screen, the computer doesn't surround the character with single quote marks.

I digress . . .

A while ago, I wondered what would happen if I called the `Character.toUpperCase` method and fed the method a character that isn't lowercase to begin with. I yanked out the Java API documentation, but I found no useful information. The documentation said that `toUpperCase` "converts the character argument to uppercase using case mapping information from the UnicodeData file." Thanks, but that's not useful to me.

Silly as it seems, I asked myself what I'd do if I were the toUpperCase method. What would I say if someone handed me a capital R and told me to capitalize that letter? I'd say, "Take back your stinking capital R." In the lingo of computing, I'd send that person an error message. So I wondered whether I'd get an error message if I applied Character.toUpperCase to the letter R.

I tried it. I cooked up the experiment in Listing 8-2.

LISTING 8-2: Investigating the Behavior of toUpperCase

```
public class MyExperiment {

    public static void main(String[] args) {
        char smallLetter, bigLetter;

        smallLetter = 'R';
        bigLetter = Character.toUpperCase(smallLetter);
        System.out.println(bigLetter);

        smallLetter = '3';
        bigLetter = Character.toUpperCase(smallLetter);
        System.out.println(bigLetter);
    }
}
```

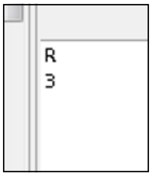
In my experiment, I didn't mix chemicals and blow things up. Here's what I did instead:

- **I assigned 'R' to smallLetter.**

  The toUpperCase method took the uppercase R and gave me back another uppercase R. (See Figure 8-3.) I got no error message. This told me what the toUpperCase method does with a letter that's already uppercase. The method does nothing.

- **I assigned '3' to smallLetter.**

  The toUpperCase method took the digit 3 and gave me back the same digit 3. (See Figure 8-3.) I got no error message. This told me what the toUpperCase method does with a character that's not a letter. It does nothing — zip, zilch, bupkis.



```
R
3
```

FIGURE 8-3: Running the code in Listing 8-2.

I write about this experiment to make an important point. When you don't understand something about computer programming, it often helps to write a test program. Make up an experiment and see how the computer responds.

I guessed that handing a capital `R` to the `toUpperCase` method would give me an error message, but I was wrong. See? The answers to questions aren't handed down from heaven. The people who created the Java API made decisions. They made some obvious choices, and they also made some unexpected choices. No one knows everything about Java's features, so don't expect to cram all the answers into your head.

The Java documentation is great, but for every question that the documentation answers, it ignores three other questions. So be bold. Don't be afraid to tinker. Write lots of short, experimental programs. You can't break the computer, so play tough with it. Your inquisitive spirit will always pay off.

One character only, please

A `char` variable stores only one character. So if you're tempted to write the following statements

```
char smallLetters;
smallLetters = 'barry'; //Don't do this
```

please resist the temptation. You can't store more than one letter at a time in a `char` variable, and you can't put more than one letter between a pair of single quotes. If you're trying to store words or sentences (not just single letters), then you need to use double quote marks, as in this statement from a listing in Chapter 7:

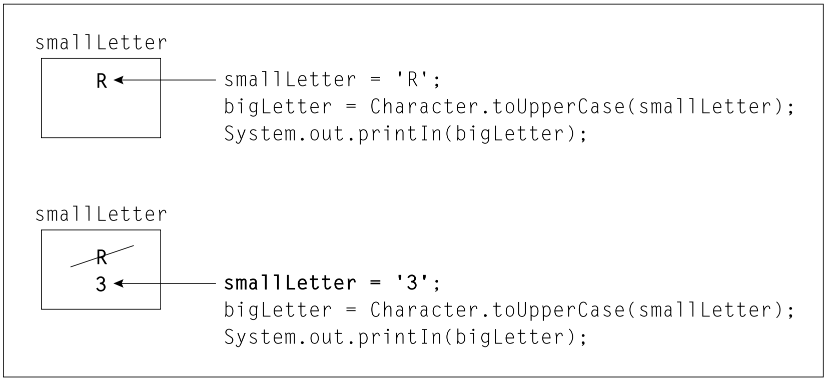
```
System.out.print("How many gumballs? How many kids? ");
```

The text in double quote marks is an example of a string. For an introduction to strings, refer to Chapter 4. For a more careful look at Java's `String` type, see Chapter 14.

Variables and recycling

In Listing 8-2, I use smallLetter twice, and I use bigLetter twice. That's why they call these things *variables*. First, the value of smallLetter is R. Later, I vary the value of smallLetter so that the value of smallLetter becomes 3.

When I assign a new value to smallLetter, the old value of smallLetter gets obliterated. For example, in Figure 8-4, the second smallLetter assignment puts 3 into smallLetter. When the computer executes this second assignment statement, the old value R is gone.



```
smallLetter = 'R';
bigLetter = Character.toUpperCase(smallLetter);
System.out.printIn(bigLetter);
```

```
smallLetter = '3';
bigLetter = Character.toUpperCase(smallLetter);
System.out.printIn(bigLetter);
```

FIGURE 8-4: Varying the value of smallLetter.

Is that okay? Can you afford to forget the value that smallLetter once had? Yes, in Listing 8-2, it's okay. After you've assigned a value to bigLetter with the statement

```
bigLetter = Character.toUpperCase(smallLetter);
```

you can forget all about the existing smallLetter value. You don't need to do this:

```
// This code is cumbersome.
// The extra variables are unnecessary.
char smallLetter1, bigLetter1;
char smallLetter2, bigLetter2;

smallLetter1 = 'R';
bigLetter1 = Character.toUpperCase(smallLetter1);
System.out.println(bigLetter1);

smallLetter2 = '3';
bigLetter2 = Character.toUpperCase(smallLetter2);
System.out.println(bigLetter2);
```

You don't need to store the old and new values in separate variables. Instead, you can reuse the variables `smallLetter` and `bigLetter` as in Listing 8-2.

This reuse of variables doesn't save you from a lot of extra typing. It doesn't save much memory space, either. But reusing variables keeps the program uncluttered. When you look at Listing 8-2, you can see at a glance that the code has two parts, and you see that both parts do roughly the same thing.

The code in Listing 8-2 is simple and manageable. In such a small program, simplicity and manageability don't matter much. But in a large program, it helps to think carefully about the use of each variable.

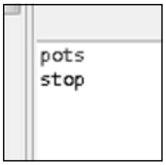
When not to reuse a variable

The previous section discusses the reuse of variables to make a program slick and easy to read. This section shows you the flip side. In this section, the problem at hand forces you to create new variables.

Suppose that you're writing code to reverse the letters in a four-letter word. You store each letter in its own, separate variable. Listing 8-3 shows the code, and Figure 8-5 shows the code in action.

LISTING 8-3: Making a Word Go Backward

```
import java.util.Scanner;

public class ReverseWord {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        char c1, c2, c3, c4;

        c1 = keyboard.findWithinHorizon(".", 0).charAt(0);
        c2 = keyboard.findWithinHorizon(".", 0).charAt(0);
        c3 = keyboard.findWithinHorizon(".", 0).charAt(0);
        c4 = keyboard.findWithinHorizon(".", 0).charAt(0);

        System.out.print(c4);
        System.out.print(c3);
        System.out.print(c2);
        System.out.print(c1);
        System.out.println();

        keyboard.close();
    }
}
```



```
pots
stop
```

FIGURE 8-5: Stop those pots!

The trick in Listing 8-3 is as follows:

- Assign values to variables `c1`, `c2`, `c3`, and `c4` in that order.
- Display these variables' values on the screen in reverse order: `c4`, `c3`, `c2`, and then `c1`, as illustrated in Figure 8-6.

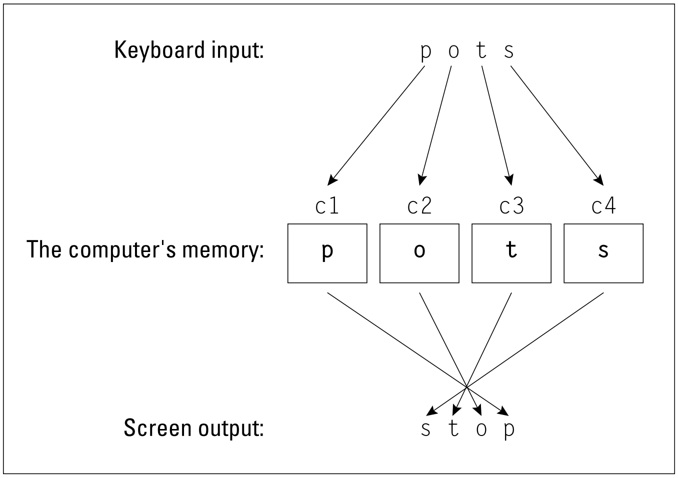


FIGURE 8-6: Using four variables.

If you don't use four separate variables, you don't get the result that you want. For example, imagine that you store characters in only one variable. You run the program and type the word `pots`. When it's time to display the word in reverse, the computer remembers the final `s` in the word `pots`. But the computer doesn't remember the `p`, the `o`, or the `t`, as shown in Figure 8-7.

I wish I could give you 12 simple rules to help you decide when and when not to reuse variables. The problem is, I can't. It all depends on what you're trying to accomplish. So, how do you figure out on your own when and when not to reuse variables? Like the guy says to the fellow who asks how to get to Carnegie Hall, "Practice, practice, practice."

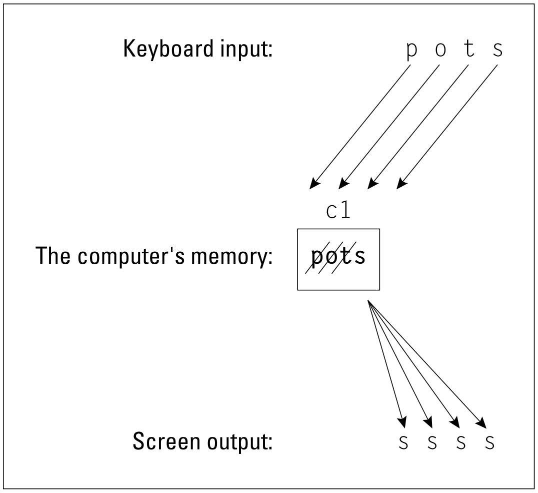


FIGURE 8-7: Getting things wrong because you used only one variable.

Reading characters

The people who created Java's `Scanner` class didn't create a `next` method for reading a single character. So, to input a single character, I paste two Java API methods together. I use the `findWithinHorizon` and `charAt` methods.

Table 5-1 (over in Chapter 5) introduces this `findWithinHorizon(".", 0).charAt(0)` technique for reading a single input character, and Listing 8-3 uses the technique to read one character at a time. (In fact, Listing 8-3 uses the technique four times to read four individual characters.)

Notice the format for the input shown earlier, in Figure 8-5. To enter the characters in the word `pots`, I type four letters, one after another, with no blank spaces between the letters and no quote marks. The `findWithinHorizon(".", 0).charAt(0)` technique works that way, but don't blame me or my technique. Other developers' character-reading methods work the same way. No matter whose methods you use, reading a character differs from reading a number. Here's how:

- **With methods like `nextDouble` and `nextInt`, you type blank spaces between numbers.**

  If I type **80 6**, then two calls to `nextInt` read the number `80`, followed by the number `6`. If I type **806**, then a single call to `nextInt` reads the number `806` (eight hundred six), as illustrated in Figure 8-8.

WHAT'S BEHIND ALL THIS FINDWITHINHORIZON NONSENSE?

Without wallowing in too much detail, here's how the `findWithinHorizon(".", 0).charAt(0)` technique works:

Java's `findWithinHorizon` method looks for things in the input. The things the method finds depend on the stuff you put in parentheses. For example, a call to `findWithinHorizon("\\d\\d\\d", 0)` looks for a group consisting of three digits. With the following line of code

```
System.out.println(keyboard.findWithinHorizon("\\d\\d\\d", 0));
```

I can type

```
Testing 123 Testing Testing
```

and the computer responds by displaying

```
123
```

In the call `findWithinHorizon("\\d\\d\\d", 0)`, each `\\d` stands for a single digit. This `\\d` business is one of many abbreviations in special code called *regular expressions*.

Now, here's something strange. In the world of regular expressions, a dot stands for any character at all. (That is, a dot stands for "any character, not necessarily a dot.") So `findWithinHorizon(".", 0)` tells the computer to find the next character of any kind that the user types on the keyboard. When you're trying to input a single character, `findWithinHorizon(".", 0)` is mighty useful.

In the call `findWithinHorizon("\\d\\d\\d", 0)`, the `0` tells `findWithinHorizon` to keep searching until the end of the input. This value `0` is a special case because anything other than `0` limits the search to a certain number of characters. (That's why the method name contains the word *horizon*. The *horizon* is as far as the method sees.) Here are a few examples:

- With the same input `Testing 123 Testing Testing`, the call `findWithinHorizon("\\d\\d\\d", 9)` returns `null`. It returns `null` because the first nine characters of the input (the characters `Testing 1` — seven letters, a blank space, and a digit) don't contain three consecutive digits. These nine characters don't match the pattern `\\d\\d\\d`.

- With the same input, the call `findWithinHorizon("\\d\\d\\d", 10)` also returns `null`. It returns `null` because the first ten characters of the input (the characters `Testing 12`) don't contain three consecutive digits.
- With the same input, the call `findWithinHorizon("\\d\\d\\d", 11)` returns `123`. It returns `123` because the first 11 characters of the input (the characters `Testing 123`) contain these three consecutive digits.
- With the input `A57B442123 Testing`, the call `findWithinHorizon("\\d\\d\\d", 12)` returns `442`. It returns `442` because, among the first 12 characters of the input (the characters `A57B442123 Test`), the first sequence consisting of three consecutive digits is the sequence `442`.

But wait! To grab a single character from the keyboard, I call `findWithinHorizon(".", 0).charAt(0)`. What's the role of `charAt(0)` in reading a single character? Unfortunately, any `findWithinHorizon` call behaves as though it's finding a bunch of characters, not just a single character. Even when you call `findWithinHorizon(".", 0)` and the computer fetches just one letter from the keyboard, the Java program treats that letter as one of possibly many input characters.

The call to `charAt(0)` takes care of the multicharacter problem. This `charAt(0)` call tells Java to pick the initial character from any of the characters that `findWithinHorizon` fetches.

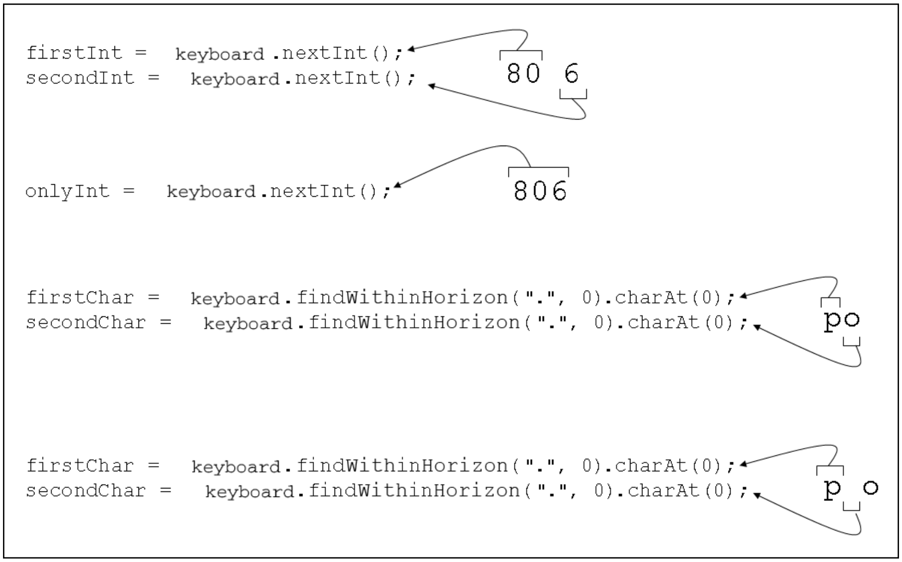
Yes, it's complicated. And yes, I don't like having to explain it. But no, you don't have to understand any of the details in this sidebar. Just read the details if you want to read them and skip the details if you don't care.

» **With `findWithinHorizon(".", 0).charAt(0)`, you don't type blank spaces between characters.**

If I type **po**, then two successive calls to `findWithinHorizon(".", 0).charAt(0)` read the letter `p`, followed by the letter `o`. If I type **p o**, then two calls to `findWithinHorizon(".", 0).charAt(0)` read the letter `p`, followed by a blank space character. (Yes, the blank space is a character!) Again, see Figure 8-8.

REMEMBER

To represent a lone character in the text of a computer program, you surround the character with single quote marks. But, when you type a character as part of a program's input, you don't surround the character with quote marks.



```
firstInt =  keyboard.nextInt();
secondInt =  keyboard.nextInt();

onlyInt =  keyboard.nextInt();

firstChar =  keyboard.findWithinHorizon(".", 0).charAt(0);
secondChar =  keyboard.findWithinHorizon(".", 0).charAt(0);

firstChar =  keyboard.findWithinHorizon(".", 0).charAt(0);
secondChar =  keyboard.findWithinHorizon(".", 0).charAt(0);
```

FIGURE 8-8: Reading numbers and characters.

WARNING

Suppose that your program calls `nextInt` and then `findWithinHorizon(".", 0).charAt(0)`. If you type **80x** on the keyboard, you get an error message. (The message says `InputMismatchException`. The `nextInt` method expects you to type a blank space after each `int` value.) Now, what happens if, instead of typing **80x**, you type **80 x** on the keyboard? Then the program gets `80` for the `int` value, followed by a blank space for the character value. For the program to get the `x`, the program has to call `findWithinHorizon(".", 0).charAt(0)` one more time. It seems wasteful, but it makes sense in the long run.

TRY IT OUT

WHAT'S IN A NAME?

In addition to its `Character.toUpperCase` method, Java has a `Character.toLowerCase` method. With that in mind, write a program that reads a three-letter word and outputs the word as it's capitalized when it's a person's name. For example, if the program reads the letters `ann`, the program outputs `Ann`. If the program inputs `BoB`, the program outputs `Bob`.

ARRANGEMENTS OF LETTERS

Write a program that reads three letters from the keyboard and outputs all possible arrangements of the three letters. For example, if the program reads the letters

```
box
```

the program outputs

```
box
bxo
obx
oxb
xbo
xob
```

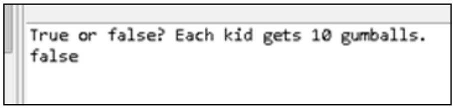
The Moment of Truth (and Falsehood)

I'm in big trouble. I have 140 gumballs, and 15 kids are running around and screaming in my living room. They're screaming because each kid wants 10 gumballs, and they're running because that's what kids do in a crowded living room. I need a program that tells me whether I can give 10 gumballs to each kid.

I need a variable of type *boolean.* A boolean variable stores one of two values — true or false (true, I can give ten gumballs to each kid; or false, I can't give ten gumballs to each kid). Anyway, the kids are going berserk, so I've written a short program and put it in Listing 8-4. The output of the program is shown in Figure 8-9.

LISTING 8-4: Using the boolean Type

```
public class CanIKeepKidsQuiet {

    public static void main(String[] args) {
        int gumballs;
        int kids;
        int gumballsPerKid;
        boolean eachKidGetsTen;

        gumballs = 140;
        kids = 15;
        gumballsPerKid = gumballs / kids;

        System.out.print("True or false? ");
        System.out.println("Each kid gets 10 gumballs.");
        eachKidGetsTen = gumballsPerKid >= 10;
        System.out.println(eachKidGetsTen);
    }
}
```

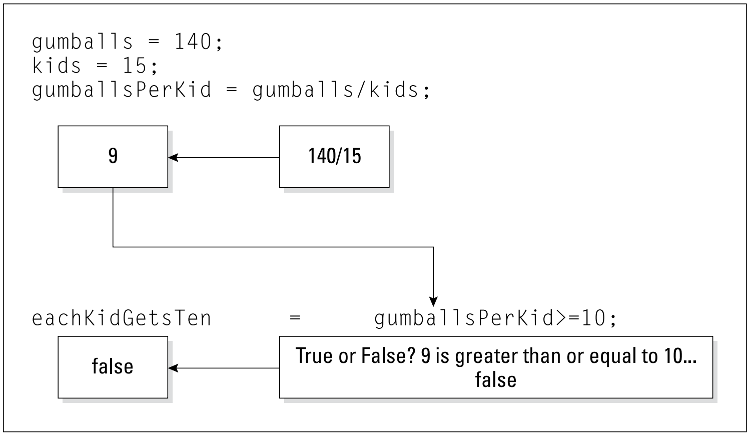


```
True or false? Each kid gets 10 gumballs.
false
```

FIGURE 8-9: Oh, no!

In Listing 8-4, the variable `eachKidGetsTen` is of type `boolean`. So the value stored in the `eachKidGetsTen` variable can be either `true` or `false`. (I can't store a number or a character in the `eachKidGetsTen` variable.)

To find a value for the variable `eachKidGetsTen`, the program checks to see whether `gumballsPerKid` is greater than or equal to ten. (The symbols `>=` stand for "greater than or equal to." What a pity! There's no ≥ key on the standard computer keyboard.) Because `gumballsPerKid` is only nine, `gumballsPerKid >= 10` is false. So `eachKidGetsTen` becomes `false`. Yikes! The kids will tear the house apart! (Before they do, take a look at Figure 8-10.)



```
gumballs = 140;
kids = 15;
gumballsPerKid = gumballs/kids;
```

```
eachKidGetsTen      =      gumballsPerKid>=10;
```

FIGURE 8-10: Assigning a value to the `eachKidGetsTen` variable.

Expressions and conditions

In Listing 8-4, the code `gumballsPerKid >= 10` is an expression. The expression's value depends on the value stored in the variable `gumballsPerKid`. On a bad day, the value of `gumballsPerKid >= 10` is `false`. So the variable `eachKidGetsTen` is assigned the value `false`.

An expression like `gumballsPerKid >= 10`, whose value is either `true` or `false`, is sometimes called a *condition.*

TECHNICAL STUFF

Values like `true` and `false` may look as though they contain characters, but they really don't. Internally, the Java Virtual Machine doesn't store `boolean` values with the letters t-r-u-e or f-a-l-s-e. Instead, the JVM stores codes, like 0 for false and 1 for true. When the computer displays a `boolean` value (as in `System.out.println(eachKidGetsTen)`), the Java Virtual Machine converts a code like 0 into the five-letter word `false`.

Comparing numbers; comparing characters

In Listing 8-4, I compare a variable's value with the number 10. I use the `>=` operator in the expression

```
gumballsPerKid >= 10
```

Of course, the greater-than-or-equal-to comparison gets you only so far. Table 8-1 shows you the operators you can use to compare things with one another.

TABLE 8-1: Comparison Operators

Operator Symbol	Meaning	Example
`==`	is equal to	`myGuess == winningNumber`
`!=`	is not equal to	`5 != numberOfCows`
`<`	is less than	`strikes < 3`
`>`	is greater than	`numberOfBoxtops > 1000`
`<=`	is less than or equal to	`lowNumber + highNumber <= 25`
`>=`	is greater than or equal to	`gumballsPerKid >= 10`

With the operators in Table 8-1, you can compare both numbers and characters.

WARNING

Notice the double equal sign in the first row of Table 8-1. Don't try to use a single equal sign to compare two values. The expression `myGuess = winningNumber` (with a single equal sign) doesn't compare `myGuess` with `winningNumber`. Instead, `myGuess = winningNumber` changes the value of `myGuess`. (It assigns the value of `winningNumber` to the variable `myGuess`.)

You can compare other things (besides numbers and characters) with the `==` and `!=` operators. But when you do, you have to be careful. For more information, see Chapter 14.

Comparing numbers

Nothing is more humdrum than comparing numbers. "True or false? Five is greater than or equal to ten." False. Five is neither greater than nor equal to ten. See what I mean? Bo-ring.

Comparing whole numbers is an open-and-shut case. But unfortunately, when you compare decimal numbers, there's a wrinkle. Take a program for converting from Celsius to Fahrenheit. Wait! Don't take just any such program; take the program in Listing 8-5.

LISTING 8-5: It's Warm and Cozy in Here

```
import java.util.Scanner;

public class CelsiusToFahrenheit {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        double celsius, fahrenheit;

        System.out.print("Enter the Celsius temperature: ");
        celsius = keyboard.nextDouble();

        fahrenheit = 9.0 / 5.0 * celsius + 32.0;

        System.out.print("Room temperature? ");
        System.out.println(fahrenheit == 69.8);

        keyboard.close();
    }
}
```

If you run the code in Listing 8-5 and input the number 21, the computer finds the value of `9.0 / 5.0 * 21 + 32.0`. Believe it or not, you want to check the computer's answer. (Who knows? Maybe the computer gets it wrong!) You need to do some arithmetic, but please don't reach for your calculator. A calculator is just a small computer, and machines of that kind stick up for one another. To check the computer's work, you need to do the arithmetic by hand. What? You say you're math-phobic? Well, don't worry. I've done all the math in Figure 8-11.

If you do the arithmetic by hand, the value you get for `9.0 / 5.0 * 21 + 32.0` is exactly 69.8. So run the code in Listing 8-5 and give `celsius` the value `21`. You should get `true` when you display the value of `fahrenheit == 69.8`, right?

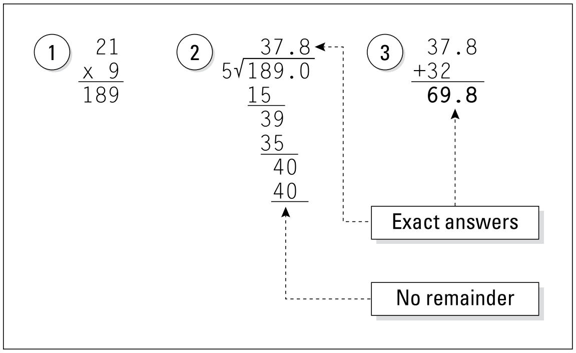
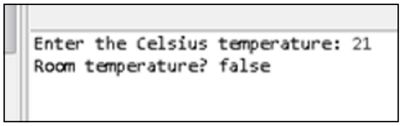


FIGURE 8-11: The Fahrenheit temperature is exactly 69.8.

Well, no. Take a look at the run in Figure 8-12. When the computer evaluates `fahrenheit == 69.8`, the value turns out to be `false`, not `true`. What's going on here?



```
Enter the Celsius temperature: 21
Room temperature? false
```

FIGURE 8-12: A run of the code in Listing 8-5.

REMEMBER

Grouping separators vary from one country to another. The run shown in Figure 8-12 works almost everywhere in the world. But if the Celsius temperature is twenty-one-and-a-half degrees, you type **21.5** (with a dot) in some countries and **21,5** (with a comma) in others. Your computer's hardware doesn't have a built-in "country-ometer," but when you install the computer's operating system, you tell it which country you live in. Java programs access this information and use it to customize the way the `nextDouble` method works.

A little detective work can go a long way. Review the facts:

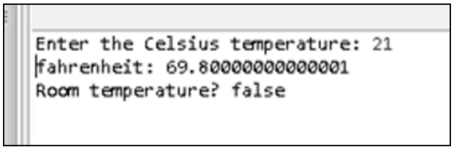
- **Fact:** The value of `fahrenheit` should be exactly `69.8`.
- **Fact:** If `fahrenheit` is 69.8, then `fahrenheit == 69.8` is true.
- **Fact:** In Figure 8-12, the computer displays the word `false`. So the expression `fahrenheit == 69.8` isn't true.

How do you reconcile these facts? There can be little doubt that `fahrenheit == 69.8` is `false`, so what does that say about the value of `fahrenheit`? Nowhere in Listing 8-5 is the value of `fahrenheit` displayed. Could that be the problem?

At this point, I use a popular programmer's trick. I add statements to display the value of fahrenheit:

```
fahrenheit = 9.0 / 5.0 * celsius + 32.0;
System.out.print("fahrenheit: ");        //Added
System.out.println(fahrenheit);          //Added
```

A run of the enhanced code is shown in Figure 8-13. As you can see, the computer misses its mark. Instead of the expected value 69.8, the computer's value for 9.0 / 5.0 * 21 + 32.0 is 69.80000000000001. That's just the way the cookie crumbles. The computer does all its arithmetic with zeros and ones, so the computer's arithmetic doesn't look like the base-10 arithmetic in Figure 8-11. The computer's answer isn't wrong. The answer is just slightly inaccurate.



```
Enter the Celsius temperature: 21
fahrenheit: 69.80000000000001
Room temperature? false
```

FIGURE 8-13: The fahrenheit variable's full value.

In an example in Chapter 7, Java's remainder operator (%) gives you the answer 0.1299999999999999 instead of the 0.13 that you expect. The same strange kind of thing happens in this section's example. But this section's code doesn't use an exotic remainder operator. This section's code uses your old friends division, multiplication, and addition.

Be careful when you compare two numbers for equality (with ==) or for inequality (with !=). Little inaccuracies can creep in almost anywhere when you work with Java's double type or with Java's float type. And several little inaccuracies can build on one another to become very large inaccuracies. When you compare two double values or two float values, the values are almost never dead-on equal to one another.

REMEMBER

If your program isn't doing what you think it should do, check your suspicions about the values of variables. Add print and println statements to your code.

TECHNICAL STUFF

When you compare double values, give yourself some leeway. Instead of comparing for exact equality, ask whether a particular value is reasonably close to the expected value. For example, use a condition like fahrenheit >= 69.8 - 0.01 && fahrenheit <= 69.8 + 0.01 to find out whether fahrenheit is within 0.01 of the value 69.8. To read more about conditions containing Java's && operator, see Chapter 10.

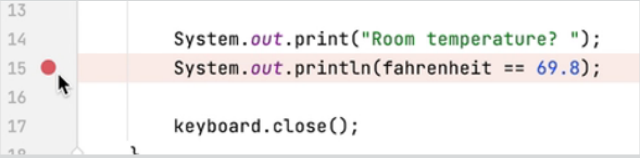
AUTOMATED DEBUGGING

If your program isn't working correctly, you can try something called a debugger. An *automated debugger* can pause your program's run and accept special commands to display variables' values. With some debuggers, you can pause a run and change a variable's value (just to see whether things go better when you do).

In this book, I don't promote the use of an automated debugger. But for any large programming project, automated debugging is an essential tool. If you plan to write bigger and better programs, please give IntelliJ's debugging capabilities a try. For a peek at the things IntelliJ's debugger can do, follow these steps:

1. **Create an IntelliJ project containing Listing 8-5.**
2. **In IntelliJ's editor, click in the margin to the left of a line of code.**

   In the first sidebar figure, I click the `System.out.println` line from Listing 8-5. A little red dot appears in the editor's margin. This dot indicates a *breakpoint* in the code. In the steps that follow, you'll make the run of the program pause immediately before the line with the breakpoint.



```
        System.out.print("Room temperature? ");
        System.out.println(fahrenheit == 69.8);

        keyboard.close();
    }
```

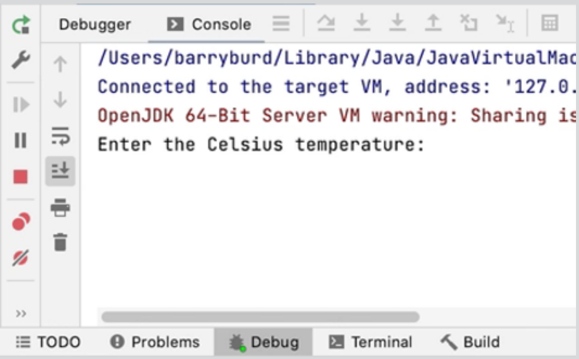
3. **Right-click either the `CelsiusToFahrenheit` branch in the Project tool window or the `CelsiusToFahrenheit` tab at the top of the editor.**
4. **In the resulting context menu, select `Debug 'CelsiusToFahrenheit.main()'`.**

   Remember to select the Debug menu item, not the Run item.

   When you click `Debug 'CelsiusToFahrenheit.main()'`, your code begins running. IntelliJ replaces the Run tool window with its Debug tool window. The Debug tool window has two tabs — a Debugger tab and a Console tab. (See this sidebar's second figure.) On the Console tab, IntelliJ prompts you to enter the Celsius temperature.

(continued)

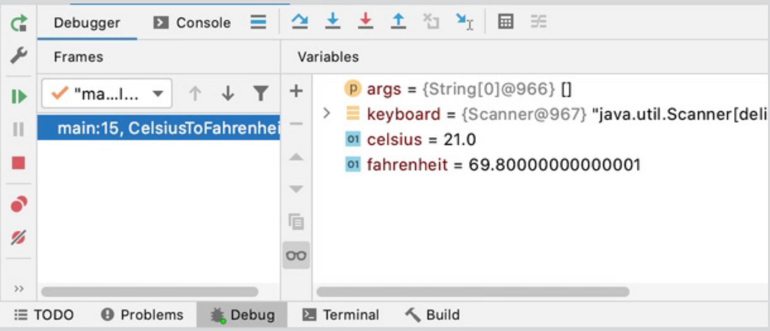
(continued)



```
/Users/barryburd/Library/Java/JavaVirtualMac
Connected to the target VM, address: '127.0.
OpenJDK 64-Bit Server VM warning: Sharing is
Enter the Celsius temperature:
```

5. **On the Console tab, type the number** 21 **and then press Enter.**

   Your code continues running until execution reaches the breakpoint. At the breakpoint, the execution pauses and IntelliJ switches to the Debugger tab. (See this sidebar's third figure.)



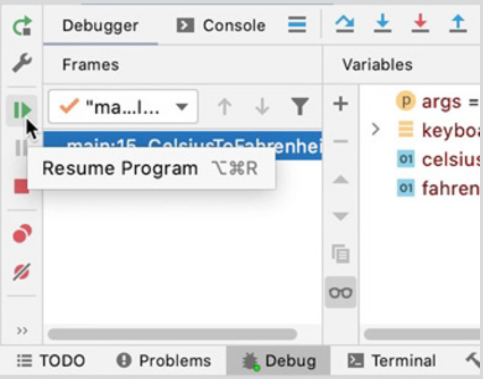
main:15, CelsiusToFahrenhei

```
args = {String[0]@966} []
keyboard = {Scanner@967} "java.util.Scanner[deli
celsius = 21.0
fahrenheit = 69.80000000000001
```

The Debugger tab has two panels — a Frames panel and a Variables panel. The Variables panel displays the values of the program's variables. (That's not surprising.) In this sidebar's third figure, the `fahrenheit` variable's value is `69.80000000000001`. How nice! Using the debugging tools, you can examine variables' values in the middle of a run!

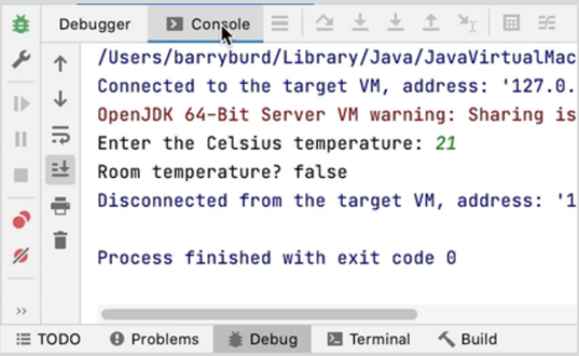
If you compare the `fahrenheit` variable's actual value, `69.80000000000001`, with the number `69.8` the code, you understand why the program is about to display the word `false`.

6. **To finish running your program, click the Resume Program icon along the left edge of the Debug tool window. (See this sidebar's fourth figure.)**



   When the program's run ends, IntelliJ continues to display the Debugger tab.

7. **To see the program's output, select the Debug tool window's Console tab. (See this sidebar's final figure.)**



```
/Users/barryburd/Library/Java/JavaVirtualMac
Connected to the target VM, address: '127.0.
OpenJDK 64-Bit Server VM warning: Sharing is
Enter the Celsius temperature: 21
Room temperature? false
Disconnected from the target VM, address: '1

Process finished with exit code 0
```

Comparing characters

The comparison operators in Table 8-1 work overtime for characters. Roughly speaking, the operator `<` means "comes earlier in the alphabet." But you have to be careful of the following:

- Because `B` comes alphabetically before `H`, the condition `'B' < 'H'` is `true`. That's not surprising.

- Because b comes alphabetically before h, the condition `'b' < 'h'` is `true`. That's no surprise, either.
- Every uppercase letter comes before any of the lowercase letters, so the condition `'b' < 'H'` is false. Now, that's a surprise. (See Figure 8-14.)

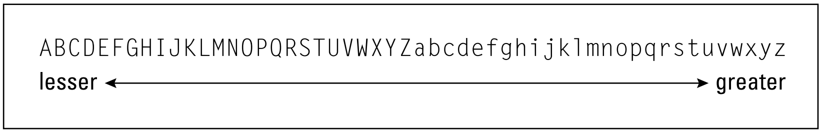


FIGURE 8-14: The ordering of the letters.

In practice, you seldom have reason to compare one letter with another. But in Chapter 14, you can read about Java's `String` type. With the `String` type, you can compare words, names, and other good stuff. At that point, you have to think carefully about alphabetical ordering, and the ideas in Figure 8-14 come in handy.

TECHNICAL STUFF

Under the hood, the letters `A` through `Z` are stored with numeric codes 65 through 90. The letters `a` through `z` are stored with codes 97 through 122. That's why each uppercase letter is "less than" any of the lowercase letters.

The Remaining Primitive Types

In Chapter 7, I tell you that Java has eight primitive types, but Table 7-1 lists only six of eight types. Table 8-2 describes the remaining two types — the types `char` and `boolean`. Table 8-2 isn't too exciting, but I can't just leave you with the incomplete story in Table 7-1.

TABLE 8-2: Java's Primitive Non-Numeric Types

Type Name	Range of Values
Character Type	
`char`	Thousands of characters, glyphs, and symbols
Logical Type	
`boolean`	Only `true` or `false`

TECHNICAL STUFF

If you dissect parts of the Java Virtual Machine, you find that Java considers `char` to be a numeric type. That's because Java represents characters with something called *Unicode* — an international standard for representing alphabets of the world's many languages. For example, the Unicode representation of an uppercase letter C is 67. The representation of a Hebrew letter א is 1488. And (to take a more obscure example) the representation for the voiced retroflex approximant in phonetics is 635. But don't worry about all of this. The only reason I'm writing about the `char` type's being numeric is to save face among my techie friends.

TECHNICAL STUFF

After looking at Table 8-2, you may be wondering what a glyph is. (In fact, I'm proud to be writing about this esoteric concept, whether you have any use for the information or not.) A *glyph* is a particular representation of a character. For example, a and *a* are two different glyphs, but both of these glyphs represent the same lowercase letter of the Roman alphabet. (Because these two glyphs have the same meaning, the glyphs are called *allographs*. If you want to sound smart, find a way to inject the words *glyph* and *allograph* into a casual conversation!)

TRY IT OUT

MORE CHARACTER METHODS

Type the following code into JShell to see how JShell responds:

```
Character.isDigit('a')

Character.isDigit('2')

Character.isLetter('a')

Character.isLetter('2')

Character.isLetterOrDigit('4')

Character.isLetterOrDigit('@')

Character.isLowerCase('b')

Character.isLowerCase('B')

Character.isLowerCase('7')

Character.isJavaIdentifierPart('x')

Character.isJavaIdentifierPart('7')

Character.isJavaIdentifierPart('-')

Character.isJavaIdentifierPart(' ')
```

3 Controlling the Flow

IN THIS PART . . .

Making big decisions (or, more accurately, making not-so-big decisions)

Repeating yourself

Repeating yourself

Repeating yourself again

IN THIS CHAPTER

- **Writing statements that choose between alternatives**
- **Putting statements inside one another**
- **Writing several kinds of decision-making statements**

Chapter 9

Forks in the Road

Here's an excerpt from *Beginning Programming with Java For Dummies,* 6th Edition, Chapter 8:

> If you're trying to store words or sentences (not just single letters), then you need to use something called a *String.*[*]

This excerpt illustrates two important points: First, you may have to use something called a *String*. Second, your choice of action can depend on something being true or false:

> If it's true that you're trying to store words or sentences,
>
> you need to use something called a *String*.

This chapter deals with decision-making, which plays a fundamental role in the creation of instructions. With the material in this chapter, you expand your programming power by leaps and bounds.

[*] This excerpt is reprinted with permission from John Wiley & Sons, Inc. If you can't find a copy of *Beginning Programming with Java For Dummies,* 6th Edition, in your local bookstore, visit `www.dummies.com`.

Decisions, Decisions!

Picture yourself walking along a quiet country road. You're enjoying a pleasant summer day. It's not too hot, and a gentle breeze from the north makes you feel fresh and alert. You're holding a copy of this book, opened to Chapter 9. You read the paragraph about storing words or sentences, and then you look up.

You see a fork in the road. You see two signs — one pointing to the right and the other pointing to the left. One sign reads, "Storing words or sentences? True." The other sign reads, "Storing words or sentences? False." You evaluate the words-or-sentences situation and march on, veering right or left depending on your software situation. A diagram of this story is shown in Figure 9-1.

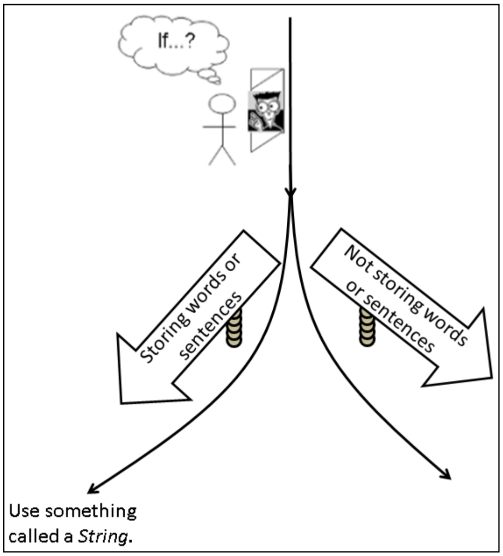

If...?
Storing words or sentences
Not storing words or sentences
Use something called a *String*.

FIGURE 9-1: Which way to go?

Life is filled with forks in the road. Take an ordinary set of directions for heating a frozen snack:

- **Microwave cooking directions:**
  - Place on microwave-safe plate.
  - Microwave on high for 2 minutes.
  - Turn product.
  - Microwave on high for 2 more minutes.

- **Conventional oven directions:**
  - Preheat oven to 350 degrees.
  - Place product on baking sheet.
  - Bake for 25 minutes.

Again, you choose between alternatives. If you use a microwave oven, do this. Otherwise, do that.

In fact, it's hard to imagine useful instructions that don't involve choices. If you're a homeowner with two dependents and earning more than $30,000 per year, check here. If you don't remember how to use curly braces in Java programs, see Chapter 4. Did the user correctly type the password? If yes, then let the user log in; if no, then kick the bum out. If you think the market will go up, then buy stocks; otherwise, buy bonds. And if you buy stocks, which should you buy? And when should you sell?

Making Decisions (Java if Statements)

When you work with computer programs, you make one decision after another. Almost every programming language has a way of branching in one of two directions. In Java (and in many other languages), the branching feature is called an *if statement*. Check out Listing 9-1 to see an `if` statement.

LISTING 9-1: An if Statement

```
if (randomNumber > 5) {
    System.out.println("Yes. Isn't it obvious?");
} else {
    System.out.println("No, and don't ask again.");
}
```

To see a complete program containing the code from Listing 9-1, skip to Listing 9-2 (or, if you prefer, walk, jump, or run to Listing 9-2).

The `if` statement in Listing 9-1 represents a branch, a decision, two alternative courses of action. In plain English, this statement has the following meaning:

```
If the randomNumber variable's value is greater than 5,
    display "Yes. Isn't it obvious?" on the screen.
Otherwise,
    display "No, and don't ask again." on the screen.
```

Pictorially, you get the fork shown in Figure 9-2.

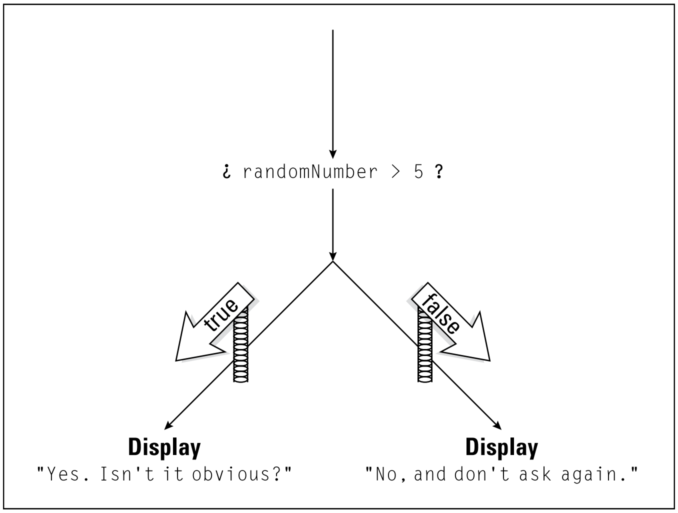


FIGURE 9-2: A random number decides your fate.

A careful look at if statements

An `if` statement can take the following form:

```
if (Condition) {
    SomeStatements
} else {
    OtherStatements
}
```

To get a real-life `if` statement, substitute meaningful text for the three placeholders *Condition*, *SomeStatements*, and *OtherStatements.* Here's how I make the substitutions in Listing 9-1:

- I substitute `randomNumber > 5` for *Condition*.
- I substitute `System.out.println("Yes. Isn't it obvious?");` for *SomeStatements*.
- I substitute `System.out.println("No, and don't ask again.");` for *OtherStatements*.

The substitutions are illustrated in Figure 9-3.

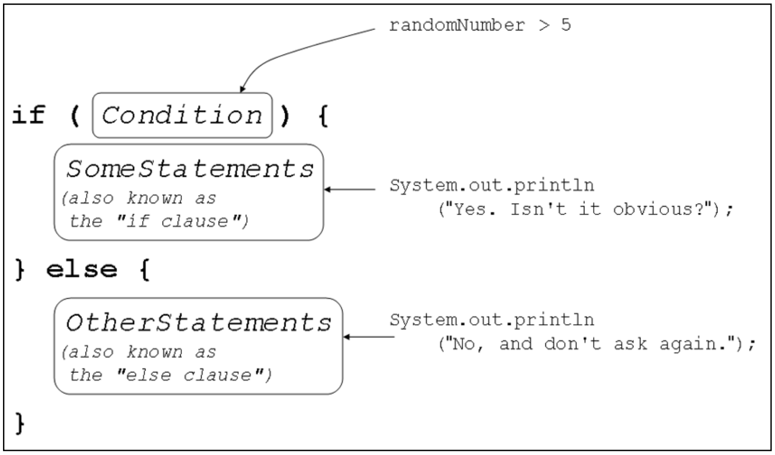

randomNumber > 5
if (Condition) {
SomeStatements
(also known as the "if clause")
System.out.println ("Yes. Isn't it obvious?");
} else {
OtherStatements
(also known as the "else clause")
System.out.println ("No, and don't ask again.");
}

FIGURE 9-3: An `if` statement and its format.

Sometimes, I need alternative names for parts of an `if` statement. I call them the *if clause* and the *else clause:*

```
if (Condition) {
    if clause
} else {
    else clause
}
```

An `if` statement is an example of a *compound statement* — a statement that includes other statements within it. The `if` statement in Listing 9-1 includes two `println` calls, and these calls to `println` are statements.

Notice how I use parentheses and semicolons in the `if` statement of Listing 9-1. In particular, notice the following:

- The condition must be in parentheses.
- Statements inside the `if` clause end with semicolons. So do statements inside the `else` clause.
- There's no semicolon immediately after the condition.
- There's no semicolon immediately after the word `else`.

As a beginning programmer, you may think these rules are arbitrary. But they're not. These rules belong to a carefully crafted grammar. They're like the grammar rules for English sentences, but they're even more logical! (Sorry, Becky.)

Table 9-1 shows you the kinds of things that can go wrong when you break the `if` statement's punctuation rules. The table's last two items are the most notorious. In these two situations, the compiler doesn't catch the error. This lulls you into a false sense of security. The trouble is, when you run the program, the code's behavior isn't what you expect it to be.

TABLE 9-1: Common if Statement Error Messages

Error	Example	Most Likely Messages or Results

Missing parentheses surrounding the condition	`if randomNumber > 5 {`	`'(' expected`
		`';' expected`
		`'else' without 'if'`

Missing semicolon after a statement that's inside the `if` clause or the `else` clause	if (randomNumber > 5) {
	System.out.println("Y")
	}
		`';' expected`

Semicolon immediately after the condition	if (randomNumber > 5); {
	System.out.println("Y");
	} else {
		`'if' statement has empty body`
		`'else' without 'if'`

Semicolon immediately after the word `else`	} else; {	The program compiles without errors, but the statement after the word `else` is always executed, whether the condition is true or false.

Missing curly braces	`if (randomNumber > 5)`
	`otherValue = 7;`
	`System.out.println("Y");`
	`else`
	`otherValue = 9;`
	`System.out.println("N");`
		`'else' without 'if'`
		The program sometimes compiles without errors, but the program's run may not do what you expect it to do. (The bottom line: Don't omit the curly braces.)

As you compose code, it helps to think of an `if` statement as one indivisible unit. Rather than type the whole first line (condition and all), try typing the `if` statement's skeletal outline:

```
if () {          //To do: Fill in the condition.
                 //To do: Fill in the if clause.
} else {
                 //To do: Fill in the else clause.
}
```

With the entire outline in place, you can start working on the items on your to-do list. When you apply this kind of thinking to a compound statement, it's harder to make a mistake.

A complete program

Listing 9-2 contains a complete program with a simple `if` statement. The listing's code behaves like an electronic oracle. Ask the program a yes-or-no question and the program answers you back. Of course, the answer to your question is randomly generated. Who cares? It's fun to ask anyway.

LISTING 9-2: I Know Everything

```
import java.util.Scanner;
import java.util.Random;

public class AnswerYesOrNo {

    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        Random myRandom = new Random();
        int randomNumber;

        System.out.print("Type your question, my child: ");
        keyboard.nextLine();

        randomNumber = myRandom.nextInt(10) + 1;

        if (randomNumber > 5) {
            System.out.println("Yes. Isn't it obvious?");
        } else {
            System.out.println("No, and don't ask again.");
        }

        keyboard.close();
    }
}
```

Figure 9-4 shows several runs of the program in Listing 9-2. The program's action has four parts:

1. **Prompt the user.**

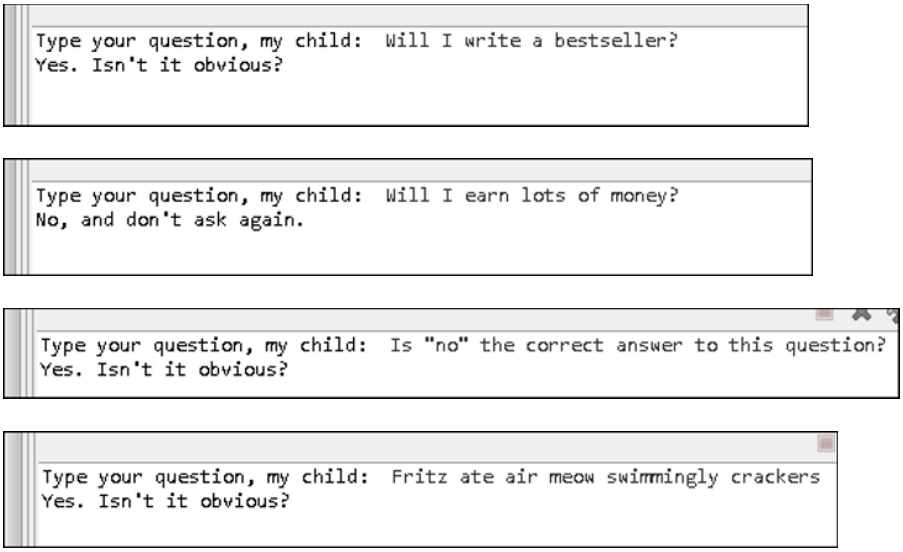
   Call `System.out.print`, telling the user to type a question.

2. **Get the user's question from the keyboard.**

   In Figure 9-4, I run the `AnswerYesOrNo` program four times, and I type a different question each time. Meanwhile, back in Listing 9-2, the statement

   ```
   keyboard.nextLine();
   ```

   swallows up my question and does absolutely nothing with it. This is an anomaly, but you're smart, so you can handle it.



```
Type your question, my child:  Will I write a bestseller?
Yes. Isn't it obvious?
```

```
Type your question, my child:  Will I earn lots of money?
No, and don't ask again.
```

```
Type your question, my child:  Is "no" the correct answer to this question?
Yes. Isn't it obvious?
```

```
Type your question, my child:  Fritz ate air meow swimmingly crackers
Yes. Isn't it obvious?
```

FIGURE 9-4: The all-knowing Java program in action.

   Normally, when a program gets input from the keyboard, the program does something with the input. For example, the program can assign the input to a variable:

   ```
   amount = keyboard.nextDouble();
   ```

   Alternatively, the program can display the input on the screen:

   ```
   System.out.println(keyboard.nextLine());
   ```

But the code in Listing 9-2 is different. When this `AnswerYesOrNo` program runs, the user has to type something. (The call to `nextLine` waits for the user to type some stuff and then press Enter.) But the `AnswerYesOrNo` program has no need to store the input for further analysis. (The computer does what I do when my wife asks me whether I plan to clean up after myself — I ignore the question and make up an arbitrary answer.) So the program doesn't do anything with the user's input. The call to `keyboard.nextLine` just sits there in a statement of its own, doing nothing, behaving like a big, black hole. It's unusual for a program to do this, but an electronic oracle is an unusual thing. It calls for some slightly unusual code.

3. **Get a random number — any `int` value from 1 to 10.**

   Okay, wise guys. You've just trashed the user's input. How will you answer yes or no to the user's question?

   No problem! None at all! You'll display an answer randomly. The user won't know the difference. (Ha-ha!) You can do this as long as you can generate random numbers. The numbers from 1 to 10 will do just fine.

   In Listing 9-2, the stuff about `Random` and `myRandom` looks much like the familiar `Scanner` code. From a beginning programmer's point of view, `Random` and `Scanner` work almost the same way. Of course, there's an important difference: A call to the `Random` class's `nextInt(10)` method doesn't fetch anything from the keyboard. Instead, this `nextInt(10)` method gets a number out of the blue.

   The name `Random` is defined in the Java API. The call to `myRandom.nextInt(10)` in Listing 9-2 gets a number from 0 to 9. Then my code adds 1 (making a number from 1 to 10) and assigns that number to the variable `randomNumber.` When that's done, you're ready to answer the user's question.

   TECHNICAL STUFF

   In Java's API, the word `Random` is the name of a Java class, and `nextInt` is the name of a Java method. For more information on the relationship between classes and methods, see Part 4.

4. **Answer yes or no.**

   Calling `myRandom.nextInt(10)` is like spinning a wheel on a TV game show. The wheel has slots numbered from 1 to 10. The `if` statement in Listing 9-2 turns your number into a yes-or-no alternative. If you roll a number that's greater than 5, the program answers `yes`. Otherwise (if you roll a number that's less than or equal to 5), the program answers `no`.

   You can trust me on this one. I've made lots of important decisions based on my `AnswerYesOrNo` program.

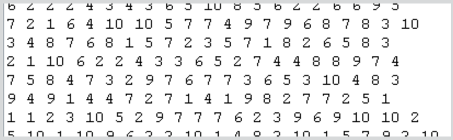
RANDOMNESS MAKES ME DIZZY

When you call `myRandom.nextInt(10) + 1`, you get a number from 1 to 10. As a test, I wrote a program that calls `myRandom.nextInt(10) + 1` 20 times:

```
Random myRandom=new Random();
System.out.print(myRandom.nextInt(10) + 1);
System.out.print(" ");
System.out.print(myRandom.nextInt(10) + 1);
System.out.print(" ");
System.out.print(myRandom.nextInt(10) + 1);
// ... And so on.
```

I ran the program several times and got the results shown in the following figure. (Actually, I copied the results from IntelliJ's Run tool window to Windows Notepad.) Stare briefly at the figure and notice two trends:

- There's no obvious way to predict what number comes next.
- No number occurs much more often than any of the others.



```
6 2 2 2 4 3 4 3 6 5 10 8 5 6 2 2 6 6 9 5
7 2 1 6 4 10 10 5 7 7 4 9 7 9 6 8 7 8 3 10
3 4 8 7 6 8 1 5 7 2 3 5 7 1 8 2 6 5 8 3
2 1 10 6 2 2 4 3 3 6 5 2 7 4 4 8 8 9 7 4
7 5 8 4 7 3 2 9 7 6 7 7 3 6 5 3 10 4 8 3
9 4 9 1 4 4 7 2 7 1 4 1 9 8 2 7 7 2 5 1
1 1 2 3 10 5 2 9 7 7 7 6 2 3 9 6 9 10 10 2
5 10 1 10 9 6 3 3 10 1 4 8 3 10 1 5 7 9 3 10
```

The Java Virtual Machine jumps through hoops to maintain these trends. That's because cranking out numbers in a random fashion is a tricky business. Here are some interesting facts about the process:

- **Scientists and nonscientists use the term *random number,* but in reality, there's no such thing as a single random number.** After all, how random is a number like 9?

  A number is *random* only when it's one in a disorderly collection of numbers. More precisely, a number is *random* if the process used to generate the number follows the two preceding trends. When they're being careful, scientists avoid the term *random number* and use the term *randomly generated number* instead.

- **It's hard to generate numbers randomly.** Computer programs do the best they can, but ultimately, today's computer programs follow a pattern, and that pattern isn't truly random.

To generate numbers in a truly random fashion, you need a big tub of ping-pong balls, like the kind they use in state lottery drawings. The problem is, most computers don't come with big tubs of ping-pong balls among their peripherals. So, strictly speaking, the numbers generated by Java's `Random` class aren't random. Instead, scientists call these numbers *pseudorandom*.

- **If you don't have a big tub of ping-pong balls, you can find other ways to generate numbers randomly.** For example, a quantum computer can create something called a *qubit* and put the qubit in a state that's halfway between 0 and 1. When you measure the qubit's state, you don't see the halfway business. You always observe either regular old 0 or regular old 1. Quantum theory dictates that the outcome of the measurement (0 or 1) is truly random.

  In 1935, a fellow named Erwin Schrödinger considered making a qubit out of a real, live cat. Schrödinger's cat could be alive (1) or dead (0), and only the random whim of nature would determine the cat's state of being. Things went well until Schrödinger tried to generate 100 values in a row. His landlady heard the meowing, smelled the cat boxes, and immediately called the local health department. Schrödinger's experiment came to an abrupt end.

  Rumors about an experiment named "Schrödinger's Landlady" persist to this day.

- **It surprises us all, but knowing one randomly generated value is of no help in predicting the next randomly generated value.**

  For example, if you toss a coin twice and it lands heads-up both times, is it more likely to land tails-up on the third flip? No. It's still 50-50.

  If you have three sons and you're expecting a fourth child, is the fourth child more likely to be a girl? No. A child's gender has nothing to do with the genders of the older children. (I'm ignoring any biological effects, which I know absolutely nothing about. Wait! I do know some biological trivia: A newborn child is more likely to be a boy than a girl. In the United States, for every 21 newborn boys, there are only 20 newborn girls. Boys are weaker, so guys like me die off faster. That's why nature makes more of us at birth.)

A treatise on the importance of helpful indentation

Notice how, in Listing 9-2, the `println` calls inside the `if` statement are indented. Strictly speaking, you don't have to indent the statements that are inside an `if` statement. For all the compiler cares, you can write your whole program on a single line or place all your statements in an artful, misshapen zigzag. The problem is, if you don't indent your statements in some logical fashion, neither you nor anyone else can make sense of your code. In Listing 9-2, the indenting of the `println` calls helps your eyes (and brain) see quickly that these statements are subordinate to the overall `if`/`else` flow.

In a small program, unindented or poorly indented code is barely tolerable. But in a complicated program, indentation that doesn't follow a neat, logical pattern is a big, ugly nightmare.

REMEMBER

Always indent your code to make the program's flow apparent at a glance.

TIP

You don't have to think about indenting your code, because IntelliJ can indent your code automatically. For details, see Chapter 4.

Variations on the Theme

I don't like to skin cats. But I've heard that, if I ever need to skin one, I have a choice of several techniques. I'll keep that in mind the next time my cat Histamine mistakes the carpet for a litter box.[*]

Anyway, whether you're skinning catfish, skinning kitties, or writing computer programs, the same principle holds true: You always have alternatives. Listing 9-2 shows you one way to write an `if` statement. The rest of this chapter (and all of Chapter 10) shows you some other ways to create `if` statements.

[*] Rick Ross, who read about skinning cats in one of my other books, sent me this information via email: " . . . you refer to 'skinning the cat' and go on to discuss litter boxes and whatnot. Please note that the phrase "more than one way to skin a cat" refers to the difficulty in removing the inedible skin from catfish, and that there is more than one way to do same. These range from nailing the critter's tail to a board and taking a pair of pliers to peel it down to letting the furry kind of cat have the darn thing and just not worrying about it. I grew up on The River (the big one running north/south down the US that begins with *M* and has so many repeated letters), so it's integral to our experience there." Another reader, Alan Wilson, added his two cents to this discussion: ". . . the phrase *skinning a cat* . . . actually has an older but equally interesting British naval origin — it refers to the activity of attaching the nine ropes to the whip used to punish recalcitrant sailors up to a couple of hundred years ago. The cat-o'-nine-tails was the name of the whip, and there was more than one way to attach the ropes or 'skin' the whip." One way or another, it's time for me to apologize to my little house pet.

. . . or else what?

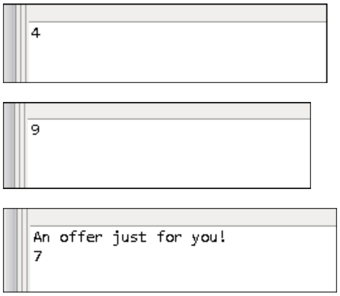
You can create an if statement without an else clause. For example, imagine a web page on which one in ten randomly chosen visitors receives a special offer. To keep visitors guessing, I call the Random class's nextInt method and make the offer to anyone whose number is lucky 7:

- If myRandom.nextInt(10) + 1 generates the number 7, display a special offer message.
- If myRandom.nextInt(10) + 1 generates any number other than 7, do nothing. Don't display a special offer message, and don't display the discouraging message "Sorry, no offer for you."

The code to implement such a strategy is shown in Listing 9-3. A few runs of the code are shown in Figure 9-5.

LISTING 9-3: Aren't You Lucky?

```
import java.util.Random;

public class SpecialOffer {

    public static void main(String[] args) {
        var myRandom = new Random();
        int randomNumber = myRandom.nextInt(10) + 1;

        if (randomNumber == 7) {
            System.out.println("An offer just for you!");
        }
        System.out.println(randomNumber);
    }
}
```



```
4
```

```
9
```

```
An offer just for you!
7
```

FIGURE 9-5: Three runs of the code in Listing 9-3.

The if statement in Listing 9-3 has no else clause. This if statement has the following form:

```
if (Condition) {
    SomeStatements
}
```

When randomNumber is 7, the computer displays An offer just for you! When randomNumber isn't 7, the computer doesn't display An offer just for you! The action is illustrated in Figure 9-6.

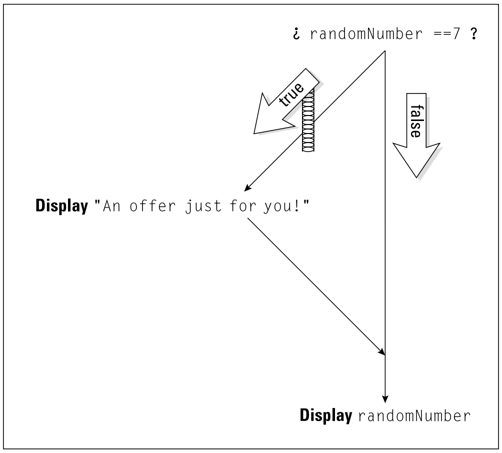


FIGURE 9-6: If you have nothing good to say, don't say anything.

REMEMBER

Always (I mean *always*) use a double equal sign when you compare two numbers or characters in an if statement's condition. Never (that's *never, ever, ever*) use a single equal sign to compare two values. A single equal sign does assignment, not comparison.

In Listing 9-3, I took the liberty of adding an extra println. This println (at the end of the main method) displays the random number generated by my call to nextInt. On a web page with special offers, you probably wouldn't see the randomly generated number, but I can't test my SpecialOffer code without knowing which numbers the code generates.

Anyway, notice that the value of randomNumber is displayed in every run. The println for randomNumber isn't inside the if statement. (This println comes after the if statement.) So the computer always executes this println. Whether randomNumber == 7 is true or false, the computer takes the appropriate if action and then marches on to execute System.out.println(randomNumber).

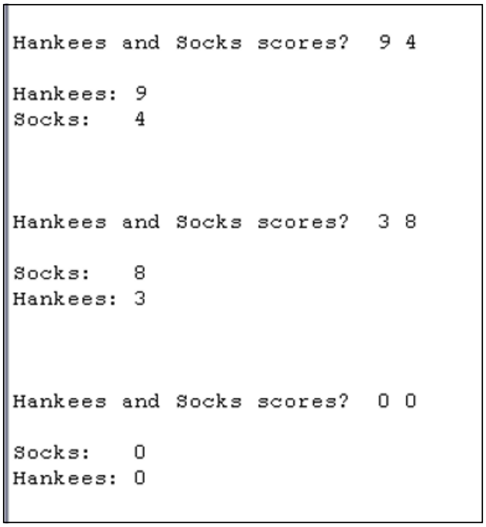
Packing more stuff into an if statement

Here's an interesting situation: You have two baseball teams — the Hankees and the Socks. You want to display the teams' scores on two separate lines, with the winner's score coming first. (On the computer screen, the winner's score is displayed above the loser's score. In case of a tie, you display the two identical scores, one above the other.) Listing 9-4 has the code.

LISTING 9-4: May the Best Team Be Displayed First

```
import java.util.Scanner;
import static java.lang.System.in;
import static java.lang.System.out;

public class TwoTeams {

    public static void main(String[] args) {
        var keyboard = new Scanner(in);
        int hankees, socks;

        out.print("Hankees and Socks scores? ");
        hankees = keyboard.nextInt();
        socks = keyboard.nextInt();
        out.println();

        if (hankees > socks) {
            out.print("Hankees: ");
            out.println(hankees);
            out.print("Socks: ");
            out.println(socks);
        } else {
            out.print("Socks: ");
            out.println(socks);
            out.print("Hankees: ");
            out.println(hankees);
        }

        keyboard.close();
    }
}
```

Figure 9-7 has a few runs of the code. (To show a few runs in one figure, I copied the results from IntelliJ's Run tool window to Windows Notepad.)



```
Hankees and Socks scores?  9 4

Hankees: 9
Socks:   4



Hankees and Socks scores?  3 8

Socks:   8
Hankees: 3



Hankees and Socks scores?  0 0

Socks:   0
Hankees: 0
```

FIGURE 9-7: See? The code in Listing 9-4 really works!

With curly braces, a bunch of `print` and `println` calls are tucked away safely inside the `if` clause. Another group of `print` and `println` calls is squished inside the `else` clause. This creates the forking situation shown in Figure 9-8.

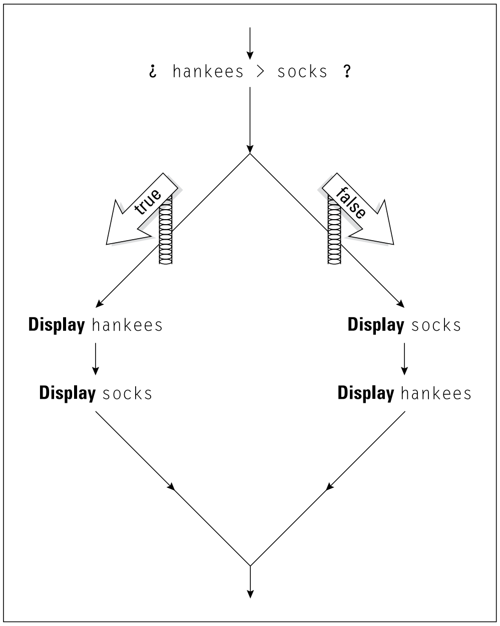


FIGURE 9-8: Cheer for your favorite team.

STATEMENTS AND BLOCKS

An elegant way to think about `if` statements is to realize that you can put only one statement inside each clause of an `if` statement:

```
if (Condition)
    aStatement
else
    anotherStatement
```

On your first reading of this 1-statement rule, you're probably thinking that there's a misprint. After all, in Listing 9-4, each clause (the `if` clause and the `else` clause) seems to contain four statements, not just one.

But technically, the `if` clause in Listing 9-4 has only one statement, and the `else` clause in Listing 9-4 has only one statement. The trick is, when you surround a bunch of statements with curly braces, you get what's called a *block,* and a block behaves, in all respects, like a single statement. In fact, the official Java documentation lists a block as a kind of statement (one of many different kinds of statements). So, in Listing 9-4, the block

```
{
    out.print("Hankees: ");
    out.println(hankees);
    out.print("Socks: ");
    out.println(socks);
}
```

is a single statement. It's a statement that has four smaller statements within it. So this big block, this single statement, serves as the one-and-only statement inside the `if` clause in Listing 9-4.

That's how the 1-statement rule works. In an `if` statement, when you want the computer to execute several statements, you combine those statements into one big statement. To do this, you make a block using curly braces.

Some handy import declarations

When I wrote this section's example, I was tired of writing the word `System`. After all, Listing 9-4 has ten `System.out.print` lines. By this point in the book, shouldn't my computer remember what `out.print` means?

Of course, computers don't work that way. If you want a computer to "know" what `out.print` means, you have to code that knowledge somewhere inside the Java compiler.

Fortunately for me, the ability to abbreviate things like `System.out.print` is available from Java 5.0 onward. (An older Java compiler simply chokes on the code in Listing 9-4.) This ability to abbreviate things is called *static import*. It's illustrated in the second and third lines of Listing 9-4.

Whenever I start a program with the line

```
import static java.lang.System.out;
```

I can replace `System.out` with plain `out` in the remainder of the program. The same holds true of `System.in`. With an import declaration near the top of Listing 9-4, I can replace `new Scanner(System.in)` with the simpler `new Scanner(in)`.

You may be wondering what all the fuss is about. If I can abbreviate `java.util.Scanner` by writing `Scanner`, what's so special about abbreviating `System.out`? And why do I have to write `out.print`? Can I trim `System.out.print` to the single word `print`? Look again at the first few lines of Listing 9-4. When do you need the word `static`? And what's the difference between `java.util` and `java.lang`?

I'm sorry. My response to these questions won't thrill you. The fact is, I can't explain away any of these issues until Chapter 14. Before I can explain static import declarations, I need to introduce some ideas. I need to describe classes, packages, and static members.

Until you reach Chapter 14, please bear with me. Just paste three import declarations to the top of your Java programs and trust that everything will work well.

REMEMBER

You can abbreviate `System.out` with the single word `out`. And you can abbreviate `System.in` with the single word `in`. Just be sure to copy the `import` declarations *exactly* as you see them in Listing 9-4. With any deviation from the lines in Listing 9-4, you may get a compiler error.

TRY IT OUT

Get some practice writing `if` statements!

OOPS!

What's wrong with the following code? How can the code be fixed?

```
System.out.println("How many donuts are in a dozen?");
int number = keyboard.nextInt();

if (number = 12) {
    System.out.println("That's correct.");
} else {
    System.out.println("Sorry. That's incorrect");
}
```

DON’T WRITE CODE THIS WAY

When I wrote the following code, I didn’t indent the code properly. What’s the output of this bad code? Why?

```
int n = 100;

if (n > 100)
System.out.println("n is big");
System.out.println("Will Java display this line of text?");
if (n <= 100)
System.out.println("n is small");
System.out.println("How about this line of text?");
```

THE WORLD SMILES WITH YOU

Write a program that asks the users whether they want to see a smiley face. If the user replies `Y` (meaning “yes”), the code displays this:

```
:-)
```

Otherwise, the code displays this:

```
:-(
```

SUCCESSIVE IF STATEMENTS

Modify the previous program (the smiley face program) to take three possibilities into account:

- If the user replies `Y` (meaning “yes”), the code displays `:-)`.
- If the user replies `N` (meaning “no”), the code displays `:-(`.
- If the user replies `?` (meaning “I don’t know”), the code displays `:-|`.

Use three separate `if` statements, one after another.

GUESSING GAME

Write a program that randomly generates a number from 1 to 10. The program then reads a number that the user enters on the keyboard. If the user's number is the same as the randomly generated number, the program displays `You win!`. Otherwise, the program displays `You lose`.

CONVERTING LENGTHS

Write a program that reads a number of meters from the keyboard. The program also reads a letter from the keyboard. If the letter is `c`, the program converts the number of meters into centimeters and displays the result. If the letter is `m`, the program converts the number of meters into millimeters and displays the result. For any other number, the program displays no result.

PUTTING SEVERAL STATEMENTS INSIDE AN IF STATEMENT

Find a short poem (maybe four or five lines long). Write a program that asks the users whether they want to read the poem. If the user replies `Y` (meaning "yes"), display the poem in IntelliJ's Run tool window. If the user's reply is anything other than `Y`, display the following:

```
Sorry!
I thought you were a poetry buff.
Maybe you'll want to see the poem the next time you run this program.
```

IN THIS CHAPTER

- » **Untangling complicated conditions**
- » **Writing cool conditional code**
- » **Intertwining your `if` statements**

Chapter 10
Which Way Did He Go?

It's tax time again. At the moment, I'm working on Form 12432-89B. Here's what it says:

> Under this regulation, the undeclared possession of certain goods, items, articles, materials, or fragments, whether living or dead, or held as proxy for other individuals or agents, or in any form comparable to the ownership of such goods will be subject to all provisions under Section 6.1.03-89(b) [and consequently 6.1.03-90(ak)] of the aforementioned act or acts, whereupon said parties will be liable for all holds, leans, and obligations in perpetuity or under such limits as provided by law, or in the execution (supplemental or non-supplemental), handling, or consideration thereof.

No wonder I have no time to write! I'm too busy interpreting these tax forms.

Anyway, this chapter deals with the potential complexity of `if` statements. This chapter has nothing as complex as Form 12432-89B, but if you ever encounter something that complicated, you'll be ready for it.

Forming Bigger and Better Conditions

In Listing 9-2 (refer to Chapter 9), the code chooses a course of action based on one call to the `Random` class's `nextInt` method. That's fine for the electronic oracle program described in Chapter 9, but what if you're rolling a pair of dice? In backgammon and other dice games, rolling 3 and 5 isn't the same as rolling 4 and 4, even though the total for both rolls is 8. The next move varies, depending on whether you roll doubles. To get the computer to roll two dice, you execute `myRandom.nextInt(6) + 1` two times. Then you combine the two rolls into a larger, more complicated `if` statement.

To simulate a backgammon game (and many other, more practical situations) you need to combine conditions:

```
If die1 + die2 equals 8 and die1 equals die2, ...
```

You need things like *and* and *or* — things that can wire conditions together. Java has operators to represent these concepts, which are described in Table 10-1 and illustrated in Figure 10-1.

TABLE 10-1: Logical Operators

Operator Symbol	Meaning	Example	Illustration
&&	and	`4 < age && age < 8`	Figure 10-1(a)
||	or	`age < 4 || 8 < age`	Figure 10-1(b)
!	not	`!eachKidGetsTen`	Figure 10-1(c)

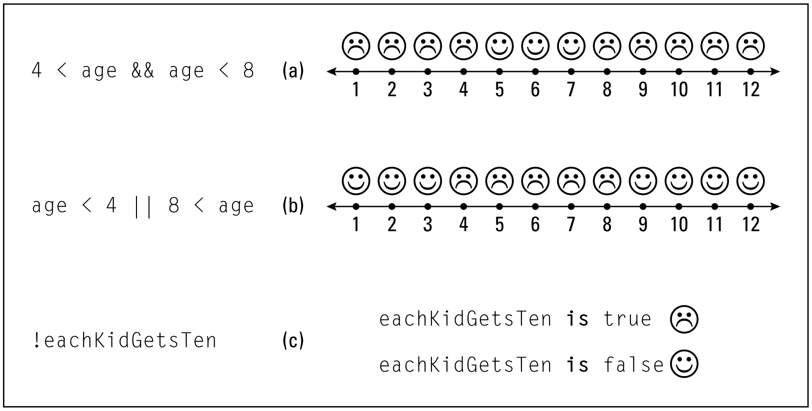


FIGURE 10-1: When you satisfy a condition, you're happy.

Combined conditions, like the ones in Table 10-1, can be mighty confusing. That's why I tread carefully when I use such things. Here's a short explanation of each example in the table:

- `4 < age && age < 8`

  The value of the `age` variable is greater than 4 *and* is less than 8. The numbers 5, 6, 7, 8, 9 . . . are all greater than 4. But among these numbers, only 5, 6, and 7 are less than 8. So only the numbers 5, 6, and 7 satisfy this combined condition.

- `age < 4 || 8 < age`

  The value of the `age` variable is less than 4 *or* is greater than 8. To create the *or* condition, you use two pipe symbols. On many US English keyboards, you can find the pipe symbol immediately above the Enter key (the same key as the backslash, but shifted).

  In this combined condition, the value of the `age` variable is either less than 4 or greater than 8. For example, if a number is less than 4, the number satisfies the condition. Numbers like 1, 2, and 3 are all less than 4, so these numbers satisfy the combined condition.

  Also, if a number is greater than 8, the number satisfies the combined condition. Numbers like 9, 10, and 11 are all greater than 8, so these numbers satisfy the condition.

- `!eachKidGetsTen`

  If I weren't experienced with computer programming languages, I'd be confused by the exclamation point. I'd think that `!eachKidGetsTen` means, "Yes, each kid *does* get ten." But that's not what this expression means. This expression says, "The variable `eachKidGetsTen` does *not* have the value `true`." In Java and other programming languages, an exclamation point stands for *negative,* for *no way,* for *not.*

  Listing 8-4 (refer to Chapter 8) has a `boolean` variable named `eachKidGetsTen`. A `boolean` variable's value is either `true` or `false`. Because `!` means not, the expressions `eachKidGetsTen` and `!eachKidGetsTen` have opposite values. So, when `eachKidGetsTen` is `true`, `!eachKidGetsTen` is `false` (and vice versa).

TIP

Java's `||` operator is *inclusive.* This means that you get `true` whenever the thing on the left side is `true`, the thing on the right side is `true`, or both things are `true`. For example, the condition `2 < 10 || 20 < 30` is true.

WARNING

In Java, you can't combine comparisons the way you do in ordinary English. In English, you may say, "We'll have between three and ten people at the dinner table." But in Java, you get an error message if you write `3 <= people <= 10`. To do this comparison, you need something like `3 <= people && people <= 10`.

Combining conditions: An example

Here's a handy example of the use of logical operators. A movie theater posts its prices for admission:

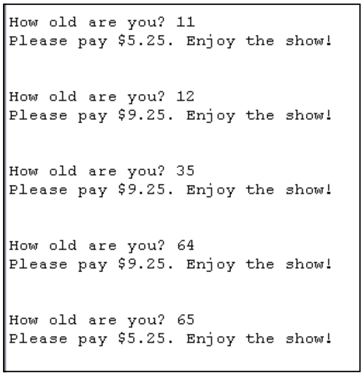
> Regular price: $9.25
>
> Kids under 12: $5.25
>
> Seniors (65 and older): $5.25

Because the kids' and seniors' prices are the same, you can combine these prices into one category. (That's not always the best programming strategy, but do it anyway for this example.) To find a particular moviegoer's ticket price, you need one or more `if` statements. You can structure the conditions in many ways, and I chose one of these ways for the code in Listing 10-1.

LISTING 10-1: Are You Paying Too Much?

```
import java.util.Scanner;

public class TicketPrice {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        int age;
        double price = 0.00;

        System.out.print("How old are you? ");
        age = keyboard.nextInt();

        if (age >= 12 && age < 65) {
            price = 9.25;
        }
        if (age < 12 || age >= 65) {
            price = 5.25;
        }

        System.out.print("Please pay $");
        System.out.print(price);
        System.out.print(". ");
        System.out.println("Enjoy the show!");

        keyboard.close();
    }
}
```

Several runs of the `TicketPrice` program (refer to Listing 10-1) are shown in Figure 10-2. (For your viewing pleasure, I've copied the runs from IntelliJ's Run tool window to Windows Notepad.) When you turn 12, you start paying full price. You keep paying the full price until you turn 65. At that point, you pay the reduced price again.



```
How old are you? 11
Please pay $5.25. Enjoy the show!


How old are you? 12
Please pay $9.25. Enjoy the show!


How old are you? 35
Please pay $9.25. Enjoy the show!


How old are you? 64
Please pay $9.25. Enjoy the show!


How old are you? 65
Please pay $5.25. Enjoy the show!
```

FIGURE 10-2: Admission prices for *Beginning Programming with Java For Dummies: The Movie.*

The pivotal part of Listing 10-1 is the lump of `if` statements in the middle, which are illustrated in Figure 10-3.

- The first `if` statement's condition tests for the regular-price group. Anyone who's at least 12 years of age *and* is under 65 belongs in this group.
- The second `if` statement's condition tests for the fringe ages. A person who's under 12 *or* is 65 or older belongs in this category.

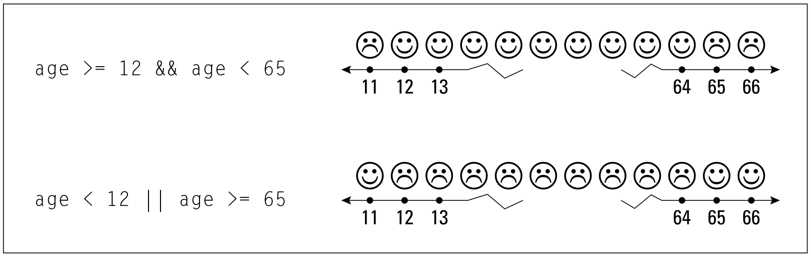


FIGURE 10-3: The meanings of the conditions in Listing 10-1.

TIP

When you form the opposite of an existing condition, you can often follow the pattern in Listing 10-1. Change `>=` to `<`. Change `<` to `>=`. Change `&&` to `||`.

WARNING

If you change the dollar amounts in Listing 10-1, you can get into trouble. For example, with the statement `price = 5.00`, the program displays `Please pay $5.0. Enjoy the show!` This happens because Java doesn't store the two zeros to the right of the decimal point (and Java doesn't know or care that 5.00 is a dollar amount). To fix this kind of thing, see the discussion of `NumberFormat.getCurrencyInstance` in Chapter 14.

When to initialize?

Take a look at Listing 10-1 and notice the `price` variable's initialization:

```
double price = 0.00;
```

This line declares the `price` variable and sets the variable's starting value to `0.00`. When I omit this initialization, I get an error message:

```
The local variable price may not have been initialized
```

What's the deal here? I don't initialize the `age` variable, but the compiler doesn't complain about that. Why is the compiler fussing over the `price` variable?

The answer is in the placement of the code's assignment statements. Consider the following two facts:

- **The statement that assigns a value to the `age` variable (`age = keyboard.nextInt()`) isn't inside an `if` statement.**

  That assignment statement always gets executed, and (as long as nothing extraordinary happens) the variable `age` is sure to be assigned a value.

- **Both statements that assign a value to the `price` variable (`price = 9.25` and `price = 5.25`) are inside `if` statements.**

  If you look at Figure 10-3, you see that every age group is covered. No one shows up at the ticket counter with an age that forces both `if` conditions to be `false`. So, whenever you run the `TicketPrice` program, either the first or the second `price` assignment is executed.

  The problem is that the compiler isn't smart enough to check all of this. The compiler just sees the structure in Figure 10-4 and becomes scared that the computer won't take either of the `true` detours.

  If (for some unforeseen reason) both of the `if` statements' conditions are `false`, then the variable `price` isn't assigned a value. So, without an initialization, `price` has no value. (More precisely, `price` has no value that's intentionally given to it in the code.)

> Eventually, the computer reaches the `System.out.print(price)` statement. It can't display `price` unless `price` has a meaningful value. At that point, the compiler throws up its virtual hands in disgust.

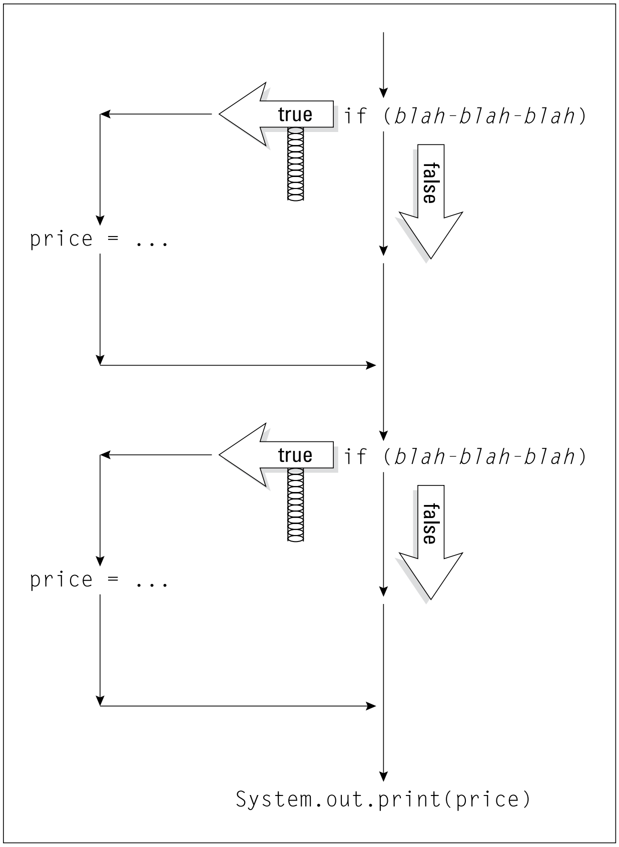


FIGURE 10-4: The choices in Listing 10-1.

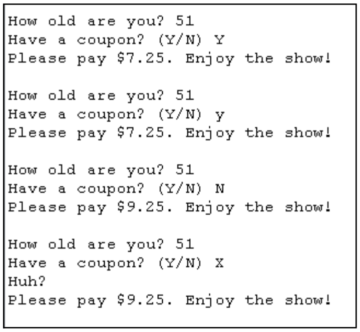
More and more conditions

Last night I had a delicious meal at the neighborhood burger joint. As part of a promotion, I got a discount coupon along with the meal. The coupon is good for $2.00 off the price of a ticket at the local movie theater.

To make use of the coupon in the `TicketPrice` program, I have to tweak the code in Listing 10-1. The revised code is in Listing 10-2. In Figure 10-5, I take that new code around the block a few times.

LISTING 10-2: Do You Have a Coupon?

```
import java.util.Scanner;

public class TicketPriceWithDiscount {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        int age;
        double price = 0.00;
        char reply;

        System.out.print("How old are you? ");
        age = keyboard.nextInt();

        System.out.print("Have a coupon? (Y/N) ");
        reply = keyboard.findWithinHorizon(".", 0).charAt(0);

        if (age >= 12 && age < 65) {
            price = 9.25;
        }
        if (age < 12 || age >= 65) {
            price = 5.25;
        }

        if (reply == 'Y' || reply == 'y') {
            price -= 2.00;
        }
        if (reply != 'Y' && reply != 'y' && reply!='N' && reply!='n') {
            System.out.println("Huh?");
        }

        System.out.print("Please pay $");
        System.out.print(price);
        System.out.print(". ");
        System.out.println("Enjoy the show!");

        keyboard.close();
    }
}
```



```
How old are you? 51
Have a coupon? (Y/N) Y
Please pay $7.25. Enjoy the show!

How old are you? 51
Have a coupon? (Y/N) y
Please pay $7.25. Enjoy the show!

How old are you? 51
Have a coupon? (Y/N) N
Please pay $9.25. Enjoy the show!

How old are you? 51
Have a coupon? (Y/N) X
Huh?
Please pay $9.25. Enjoy the show!
```

FIGURE 10-5: Running the code in Listing 10-2.

Listing 10-2 has two `if` statements whose conditions involve characters:

- In the first such statement, the computer checks to see whether the `reply` variable stores the letter `Y` or the letter `y`. If either is the case, it subtracts 2.00 from the `price`. (For information on operators like `-=`, see Chapter 7.)
- The second such statement has a hefty condition. The condition tests to see whether the `reply` variable stores any reasonable value. If the reply *isn't* `Y` *and isn't* `y` *and isn't* `N` *and isn't* `n`, then the computer expresses its concern by displaying, "Huh?" (If you're a paying customer, the word *Huh?* on the automated ticket teller's screen will certainly get your attention.)

TIP

When you create a big, multipart condition, you always have several ways to think about the condition. For example, you can rewrite the last condition in Listing 10-2 as `if (!(reply == 'Y' || reply == 'y' || reply == 'N' || reply == 'n'))`. "*If it's not the case that* the `reply` is *either* `Y`, `y`, `N`, *or* `n`, then display 'Huh?'" So, which way of writing the condition is better — the way I do it in Listing 10-2 or the way I do it in this tip? It depends on your taste. Whichever makes the logic easier for you to understand is the better way.

A condition always reveals its secrets

No matter how good a program is, you can always make it a little better. Take the code in Listing 10-2. Does the forest of `if` statements make you nervous? Do you slow to a crawl when you read each condition? Wouldn't it be nice if you could glance at a condition and make sense of it very quickly?

To some extent, you can. If you're willing to create some additional variables, you can make your code easier to read. Listing 10-3 shows you how.

LISTING 10-3: George Boole Would Be Proud

```java
import java.util.Scanner;

public class NicePrice {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        int age;
        double price = 0.00;
        char reply;
        boolean isKid, isSenior, hasCoupon, hasNoCoupon;

        System.out.print("How old are you? ");
        age = keyboard.nextInt();

        System.out.print("Have a coupon? (Y/N) ");
        reply = keyboard.findWithinHorizon(".", 0).charAt(0);

        isKid = age < 12;
        isSenior = age >= 65;
        hasCoupon = reply == 'Y' || reply == 'y';
        hasNoCoupon = reply == 'N' || reply == 'n';

        if (!isKid && !isSenior) {
            price = 9.25;
        }
        if (isKid || isSenior) {
            price = 5.25;
        }
        if (hasCoupon) {
            price -= 2.00;
        }
        if (!hasCoupon && !hasNoCoupon) {
            System.out.println("Huh?");
        }

        System.out.print("Please pay $");
        System.out.print(price);
        System.out.print(". ");
        System.out.println("Enjoy the show!");

        keyboard.close();
    }
}
```

Runs of the code in Listing 10-3 look like the stuff in Figure 10-5. The only difference between Listings 10-2 and 10-3 is the use of boolean variables. In Listing 10-3, you get past all the less-than signs and double equal signs before the start of any if statements. By the time you encounter the two if statements, the conditions can use simple words — words like isKid, isSenior, and hasCoupon. With all these boolean variables, expressing each if statement's condition is a snap. You can read more about boolean variables in Chapter 8.

Adding a boolean variable can make your code more manageable. But because some programming languages have no boolean variables, many programmers prefer to create if conditions on the fly. That's why I mix the two techniques (conditions with and without boolean variables) in this book.

Mixing different logical operators together

If you read about Listing 10-2, you know that my local movie theater offers discount coupons. The trouble is, I can't use a coupon along with any other discount. I tried to convince the ticket taker that I'm under 12 years of age, but he didn't buy it. When that didn't work, I tried combining the coupon with the senior citizen discount. That didn't work, either.

Apparently, the theater uses some software that checks for people like me. It looks something like the code in Listing 10-4. To watch the code run, take a look at Figure 10-6.

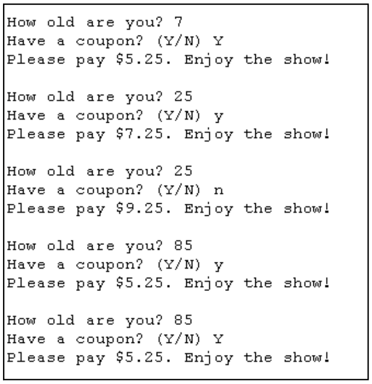
LISTING 10-4: No Extra Break for Kids or Seniors

```
import java.util.Scanner;

public class CheckAgeForDiscount {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        int age;
        double price = 0.00;
        char reply;

        System.out.print("How old are you? ");
        age = keyboard.nextInt();

        System.out.print("Have a coupon? (Y/N) ");
        reply = keyboard.findWithinHorizon(".", 0).charAt(0);
```

(continued)

LISTING 10-4: ***(continued)***

```
        if (age >= 12 && age < 65) {
            price = 9.25;
        }
        if (age < 12 || age >= 65) {
            price = 5.25;
        }

        if ((reply == 'Y' || reply == 'y') && (age >= 12 && age < 65)) {
            price -= 2.00;
        }

        System.out.print("Please pay $");
        System.out.print(price);
        System.out.print(". ");
        System.out.println("Enjoy the show!");

        keyboard.close();
    }
}
```



```
How old are you? 7
Have a coupon? (Y/N) Y
Please pay $5.25. Enjoy the show!

How old are you? 25
Have a coupon? (Y/N) y
Please pay $7.25. Enjoy the show!

How old are you? 25
Have a coupon? (Y/N) n
Please pay $9.25. Enjoy the show!

How old are you? 85
Have a coupon? (Y/N) y
Please pay $5.25. Enjoy the show!

How old are you? 85
Have a coupon? (Y/N) Y
Please pay $5.25. Enjoy the show!
```

FIGURE 10-6: Running the code in Listing 10-4.

Listing 10-4 is a lot like its predecessors, Listings 10-1 and 10-2. The big difference is the bolded `if` statement. This `if` statement tests two things, and each thing has two parts of its own:

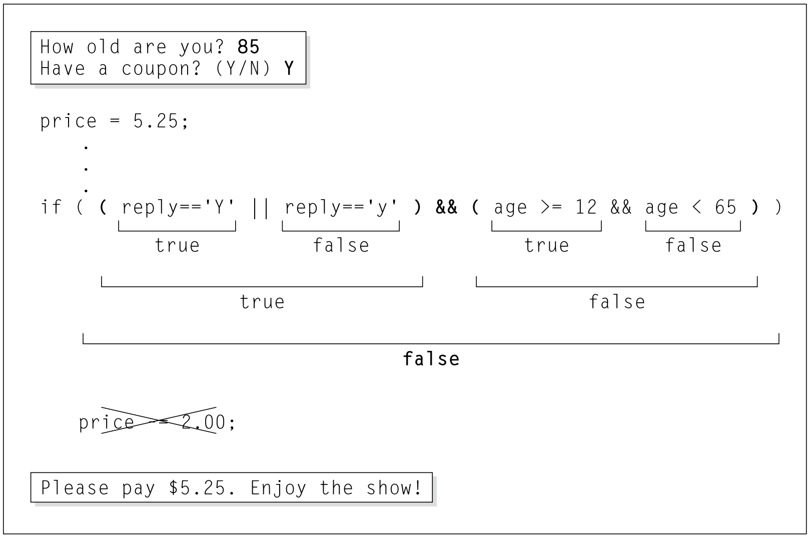
- **Does the customer have a coupon?**

  That is, did the customer reply with either `Y` *or* `y`?

- **Is the customer in the regular age group?**

  That is, is the customer at least 12 years old *and* younger than 65?

In Listing 10-4, I join items 1 and 2 using the && operator. I do this because both items (item 1 *and* item 2) must be true in order for the customer to qualify for the $2.00 discount, as illustrated in Figure 10-7.



```
How old are you? 85
Have a coupon? (Y/N) Y
```

```
price = 5.25;
    .
    .
    .
if ( ( reply=='Y' || reply=='y' ) && ( age >= 12 && age < 65 ) )
```

```
Please pay $5.25. Enjoy the show!
```

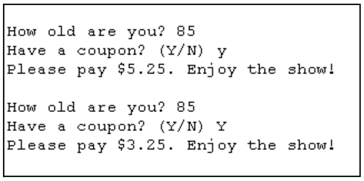
FIGURE 10-7: Both the reply and the age criteria must be true.

The mating calls of left and right parentheses

Listing 10-4 demonstrates something important about conditions: Sometimes, you need parentheses to make a condition work correctly. Take, for example, the following incorrect `if` statement:

```
//This code is incorrect:
if (reply == 'Y' || reply == 'y' && age >= 12 && age < 65) {
    price -= 2.00;
}
```

Compare this code with the correct code in Listing 10-4. This incorrect code has no parentheses to group `reply == 'Y'` with `reply == 'y'`, or to group `age >= 12` with `age < 65`. The result is the bizarre pair of runs in Figure 10-8.



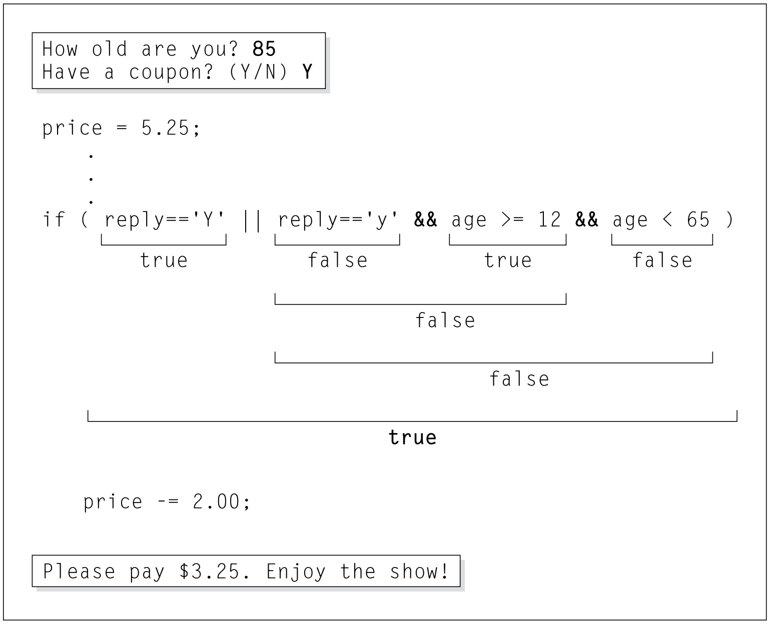
```
How old are you? 85
Have a coupon? (Y/N) y
Please pay $5.25. Enjoy the show!

How old are you? 85
Have a coupon? (Y/N) Y
Please pay $3.25. Enjoy the show!
```

FIGURE 10-8: A capital offense.

In Figure 10-8, notice that the `y` and `Y` inputs yield different ticket prices, even though the age is 85 in both runs. This happens because, without parentheses, any `&&` operator gets evaluated before any `||` operator. (That's the rule in the Java programming language — evaluate `&&` before `||`.) When `reply` is `Y`, the condition in the bad `if` statement takes the following form:

```
reply == 'Y' || some-other-stuff-that-does-not-matter
```

Whenever `reply == 'Y'` is `true`, the whole condition is automatically `true`, as illustrated in Figure 10-9.



```
How old are you? 85
Have a coupon? (Y/N) Y
```

```
price = 5.25;
    .
    .
    .
if ( reply=='Y' || reply=='y' && age >= 12 && age < 65 )

    price -= 2.00;
```

```
Please pay $3.25. Enjoy the show!
```

FIGURE 10-9: "True or false" makes "true."

Building a Nest

The year is 1968 and *The Prisoner* is on TV. In the last episode, the show's hero meets his nemesis, Number One. At first, Number One wears a spooky happy-face/sad-face mask, and when the mask comes off, there's a monkey mask underneath. To find out what's behind the monkey mask, you have to rent the series and watch it yourself. But in the meantime, notice the layering: a mask within a mask. You can do the same kind of thing with `if` statements. This section's example shows you how.

But first, take a look at Listing 10-4. In that code, the condition `age >= 12 && age < 65` is tested twice.

```
if (age >= 12 && age < 65) {
    price = 9.25;
}

// ... and a bit later ...

if ((reply == 'Y' || reply == 'y') && (age >= 12 && age < 65)) {
    price -= 2.00;
}
```

Both times, the computer sends 12, 65, and the value of `age` through its jumble of circuits; and both times, the computer gets the same answer. This is wasteful, but waste isn't your only concern.

What if you decide to change the age limit for senior tickets? From now on, no one under 100 gets a senior discount. You fish through the code and see the first `age >= 12 && age < 65` test. You change `65` to `100`, pat yourself on the back, and go home. The problem is, you've changed one of the two `age >= 12 && age < 65` tests, but you haven't changed the other. Wouldn't it be better to keep all the `age >= 12 && age < 65` testing in just one place?

Listing 10-5 comes to the rescue. In Listing 10-5, I smoosh all my `if` statements together into one big glob. The code is dense, but it gets the job done nicely.

LISTING 10-5: Nested if Statements

```
import java.util.Scanner;

public class AnotherAgeCheck {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        int age;
        double price = 0.00;
        char reply;

        System.out.print("How old are you? ");
        age = keyboard.nextInt();

        System.out.print("Have a coupon? (Y/N) ");
        reply = keyboard.findWithinHorizon(".", 0).charAt(0);
```

(continued)

LISTING 10-5: *(continued)*

```
        if (age >= 12 && age < 65) {
            price = 9.25;
            if (reply == 'Y' || reply == 'y') {
                price -= 2.00;
            }
        } else {
            price = 5.25;
        }

        System.out.print("Please pay $");
        System.out.print(price);
        System.out.print(". ");
        System.out.println("Enjoy the show!");

        keyboard.close();
    }
}
```

The best of the nest

A run of the code in Listing 10-5 looks identical to a run of Listing 10-4. You can see several runs in Figure 10-6. The main idea in Listing 10-5 is to put an `if` statement inside another `if` statement. After all, Chapter 9 says that an `if` statement can take the following form:

```
if (Condition) {
    SomeStatements
} else {
    OtherStatements
}
```

Who says *SomeStatements* can't contain an `if` statement? For that matter, *OtherStatements* can also contain an `if` statement. And, yes, you can create an `if` statement within an `if` statement within an `if` statement. There's no predefined limit on the number of `if` statements you can have.

```
if (age >= 12 && age < 65) {
    price = 9.25;
    if (reply == 'Y' || reply == 'y') {
        if (isSpecialFeature) {
           price -= 1.00;
        } else {
            price -= 2.00;
        }
    }
} else {
    price = 5.25;
}
```

When you put one if statement inside another, you create *nested* if statements. Nested statements aren't difficult to write, as long as you take things slowly and keep a clear picture of the code's flow in your mind. If it helps, draw yourself a diagram like the one shown in Figure 10-10.

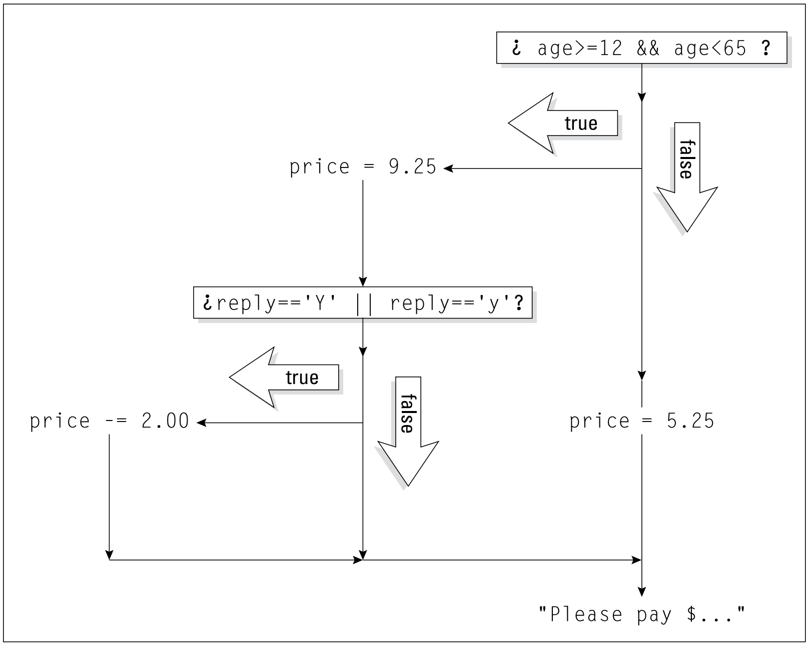
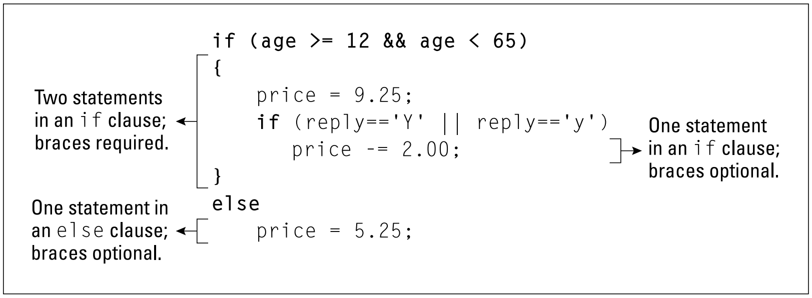


FIGURE 10-10: The flow in Listing 10-5.

WARNING

When you nest statements, you must be compulsive about the use of indentation and braces. (See Figure 10-11.) When code has misleading indentation, no one (not even the programmer who wrote the code) can figure out how the code works. A nested statement with sloppy indentation is a programmer's nightmare.



```
if (age >= 12 && age < 65)
{
    price = 9.25;
    if (reply=='Y' || reply=='y')
       price -= 2.00;
}
else
    price = 5.25;
```

Two statements in an if clause; braces required.
One statement in an if clause; braces optional.
One statement in an else clause; braces optional.

FIGURE 10-11: Be careful about adding the proper indentation and braces.

Cascading if statements

Here's a riddle: You have two baseball teams — the Hankees and the Socks. You want to display the teams' scores on two separate lines, with the winner's score listed first. (On the computer screen, the winner's score is displayed above the loser's score.) What happens when the scores are tied?

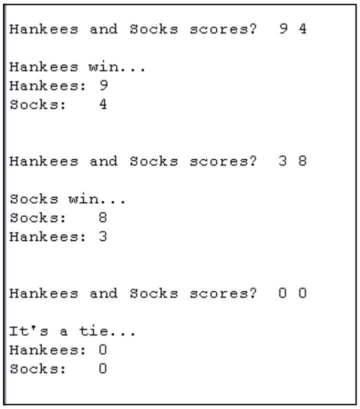
Do you give up? The answer is, there's no right answer. What happens depends on the way you write the program. Take a look at Listing 9-4, in Chapter 9. When the scores are equal, the condition `hankees > socks` is `false`. So the program's flow of execution drops down to the `else` clause. That clause displays the Socks score first and the Hankees score second. (Refer to Figure 9-7, in Chapter 9.)

The program doesn't have to work this way. If I take Listing 9-4 and change `hankees > socks` to `hankees >= socks`, then, in case of a tie, the Hankees score comes first.

Suppose that you want a bit more control. When the scores are equal, you want to see a message indicating a tie. To do this, think in terms of a three-pronged fork. You have a prong for a Hankees win, another prong for a Socks win, and a third prong for a tie. You can write this code in several different ways, but one way that makes lots of sense is in Listing 10-6. For three runs of the code in Listing 10-6, see Figure 10-12.

LISTING 10-6: In Case of a Tie . . .

```
import java.util.Scanner;

import static java.lang.System.out;

public class WinLoseOrTie {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        int hankees, socks;

        out.print("Hankees and Socks scores?  ");
        hankees = keyboard.nextInt();
        socks = keyboard.nextInt();
        out.println();

        if (hankees > socks) {
            out.println("Hankees win...");
            out.print("Hankees: ");
            out.println(hankees);
            out.print("Socks:   ");
            out.println(socks);
        } else if (socks > hankees) {
            out.println("Socks win...");
            out.print("Socks:   ");
            out.println(socks);
            out.print("Hankees: ");
            out.println(hankees);
        } else {
            out.println("It's a tie...");
            out.print("Hankees: ");
            out.println(hankees);
            out.print("Socks:   ");
            out.println(socks);
        }

        keyboard.close();
    }
}
```



```
Hankees and Socks scores?  9 4

Hankees win...
Hankees: 9
Socks:   4


Hankees and Socks scores?  3 8

Socks win...
Socks:   8
Hankees: 3


Hankees and Socks scores?  0 0

It's a tie...
Hankees: 0
Socks:   0
```

FIGURE 10-12: Go, team, go!

Listing 10-6 illustrates a way of thinking about a problem. You have one question with more than two answers. (In this section's baseball problem, the question is "Who wins?" and the answers are "Hankees," "Socks," or "Neither.") The problem begs for an `if` statement, but an `if` statement has only two branches — the `true` branch and the `false` branch. So you combine alternatives to form *cascading if statements.*

In Listing 10-6, the format of the cascading `if` statements is

```
if (Condition1) {
    SomeStatements
} else if (Condition2) {
    OtherStatements
} else {
    EvenMoreStatements
}
```

In general, you can use `else if` as many times as you want:

```
if (hankeesWin) {
    out.println("Hankees win...");
    out.print("Hankees: ");
    out.println(hankees);
    out.print("Socks:   ");
    out.println(socks);
} else if (socksWin) {
    out.println("Socks win...");
    out.print("Socks:   ");
    out.println(socks);
    out.print("Hankees: ");
    out.println(hankees);
} else if (isATie) {
    out.println("It's a tie...");
    out.print("Hankees: ");
    out.println(hankees);
    out.print("Socks:   ");
    out.println(socks);
} else if (gameCancelled) {
    out.println("Sorry, sports fans.");
} else {
    out.println("The game isn't over yet.");
}
```

Nothing is special about cascading `if` statements. This isn't a new programming language feature. Cascading `if` statements take advantage of a loophole in Java — a loophole about omitting curly braces in certain circumstances. Other than that, cascading `if` statements just give you a new way to think about decisions within your code.

Note: Listing 10-6 uses a static import declaration to avoid needless repetition of the words `System.out`. To read a little bit about the static import declaration (along with an apology for my not explaining this concept more thoroughly), see Chapter 9. Then to get the real story on static import declarations, see Chapter 14.

Enumerating the Possibilities

Chapter 8 describes Java's `boolean` type — the type with only two values (`true` and `false`). The `boolean` type is handy, but sometimes you need more values. After all, a traffic light's values can be `green`, `yellow`, or `red`. A playing card's suit can be `spade`, `club`, `heart`, or `diamond`. And a weekday can be `Monday`, `Tuesday`, `Wednesday`, `Thursday`, or `Friday`.

Life is filled with small sets of possibilities, and Java has a feature that can reflect these possibilities. The feature is called an `enum` type.

Creating an enum type

The story in Listing 10-6 has three possible endings — the Hankees win, the Socks win, or the game is tied. You can represent the possibilities with the following line of Java code:

```
enum WhoWins {home, visitor, neither}
```

This week's game is played at Hankeeville's SnitSoft Stadium, so the value `home` represents a win for the Hankees, and the value `visitor` represents a win for the Socks.

One goal in computer programming is for each program's structure to mirror whatever problem the program solves. When a program reminds you of its underlying problem, the program is easy to understand and inexpensive to maintain. For example, a program to tabulate customer accounts should use names like `customer` and `account`. And a program that deals with three possible outcomes (home wins, visitor wins, and tie) should have a variable with three possible values. The line `enum WhoWins{home, visitor, neither}` creates a type to store three values.

The `WhoWins` type is called an *enum type.* Think of the new `WhoWins` type as a `boolean` on steroids. Instead of two values (`true` and `false`), the `WhoWins` type has three values (`home`, `visitor`, and `neither`). You can create a variable of type `WhoWins`:

```
WhoWins who;
```

and then assign a value to the new variable:

```
who = WhoWins.home;
```

In the next section, I put the `WhoWins` type to good use.

Using an enum type

Listing 10-7 shows you how to use the brand-new `WhoWins` type.

LISTING 10-7: Proud Winners and Sore Losers

```
import java.util.Scanner;

import static java.lang.System.out;

public class Scoreboard {

    enum WhoWins {home, visitor, neither}

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        int hankees, socks;
        WhoWins who;

        out.print("Hankees and Socks scores?  ");
        hankees = keyboard.nextInt();
        socks = keyboard.nextInt();
        out.println();

        if (hankees > socks) {
            who = WhoWins.home;
            out.println("The Hankees win :-)");
        } else if (socks > hankees) {
            who = WhoWins.visitor;
            out.println("The Socks win :-(");
        } else {
            who = WhoWins.neither;
            out.println("It's a tie :-|");
        }

        out.println();
        out.println("""
                Today's game is brought to you by
                SnitSoft, the number one software
                vendor in the Hankeeville area.
                SnitSoft is featured proudly in
                Chapter 6. And remember, four out
                of five doctors recommend
                SnitSoft to their patients.""");
        out.println();

        if (who == WhoWins.home) {
            out.println("We beat 'em good. Didn't we?");
        }

        if (who == WhoWins.visitor) {
            out.println("The umpire made an unfair call.");
        }

        if (who == WhoWins.neither) {
            out.println("The game goes into extra innings.");
        }

        keyboard.close();
    }
}
```

Three runs of the program in Listing 10-7 are pictured in Figure 10-13.

Here's what happens in Listing 10-7:

- **I create a variable to store values of type `WhoWins`.**

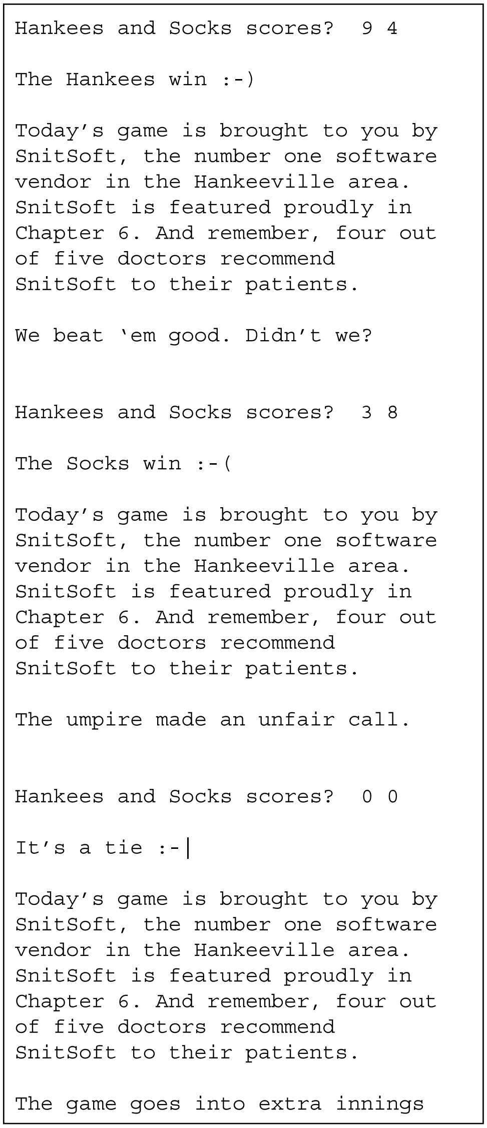
  Just as the line

  ```
  double amount;
  ```

  declares `amount` to store `double` values (values like `5.95` and `30.95`), the line

  ```
  WhoWins who;
  ```

  declares `who` to store `WhoWins` values (values like `home`, `visitor`, and `neither`).



```
Hankees and Socks scores?  9 4

The Hankees win :-)

Today's game is brought to you by
SnitSoft, the number one software
vendor in the Hankeeville area.
SnitSoft is featured proudly in
Chapter 6. And remember, four out
of five doctors recommend
SnitSoft to their patients.

We beat 'em good. Didn't we?


Hankees and Socks scores?  3 8

The Socks win :-(

Today's game is brought to you by
SnitSoft, the number one software
vendor in the Hankeeville area.
SnitSoft is featured proudly in
Chapter 6. And remember, four out
of five doctors recommend
SnitSoft to their patients.

The umpire made an unfair call.


Hankees and Socks scores?  0 0

It's a tie :-|

Today's game is brought to you by
SnitSoft, the number one software
vendor in the Hankeeville area.
SnitSoft is featured proudly in
Chapter 6. And remember, four out
of five doctors recommend
SnitSoft to their patients.

The game goes into extra innings
```

FIGURE 10-13: Joy in Hankeeville?

- **I assign a value to the** `who` **variable.**

  I execute one of the

  ```
  who = WhoWins.something;
  ```

  assignment statements. The statement that I execute depends on the outcome of the `if` statement's `hankees > socks` comparison.

Notice that I refer to each of the WhoWins values in Listing 10-7. I write WhoWins.home, WhoWins.visitor, or WhoWins.neither. If I forget the WhoWins prefix and type

```
who = home; //This assignment doesn't work!
```

the compiler gives me a home cannot be resolved to a variable error message. That's just the way enum types work.

- **I compare the variable's value with each of the WhoWins values.**

  In one if statement, I check the who == WhoWins.home condition. In the remaining two if statements, I check for the other WhoWins values.

Near the end of Listing 10-7, I could have done without enum values. I could have tested things like hankees > socks a second time:

```
if (hankees > socks) {
    out.println("The Hankees win :-)");
}

// And later in the program ...

if (hankees > socks) {
    out.println("We beat 'em good. Didn't we?");
}
```

But that tactic would be clumsy. In a more complicated program, I may end up checking hankees > socks a dozen times. It would be like asking the same question over and over again.

Rather than repeatedly check the hankees > socks condition, I store the game's outcome as an enum value. Then I check the enum value as many times as I want. That's a tidy way to solve the repeated checking problem.

When One Line Isn't Enough

Listing 10-7 uses a feature that didn't become an official part of Java until September 2020 (with Java 15). A *text block* is a bunch of text surrounded on both sides by three double quotes (""").

```
out.println("""
        Today's game is brought to you by
        SnitSoft, the number one software
        ...
        of five doctors recommend
        SnitSoft to their patients.""");
```

A text block starts with three double quotation marks, and the remainder of that block's first line must be blank. If you mistakenly put text after those first three quotation marks, Java becomes sick to its stomach:

```
// Don't do this:
out.println("""Today's game is brought to you by
        SnitSoft, the number one software
        vendor in the Hankeeville area.""");
```

Text blocks are useful because the text inside a block can straddle more than one line. Without text blocks, you may be tempted to put one quotation mark at each end, but that doesn't work. The following code, with traditional Java string notation, is forbidden:

```
// This code is incorrect:
out.println("
        Today's game is brought to you by
        SnitSoft, the number one software
        vendor in the Hankeeville area.");
```

This book's 5th edition hit the shelves in 2017 before Java had text blocks. In that edition, Listing 10-7 had a truckload of `out.println` calls:

```
out.println("Today's game is brought to you by");
out.println("SnitSoft, the number one software");
out.println("vendor in the Hankeeville area.");
out.println("SnitSoft is featured proudly in");
out.println("Chapter 6. And remember, four out");
out.println("of five doctors recommend");
out.println("SnitSoft to their patients.");
```

That was very ugly code! In retrospect, I could have used an escape sequence — a trick that's featured in the very next section.

Escape to the \next li\ne o\n the scree\n

Robert Louis Stevenson was born in 1850 and died in 1894. His poem "Bed in Summer" started with these two lines:

- In winter I get up at night
- And dress by yellow candle-light.

Stevenson never wrote any Java code. But, if he could, he might have composed the lines this way:

```
out.println("In winter I get up at night\nAnd dress by yellow candle-light.");
```

The `\n` combination of characters is an example of an *escape sequence*. Think of `\n` as a temporary escape from the burden of displaying exactly the characters in the quoted string. When you run the code, you don't see `night\nAnd` in IntelliJ's Run tool window. Instead, you see the word `night` followed by the start of a new line and then the word `And`. It's as if the string of characters looked like this:

```
// This is fake code:
"In winter I get up at night(Go to the next line)And dress by ..."
```

You can replace the text block in Listing 10-7 with one long line of code:

```
out.println("Today's game is brought to you by\nSnitSoft, the number one ...
```

Imagine that this line of code is 241 characters wide. (In this book's seventh edition, a holographic image will extend this line far to the right of the book's physical page!)

If you don't like long lines of code, you can combine the use of escape sequences and string concatenation:

```
out.println("Today's game is brought to you by\n" +
            "SnitSoft, the number one software\n" +
            "vendor in the Hankeeville area.\n" +
            "SnitSoft is featured proudly in\n" +
            "Chapter 6. And remember, four out\n" +
            "of five doctors recommend\n" +
            "SnitSoft to their patients.");
```

Code of this kind isn't pleasant to write. When I write it, I always make mistakes. So why in the world would you want to know about the `\n` escape sequence? I can think of three reasons:

- The `\n` sequence appears in millions of existing Java programs.
- The `\n` sequence works in many other programming languages including C/C++, JavaScript, Python, Ruby, and R.

- Java has several other kinds of escape sequences. Even if you never use \n, there's no escaping the use of escape sequences in computer programming.

Table 10-2 lists some of Java's most useful escape sequences:

TABLE 10-2: Escape Sequences

Escape sequence	Meaning	Where to read about it
\n	New line	In this section
\"	Double quote	In the next section (Keep reading!)
\t	Tab	In Chapter 11
\s	Blank space	In Chapter 14
\\	Backslash	In Chapter 16

More escapism

The poet Henry Wadsworth Longfellow was born in 1807 and died in 1882. Who knows? Maybe Longfellow did some computer programming during his lifetime. If so, I hope he didn't write this `println` call:

```
// Think of this ode As very bad code.
out.println("Then he said, "Good night!" and with muffled oar\n" +
            "Silently rowed to the Charlestown shore,");
```

In this code, the first line of poetry has four quote marks. That makes it impossible for Java to decide where the line of poetry starts and ends. (See Figure 10-14.)

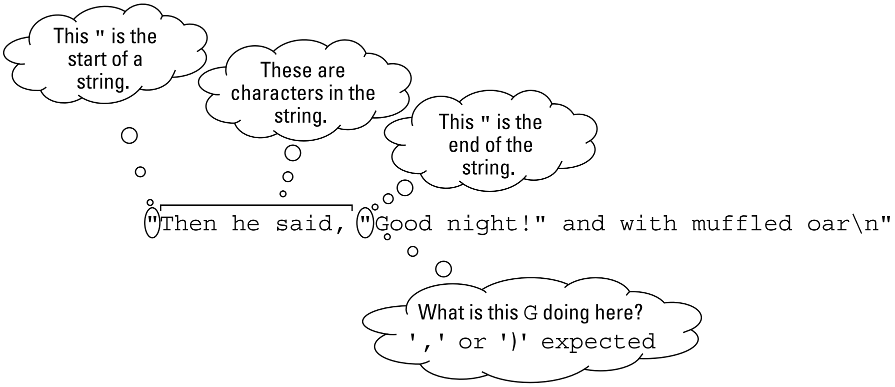

This " is the start of a string.
These are characters in the string.
This " is the end of the string.
"Then he said, "Good night!" and with muffled oar\n"
What is this G doing here?
',' or ')' expected

FIGURE 10-14: Java is confused.

Wouldn’t it be nice if there was a way to tell Java that the middle two quote marks don’t signal the boundaries of a string of characters?

Well, wha’ da’ ya’ know? There is a way! Figure 10-15 illustrates the use of Java’s \" escape sequence.

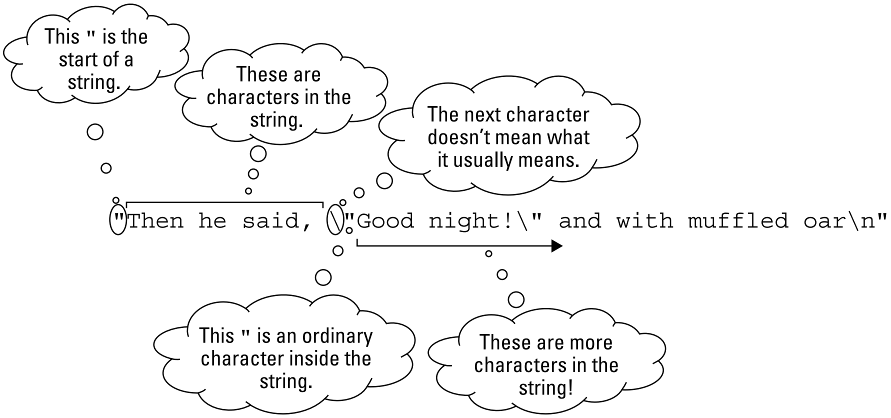

This " is the start of a string.
These are characters in the string.
The next character doesn’t mean what it usually means.
"Then he said, \"Good night!\" and with muffled oar\n"
This " is an ordinary character inside the string.
These are more characters in the string!

FIGURE 10-15: Java understands everything.

In Figure 10-15, each backslash tells Java to escape from the usual interpretation of a double quotation mark. So Java doesn’t think of the quotation mark as the beginning or end of a string. Instead, Java thinks of it as an ordinary character — a character to be displayed in IntelliJ’s Run tool window.

Of course, this isn’t the end of the escape sequence story. For more escapist entertainment, see Chapters 14 and 16.

TRY IT OUT

It’s okay to read about Java, but it’s even better if you *work* with Java. Here are some things you can do to flex your Java muscles:

MYSTERIOUS WAYS

Explain why the following code always displays `The first is smaller`, no matter which numbers the user types. For example, if the user types 7 and then 5, the program displays `The first is smaller`:

```
int firstNumber = keyboard.nextInt();
int secondNumber = keyboard.nextInt();
boolean firstSmaller = firstNumber < secondNumber;

if (firstSmaller = true) {
    System.out.println("The first is smaller.");
}
```

WHAT KIND OF NUMBER?

Write a program that reads a number from the keyboard and displays one of the words `positive`, `negative`, or `zero` to describe that number. Use cascading `if` statements.

APPROACHING A TRAFFIC SIGNAL

Your driver's handbook says, "When approaching a green light, proceed through the intersection unless it's unsafe to do so, or unless a police officer directs you to do otherwise."

Write a program that asks the user three questions:

- Are you approaching a green light?
- Is it safe to proceed through the intersection?
- Is a police officer directing you not to proceed?

In response to each question, the user replies `Y` or `N`. Based on the three replies, the program displays either `Go` or `Stop`.

Needless to say, you shouldn't run this program while you drive a vehicle.

"YES" AND "YES" AGAIN

Modify the program in the earlier task "Approaching a traffic signal" so that the program allows the user to reply "yes" with either an uppercase letter `Y` or a lowercase letter `y`.

RED LIGHT OR YELLOW LIGHT

Modify the program in the "Approaching a traffic signal" task so that the program asks `What color is the traffic light? (G/Y/R)`. When the user replies `Y` or `R`, and either it's unsafe to proceed or an officer is directing drivers not to proceed, the program displays `Stop`. Otherwise, the program doesn't display anything.

WHAT? ANOTHER TRAFFIC SIGNAL PROGRAM?

You can use `System.out.println` to display an `enum` value. For example, in Listing 10-7, if you add the statement

```
System.out.println(who);
```

to the end of the `main` method, the program displays one of the words `home`, `visitor`, or `neither`. Try this by creating an `enum` named `Color` with values `green`, `yellow`, and `red`. Write a program that asks `What color is the traffic light? (G/Y/R)`. Use the user's response to assign one of the values `Color.green`, `Color.yellow`, or `Color.red` to a variable named `signal`. Use `System.out.println` to display the value of the `signal` variable.

BUYING 3D GLASSES

My local movie theater charges an extra three dollars for a movie showing in 3D. (The theater makes me buy a new pair of 3D glasses. I can't bring the pair that I bought the last time I saw a 3D movie.) Modify the code in Listing 10-5 so that the program asks `How many dimensions: 2 or 3?`. For a 3D movie, the program adds three dollars to the price of admission. (***Note:*** The old *Twilight Zone* television series began with narrator Rod Serling talking about a fifth dimension. I wonder what I'd have to pay nowadays to see the *Twilight Zone* in my local movie theater!)

GOING ALL \N

In Listing 10-7, I surround the SnitSoft advertisement with two `out.println()` lines. Neither line has anything inside the call's parentheses. Modify the code so that it has exactly the same output but doesn't use either of these `out.println()` lines. Make sure to display blank lines before and after the SnitSoft advertisement.

IN THIS CHAPTER

- Creating program loops
- Formulating solutions to problems with loops
- Diagnosing loop problems

Chapter 11

Around and Around It Goes

Chapter 8 has code to reverse the letters in a four-letter word that the user enters. In case you haven't read Chapter 8 or you just don't want to flip to it, here's a quick recap of the code:

```
c1 = keyboard.findWithinHorizon(".",0).charAt(0);
c2 = keyboard.findWithinHorizon(".",0).charAt(0);
c3 = keyboard.findWithinHorizon(".",0).charAt(0);
c4 = keyboard.findWithinHorizon(".",0).charAt(0);

System.out.print(c4);
System.out.print(c3);
System.out.print(c2);
System.out.print(c1);
```

The code is just dandy for words with exactly four letters, but how do you reverse a five-letter word? As the code stands, you have to add two new statements:

```
c1 = keyboard.findWithinHorizon(".",0).charAt(0);
c2 = keyboard.findWithinHorizon(".",0).charAt(0);
c3 = keyboard.findWithinHorizon(".",0).charAt(0);
```

```
c4 = keyboard.findWithinHorizon(".",0).charAt(0);
c5 = keyboard.findWithinHorizon(".",0).charAt(0);

System.out.print(c5);
System.out.print(c4);
System.out.print(c3);
System.out.print(c2);
System.out.print(c1);
```

What a drag! You add statements to a program whenever the size of a word changes! You remove statements when the input shrinks! That can’t be the best way to solve the problem. Maybe you can command a computer to add statements automatically. (But then again, maybe you can’t.)

As luck would have it, you can do something that’s even better: You can write a statement once and tell the computer to execute the statement many times. How many times? You can tell the computer to execute a statement as many times as it needs to be executed.

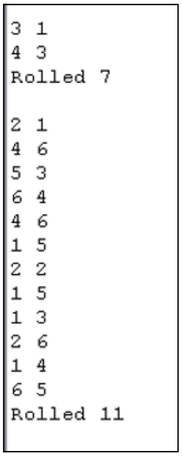
That’s the big idea. The rest of this chapter has the details.

Repeating Instructions Again and Again and Again and Again

Here’s a simple dice game: Keep rolling two dice until you roll 7 or 11. Listing 11-1 has a program that simulates the action in the game, and Figure 11-1 shows two runs of the program.

LISTING 11-1: Roll 7 or 11

```
import java.util.Random;

import static java.lang.System.out;

public class SimpleDiceGame {

  public static void main(String[] args) {
        var myRandom = new Random();
        int die1 = 0, die2 = 0;

        while (die1 + die2 != 7 && die1 + die2 != 11) {
            die1 = myRandom.nextInt(6) + 1;
            die2 = myRandom.nextInt(6) + 1;
            out.print(die1);
            out.print(" ");
            out.println(die2);
        }

        out.print("Rolled ");
        out.println(die1 + die2);
    }
}
```



```
3 1
4 3
Rolled 7

2 1
4 6
5 3
6 4
4 6
1 5
2 2
1 5
1 3
2 6
1 4
6 5
Rolled 11
```

FIGURE 11-1: Momma needs a new pair of shoes.

At the core of Listing 11-1 is a thing called a *while statement* (also known as a *while loop*). A `while` statement has the following form:

```
while (Condition) {
    Statements
}
```

Rephrased in English, the `while` statement in Listing 11-1 would say

```
while the sum of the two dice isn't 7 and isn't 11
keep doing all the stuff in curly braces: {

}
```

The stuff in curly braces (the stuff that's repeated over and over) is the code that gets two new random numbers and displays those random numbers' values.

The statements in curly braces are repeated as long as `die1 + die2 != 7 && die1 + die2 != 11` keeps being true.

Each repetition of the statements in the loop is called an *iteration* of the loop. In Figure 11-1, the first run has 2 iterations, and the second run has 12 iterations.

When `die1 + die2 != 7 && die1 + die2 != 11` is no longer true (that is, when the `sum` is either 7 or 11), the repeating of statements stops dead in its tracks. The computer marches on to the statements that come after the loop.

Following the action in a loop

To trace the action of the code in Listing 11-1, I'll borrow numbers from the first run in Figure 11-1:

- At the start, the values of `die1` and `die2` are both `0`.
- The computer reaches the top of the `while` statement and checks to see whether `die1 + die2 != 7 && die1 + die2 != 11` is true. (See Figure 11-2.) The condition is true, so the computer takes the `true` path in Figure 11-3.

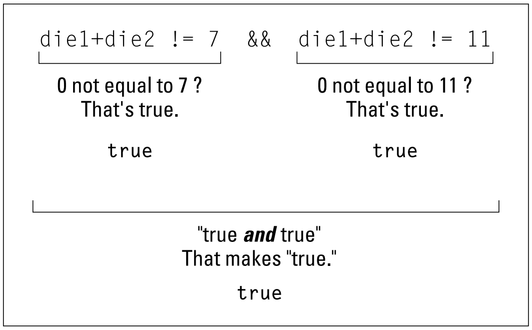
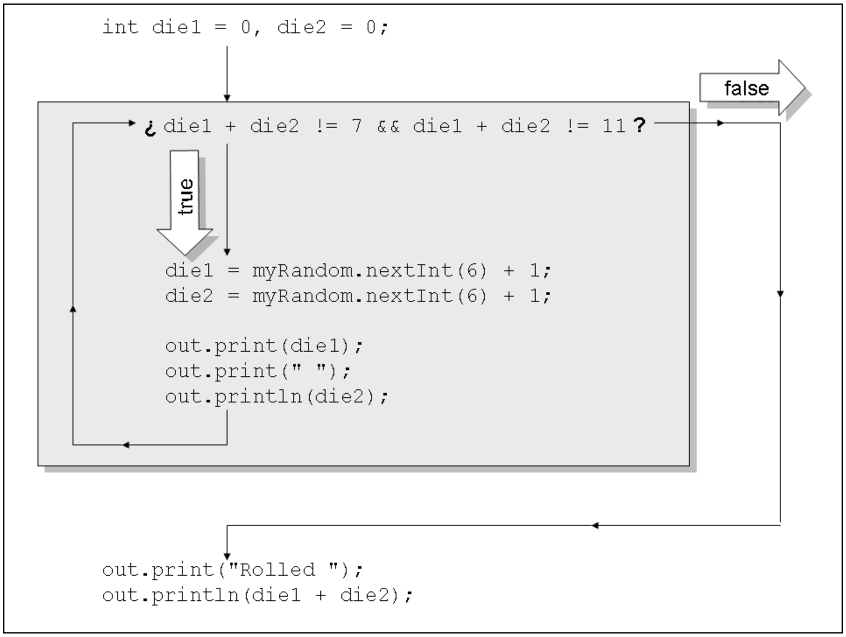


FIGURE 11-2: Two wrongs don't make a right, but two `true`s make a `true`.

The computer performs an iteration of the loop. During this iteration, the computer gets new values for `die1` and `die2` and prints those values on the screen. In the first run of Figure 11-1, the new values are `3` and `1`.

- The computer returns to the top of the `while` statement and checks to see whether `die1 + die2 != 7 && die1 + die2 != 11` is still true. The condition is true, so the computer takes the `true` path in Figure 11-3.



```
int die1 = 0, die2 = 0;

¿ die1 + die2 != 7 && die1 + die2 != 11 ?

    die1 = myRandom.nextInt(6) + 1;
    die2 = myRandom.nextInt(6) + 1;

    out.print(die1);
    out.print(" ");
    out.println(die2);

out.print("Rolled ");
out.println(die1 + die2);
```

FIGURE 11-3: Paths through the code in Listing 11-1.

The computer performs another iteration of the loop. During this iteration, the computer gets new values for `die1` and `die2` and prints those values on the screen. In Figure 11-1, the new values are 4 and 3.

- The computer returns to the top of the `while` statement and checks to see whether `die1 + die2 != 7 && die1 + die2 != 11` is still true. Lo and behold! This condition has become false! (See Figure 11-4.) The computer takes the `false` path in Figure 11-3.

  The computer leaps to the statements after the loop. The computer displays `Rolled 7` and ends its run of the program.

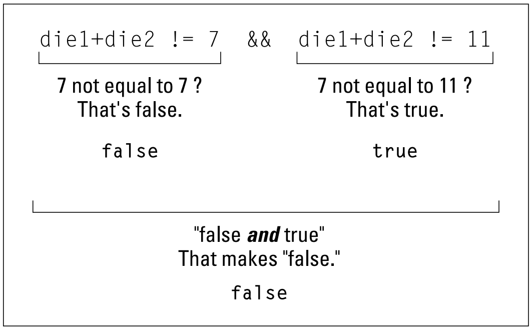


FIGURE 11-4: Look! I rolled a seven!

No early bailout

In Listing 11-1, when the computer finds `die1 + die2 != 7 && die1 + die2 != 11` to be true, the computer marches on and executes all five statements inside the loop's curly braces. The computer executes

```
die1 = myRandom.nextInt(6) + 1;
die2 = myRandom.nextInt(6) + 1;
```

STATEMENTS AND BLOCKS (PLAGIARIZING MY OWN SENTENCES FROM CHAPTER 9)

Java's `while` statements have a lot in common with `if` statements. Like an `if` statement, a `while` statement is a *compound statement* — that is, a `while` statement includes other statements within it. Also, both `if` statements and `while` statements typically include *blocks* of statements. When you surround a bunch of statements with curly braces, you get what's called a *block,* and a block behaves, in all respects, like a single statement.

In a typical `while` statement, you want the computer to repeat several smaller statements over and over again. To repeat several smaller statements, you combine those statements into one big statement. To do this, you make a block using curly braces.

In Listing 11-1, the block

```
{
  die1=myRandom.nextInt(6)+1;
  die2=myRandom.nextInt(6)+1;

  out.print(die1);
  out.print(" ");
  out.println(die2);
}
```

is a single statement. It's a statement that has, within it, five smaller statements. So this big block (this single statement) serves as one big statement inside the `while` statement in Listing 11-1.

That's the story about `while` statements and blocks. To find out how this stuff applies to `if` statements, see the "Statements and blocks" sidebar near the end of Chapter 9.

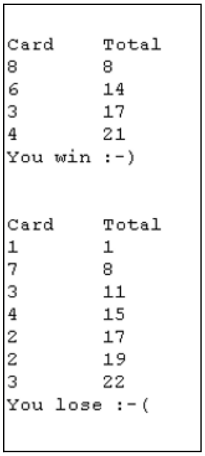
Maybe (just maybe), the new values of die1 and die2 add up to 7. Even so, the computer doesn't jump out in midloop. The computer finishes the iteration and executes

```
out.print(die1);
out.print(" ");
out.println(die2);
```

one more time. The computer performs the test again (to see whether die1+ die2 != 7 && die1 + die2 != 11 is still true) only after it fully executes all five statements in the loop.

Where Does Each Statement Belong?

Here's a simplified version of the card game Twenty-One: You keep taking cards until the total is 21 or higher. Then, if the total is 21, you win. If the total is higher, you lose. (By the way, each face card counts as a 10.) To play this game, you want a program whose runs look like the runs in Figure 11-5.



```
Card     Total
8        8
6        14
3        17
4        21
You win :-)


Card     Total
1        1
7        8
3        11
4        15
2        17
2        19
3        22
You lose :-(
```

FIGURE 11-5: You win sum; you lose sum.

In most sections of this book, I put a program's listing before the description of the program's features. But this section is different. This section deals with strategies for composing code. So in this section, I start by brainstorming about strategies.

Finding some pieces

How do you write a program that plays a simplified version of Twenty-One? I start by fishing for clues in the game's rules, spelled out in this section's first paragraph. The big fishing expedition is illustrated in Figure 11-6.

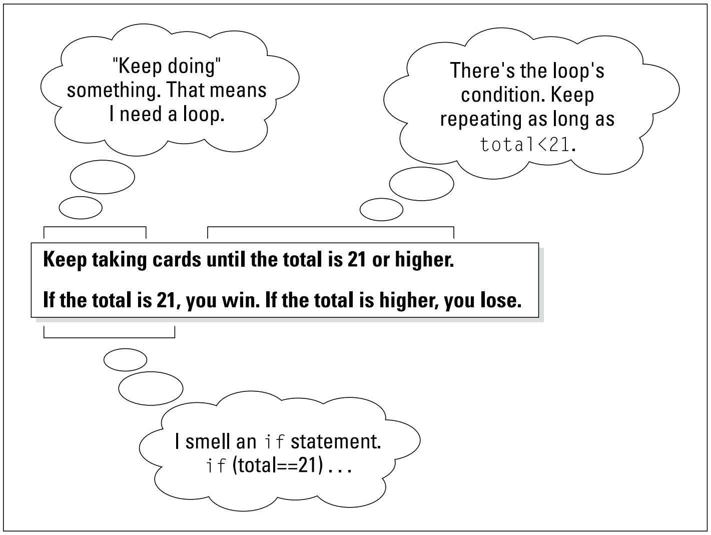

"Keep doing" something. That means I need a loop.
There's the loop's condition. Keep repeating as long as total<21.
Keep taking cards until the total is 21 or higher.
If the total is 21, you win. If the total is higher, you lose.
I smell an if statement. if (total==21) . . .

FIGURE 11-6: Thinking about a programming problem.

With the reasoning in Figure 11-6, I need a loop and an `if` statement:

```
while (total < 21) {
    //do stuff
}

if (total == 21) {
    //You win
} else {
    //You lose
}
```

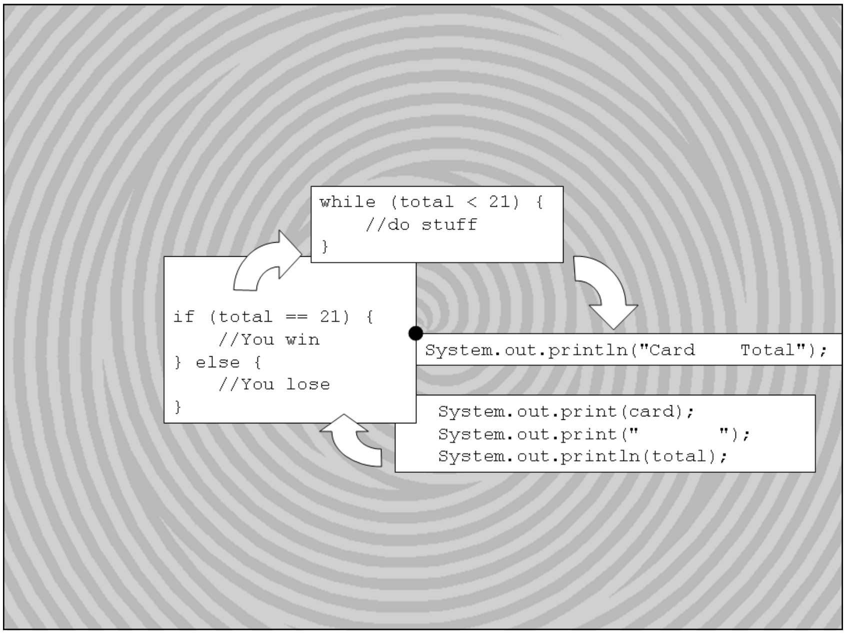
What else do I need to make this program work? Look at the sample output in Figure 11-5. I need a heading with the words `Card` and `Total`. That's a call to `System.out.println`:

```
System.out.println("Card Total");
```

I also need several lines of output — each containing two numbers. For example, in Figure 11-5, the line `6  14` displays the values of two variables. One variable stores the most recently picked card; the other variable stores the total of all cards picked so far:

```
System.out.print(card);
System.out.print("      ");
System.out.println(total);
```

Now I have four chunks of code, but I haven't decided how they all fit together. Well, you can go right ahead and call me crazy. But at this point in the process, I imagine those four chunks of code circling around one another, like part of a dream sequence in a low-budget movie. As you may imagine, I'm not very good at illustrating circling code in dream sequences. Even so, I handed my idea to the art department folks at Wiley Publishing and they came up with the picture in Figure 11-7.



```
while (total < 21) {
    //do stuff
}
```

```
if (total == 21) {
    //You win
} else {
    //You lose
}
```

```
System.out.println("Card    Total");
```

```
System.out.print(card);
System.out.print("      ");
System.out.println(total);
```

FIGURE 11-7: . . . and where they stop, nobody knows.

Assembling the pieces

Where should I put each piece of code? The best way to approach the problem is to ask how many times each piece of code should be executed:

- **The program displays `card` and `total` values more than once.** For example, in the first run of Figure 11-5, the program displays these values four times (first `8 8` and then `6 14` and so on). To get this repeated display, I put the code that creates the display inside the loop:

  ```
  while (total < 21) {
      System.out.print(card);
      System.out.print("       ");
      System.out.println(total);
  }
  ```

- **The program displays the `Card Total` heading only once per run.** This display comes before any of the repeated number displays, so I put the heading code before the loop:

  ```
  System.out.println("Card    Total");

  while (total < 21) {
      System.out.print(card);
      System.out.print("       ");
      System.out.println(total);
  }
  ```

- **The program displays `You win` or `You lose` only once per run.** This message display comes after the repeated number displays. So I put the win/lose code after the loop:

  ```
  //Preliminary draft code -- NOT ready for prime time:
  System.out.println("Card    Total");

  while (total < 21) {
      System.out.print(card);
      System.out.print("       ");
      System.out.println(total);
  }

  if (total == 21) {
      System.out.println("You win :-)");
  } else {
      System.out.println("You lose :-(");
  }
  ```

Getting values for variables

I almost have a working program. But if I take the code that I've developed for a mental test run, I face a few problems. To see what I mean, picture yourself in the computer's shoes for a minute. (Well, a computer doesn't have shoes. Picture yourself in the computer's boots.)

You start at the top of the code shown in the previous section (the code that starts with the `Preliminary draft` comment). In the code's first statement, you display the words `Card Total`. So far, so good. But then you encounter the `while` loop and test the condition `total < 21`. Well, is `total` less than `21`, or isn't it? Honestly, I'm tempted to make up an answer because I'm embarrassed about not knowing what the `total` variable's value is. (I'm sure the computer is embarrassed, too.)

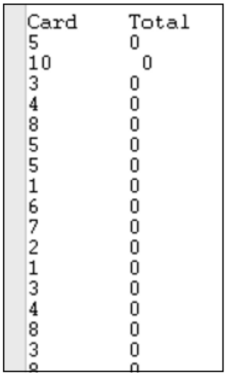
The variable `total` must have a known value before the computer reaches the top of the `while` loop. Because a player starts with no cards at all, the initial `total` value should be `0`. That settles it. I declare `int total = 0` at the top of the program.

But what about my friend the `card` variable? Should I set `card` to zero also? No. There's no zero-valued card in a deck (at least, not when I'm playing fair). Besides, `card` should get a new value several times during the program's run.

Wait! In the previous sentence, the phrase *several times* tickles a neuron in my brain. It stimulates the *inside a loop* reflex. So I place an assignment to the `card` variable inside my `while` loop:

```
//This is a DRAFT -- still NOT ready for prime time:
int card, total = 0;

System.out.println("Card    Total");

while (total < 21) {
    card = myRandom.nextInt(10) + 1;

    System.out.print(card);
    System.out.print("       ");
    System.out.println(total);
}

if (total == 21) {
    System.out.println("You win :-)");
} else {
    System.out.println("You lose :-(");
}
```

The code still has an error, and I can probably find the error with more computer role-playing. But instead, I get daring. I run this beta code to see what happens. Figure 11-8 shows part of a run.



```
Card    Total
5       0
10        0
3       0
4       0
8       0
5       0
5       0
1       0
6       0
7       0
2       0
1       0
3       0
4       0
8       0
3       0
```

FIGURE 11-8: An incorrect run.

Unfortunately, the run in Figure 11-8 doesn't stop on its own. This kind of processing is called an *infinite loop.* The loop runs and runs until someone trips over the computer's extension cord.

REMEMBER

You can stop a program's run dead in its tracks. Look for the tiny red rectangle at the top of IntelliJ's main window. When you hover the cursor over the rectangle, the tooltip displays the word *Stop* along with the name of whatever class is running. When you click the rectangle, the active Java program stops running and the rectangle turns grey.

From infinity to affinity

For some problems, an infinite loop is normal and desirable. Consider, for example, a real-time, mission-critical application — air traffic control or the monitoring of a heart-lung machine. In these situations, a program should run and run and run.

But a game of Twenty-One should end pretty quickly. In Figure 11-8, the game doesn't end, because the `total` never reaches 21 or higher. In fact, the `total` is always 0. The problem is that my code has no statement to change the `total` variable's value. I should add each card's value to the `total`:

```
total += card;
```

Again, I ask myself where this statement belongs in the code. How many times should the computer execute this assignment statement? Once at the start of the program? Once at the end of the run? Repeatedly?

The computer should repeatedly add a card's value to the running total. In fact, the computer should add to the total each time a card gets drawn. So the preceding assignment statement should be inside the `while` loop, right alongside the statement that gets a new `card` value:

```
card = myRandom.nextInt(10) + 1;
total += card;
```

With this revelation, I'm ready to see the complete program. The code is in Listing 11-2, and two runs of the code are shown in Figure 11-5.

LISTING 11-2: A Simplified Version of the Game Twenty-One

```
import java.util.Random;

public class PlayTwentyOne {

    public static void main(String[] args) {
        var myRandom = new Random();
        int card, total = 0;

        System.out.println("Card    Total");

        while (total < 21) {
            card = myRandom.nextInt(10) + 1;
            total += card;

            System.out.print(card);
            System.out.print("       ");
            System.out.println(total);
        }

        if (total == 21) {
            System.out.println("You win :-)");
        } else {
            System.out.println("You lose :-(");
        }
    }
}
```

If you've read this whole section, you're probably exhausted. Creating a loop can be a lot of work. Fortunately, the more you practice, the easier it becomes.

REAL ALIGNMENT

In Figure 11-5, you see the numbers `8 8`, and then `6 14` (and so on) displayed. I wanted these numbers to be right under the heading words `Card` and `Total`. So, in Listing 11-2, I put blank spaces in `print` and `println` calls. This strategy doesn't work if the program randomly draws a 10 card. The two digits in the number `10` throw the alignment out of whack.

```
Card    Total
6       6
10      16
9       25
```

You may be tempted to replace the blank spaces with tabs. Java's `\t` escape sequence stands for the tab character.

```
System.out.println("Card\tTotal");
...
System.out.print(card);
System.out.print("\t");
System.out.println(total);
```

Unfortunately, the tab idea has problems of its own. In the Windows command prompt and Macintosh Terminal, tab stops are eight spaces apart. So, in those environments, the program's output may look like this:

```
Card    Total
6       6
7       13
8       21
```

But in IntelliJ's Run tool window, tab stops are only four spaces apart, in which case you may see this:

```
Card    Total
6       6
7       13
8       21
```

I never use tabs — not in my code and not in the code's output.* For more reliable alignment of the columns and their headings, I recommend Java's `System.out.printf` method.

```
System.out.println("Card    Total");
System.out.printf("%2d", card);
System.out.print("       ");
System.out.printf("%2d\n", total);
```

You can learn all about the `printf` method by visiting `https://docs.oracle.com/javase/tutorial/java/data/numberformat.html`. On the other hand, this book is a real page-turner. You may not want to put it down to learn about `printf`. For a nice way to display currency amounts, see Chapter 14.

Priming the Pump

I remember when I was a young boy. We lived on Front Street in Philadelphia, near where the El train turned onto Kensington Avenue. Come early morning, I'd have to go outside and get water from the well. I'd pump several times before any water would come out. Ma and Pa called it "priming the pump."

These days, I don't prime pumps. I prime `while` loops. Consider the case of a busy email system administrator. She needs a program that extracts a username from an email address. For example, the program reads

```
John@BurdBrain.com
```

and writes

```
John
```

* A "religious war" among programmers concerning tabs versus spaces rages on to this day. The *Silicon Valley* TV show drew laughs with the issue in Season 3 Episode 6. There's even been a study to determine who earns more money — programmers who use tabs or programmers who use spaces. If you believe in such studies, visit `https://stackoverflow.blog/2017/06/15/developers-use-spaces-make-money-use-tabs/`.

How does the program do it? Like other examples in this chapter, this problem involves repetition:

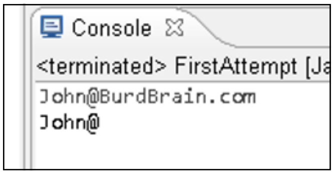
```
Repeatedly do the following:
    Read a character.
    Write the character.
```

The program then stops the repetition when it finds the @ sign. I take a stab at writing this program. My first attempt doesn't work, but it's a darn good start. It's in Listing 11-3.

LISTING 11-3: Trying to Get a Username from an Email Address

```
/*
 * This code does NOT work, but I'm not discouraged.
 */

import java.util.Scanner;

public class FirstAttempt {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        char symbol = ' ';

        while (symbol != '@') {
            symbol = keyboard.findWithinHorizon(".",0).charAt(0);
            System.out.print(symbol);
        }
        System.out.println();

        keyboard.close();
    }
}
```

When you run the code in Listing 11-3, you get the output shown in Figure 11-9. The user types one character after another — the letter `J`, then `o`, then `h`, and so on. At first, the program in Listing 11-3 does nothing. (The computer sends the user's input to the program only after the user presses Enter.) After the user types a whole email address and presses Enter, the program gets its first character (the `J` in `John`).



```
John@BurdBrain.com
John@
```

FIGURE 11-9: Oops! Got the @ sign, too.

Unfortunately, the program's output isn't what you expect. Rather than just the username John, you get the username and the @ sign.

To find out why this happens, follow the computer's actions as it reads the input John@BurdBrain.com:

```
Set symbol to ' ' (a blank space).

Is that blank space the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'J'.
    Print the letter 'J'.

Is that 'J' the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'o'.
    Print the letter 'o'.

Is that 'o' the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'h'.
    Print the letter 'h'.

Is that 'h' the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'n'.
    Print the letter 'n'.

Is that 'n' the same as an @ sign?  //Here's the problem.
No, so perform a loop iteration.
    Input the @ sign.
    Print the @ sign.                //Oops!

Is that @ sign the same as an @ sign?
Yes, so stop iterating.
```

Near the end of the program's run, the computer compares the letter `n` with the `@` sign. Because `n` isn't an @ sign, the computer dives right into the loop:

- The first statement in the loop reads an @ sign from the keyboard.
- The second statement in the loop doesn't check to see whether it's time to stop printing. Instead, that second statement just marches ahead and displays the @ sign.

After you've displayed the @ sign, there's no going back. You can't change your mind and undisplay the @ sign. So the code in Listing 11-3 doesn't quite work.

Working on the problem

You learn from your mistakes. The problem with Listing 11-3 is that, between reading and writing a character, the program doesn't check for an @ sign. Rather than do "Test, Input, Print," it should do "Input, Test, Print." That is, rather than do this:

```
Is that 'o' the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'h'.
    Print the letter 'h'.

Is that 'h' the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'n'.
    Print the letter 'n'.

Is that 'n' the same as an @ sign?  //Here's the problem.
No, so perform a loop iteration.
    Input the @ sign.
    Print the @ sign.               //Oops!
```

the program should do this:

```
    Input the letter 'o'.
Is that 'o' the same as an @ sign?
No, so perform a loop iteration.
    Print the letter 'o'.

    Input the letter 'n'.
Is that 'n' the same as an @ sign?
```

```
No, so perform a loop iteration.
    Print the letter 'n'.

    Input the @ sign.
Is that @ sign the same as an @ sign?
Yes, so stop iterating.
```

This cycle is shown in Figure 11-10.

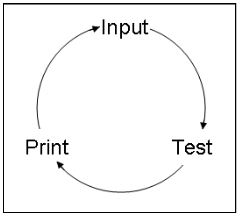


FIGURE 11-10: What the program needs to do.

You can try to imitate the following informal pattern:

```
    Input a character.
Is that character the same as an @ sign?
If not, perform a loop iteration.
    Print the character.
```

The problem is, you can't put a `while` loop's test in the middle of the loop:

```
//This code doesn't work the way you want it to work:
{
   symbol = keyboard.findWithinHorizon(".",0).charAt(0);
while (symbol != '@')
   System.out.print(symbol);
}
```

You can't sandwich a `while` statement's condition between two of the statements that you intend to repeat. What can you do? You need to follow the flow in Figure 11-11. Because every `while` loop starts with a test, that's where you jump into the circle, First Test, and then Print, and then, finally, Input.

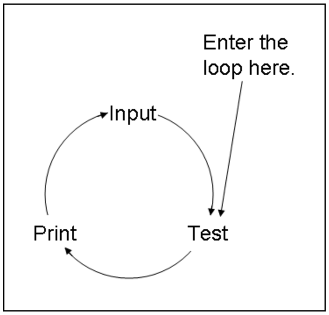


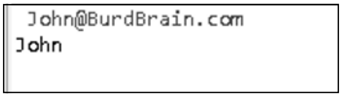
FIGURE 11-11: Jumping into a loop.

Listing 11-4 shows the embodiment of this “test, then print, then input” strategy.

LISTING 11-4: **Nice Try, But . . .**

```
/*
 * This code almost works, but there's one tiny error:
 */

import java.util.Scanner;

public class SecondAttempt {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        char symbol = ' ';

        while (symbol != '@') {
            System.out.print(symbol);
            symbol = keyboard.findWithinHorizon(".",0).charAt(0);
        }

        System.out.println();

        keyboard.close();
    }
}
```

A run of the code in Listing 11-4 is shown in Figure 11-12.



```
 John@BurdBrain.com
John
```

FIGURE 11-12: The computer displays an extra blank space.

The code is almost correct, but I still have a slight problem. Notice the blank space before the user's input. The program races prematurely into the loop. The first time the computer executes the statements

```
System.out.print(symbol);
symbol = keyboard.findWithinHorizon(".",0).charAt(0);
```

the computer displays an unwanted blank space. Then the computer gets the `J` in `John`. In some applications, an extra blank space is no big deal. But in other applications, extra output can be disastrous.

Fixing the problem

Disastrous or not, an unwanted blank space is the symptom of a logical flaw. The program shouldn't display results before it has any meaningful results to display. The solution to this problem is called — drumroll, please — *priming the loop.* You pump `findWithinHorizon(".",0).charAt(0)` once to get some values flowing. Listing 11-5 shows you how to do it.

LISTING 11-5: How to Prime a Loop

```
/*
 * This code works correctly!
 */

import java.util.Scanner;

public class GetUserName {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        char symbol;

        symbol = keyboard.findWithinHorizon(".", 0).charAt(0);

        while (symbol != '@') {
            System.out.print(symbol);
            symbol = keyboard.findWithinHorizon(".", 0).charAt(0);
        }

        System.out.println();

        keyboard.close();
    }
}
```

Listing 11-5 follows the strategy shown in Figure 11-13. First you get a character (the letter `J` in `John`, for example), and then you enter the loop. After you're in the loop, you test the letter against the @ sign and print the letter if it's appropriate to do so. Figure 11-14 shows a beautiful run of the `GetUserName` program.

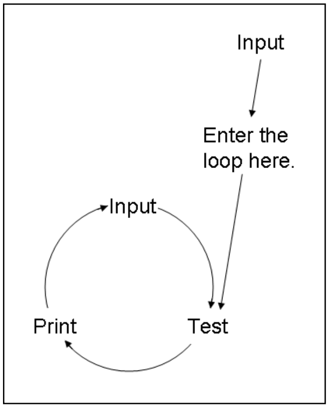
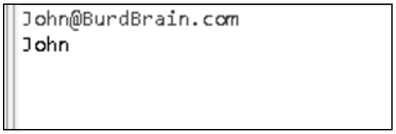


FIGURE 11-13: The strategy in Listing 11-5.



```
John@BurdBrain.com
John
```

FIGURE 11-14: A run of the code in Listing 11-5.

The priming of loops is an important programming technique. But it's not the end of the story. In Chapters 12, 18, and 19, you can read about some other useful looping tricks.

TRY IT OUT

Face Java's loops head-on with the following programming challenges.

LIVING LARGE

Write a program that repeatedly reads numbers from the user's keyboard. The program stops looping when the user types a number that's larger than 100.

ARE WE THERE YET?

Write a program that repeatedly prompts the user with the same question: `Are we there yet?` The program stops looping when the user replies with the uppercase letter `Y` or the lowercase letter `y`. Here's a sample run:

```
Are we there yet? N
Are we there yet? n
Are we there yet? N
Are we there yet? N
Are we there yet? N
Are we there yet? y
Whew!
```

TALLY UP

Write a program that uses a loop to repeatedly read numbers from the keyboard. The program stops reading numbers when the user enters a negative number. The program reports the sum of the numbers, excluding the last (negative) number.

GUESS AGAIN

In Chapter 9, you write a guessing-game program. The program compares whatever number the user enters with a number that the program generates randomly. The user wins if the two numbers are equal, and loses otherwise.

Modify this program as follows:

- As in the original version, the program generates a number randomly only once, but . . .
- . . . the program repeatedly asks the user for guesses until the user guesses correctly.

TWO IN A ROW

Write a program that repeatedly reads numbers from the user's keyboard. The program stops looping when the user types the same number twice in a row. Here's a sample run of the program:

```
5
13
21
5
4
5
5
Done!
```

IN THIS CHAPTER

- » Creating repetitive actions
- » Insisting on a valid response from the user
- » Deleting incriminating evidence

Chapter 12

Circling Back to Java Loops

I remember it distinctly — the sense of dread I would feel on the way to Aunt Edna's house. She was a kind old woman, and her intentions were good. But visits to her house were always agonizing.

First, we'd sit in the living room and talk about our other relatives. That was okay, as long as I understood what people were talking about. Sometimes, the gossip would be about adult topics and I'd become bored.

After all the family chatter ended, my father would help Aunt Edna with paying her bills. That was fun to watch because she had a genetically inherited family ailment: Like me and many of my ancestors, Aunt Edna couldn't keep track of paperwork to save her life. It was as if the paper had allergens that made Aunt Edna's skin crawl. After ten minutes of useful bill paying, my father would find a mistake, an improper tally, or something else in the ledger that needed attention. He'd ask Aunt Edna about it, and she'd shrug her shoulders. He'd become agitated trying to track down the problem while Aunt Edna rolled her eyes and smiled with ignorant satisfaction. It was great entertainment.

Then, when the bill paying was done, we'd sit down to eat dinner. That's when I would remember why I dreaded these visits: Dinner was unbearable. Aunt Edna believed in Fletcherism — a health-oriented movement whose followers chewed each mouthful of food 100 times. The more devoted followers used a chart, with a different number for the mastication of each kind of food. The minimal number of chews for any food was 32 — one chomp for each tooth in your mouth. People who did this said they were "Fletcherizing."

Mom and Dad thought the whole Fletcher business was silly, but they respected Aunt Edna and felt that people her age should be humored, not defied. As for me, I thought I'd explode from the monotony. Each meal lasted forever. Each mouthful was an ordeal. I can still remember my mantra — the words I'd say to myself without meaning to do so:

```
I've chewed 0 times so far.
Have I chewed 100 times yet? If not, then
    Chew!
    Add 1 to the number of times that I've chewed.
    Go back to "Have I chewed" to find out if I'm done yet.
```

Repeating Statements a Certain Number of Times (Java for Statements)

Life is filled with examples of counting loops. And computer programming mirrors life. (Or is it the other way around?) When you tell a computer what to do, you're often telling the computer to print three lines, process ten accounts, dial a million phone numbers, or whatever. Because counting loops are common in programming, the people who create programming languages have developed statements just for loops of this kind. In Java, the statement that repeats something a certain number of times is called a *for* statement. An example of a `for` statement is in Listing 12-1.

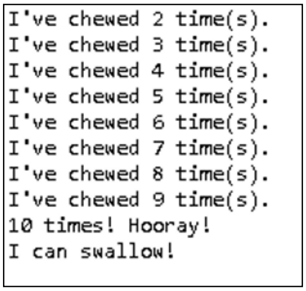
LISTING 12-1: Horace Fletcher's Revenge

```
import static java.lang.System.out;

public class AuntEdnaSettlesForTen {

    public static void main(String[] args) {

        for (int count = 0; count < 10; count++) {
            out.print("I've chewed ");
            out.print(count);
            out.println(" time(s).");
        }
```

```
         out.println("10 times! Hooray!");
         out.println("I can swallow!");
   }
}
```

Figure 12-1 shows you what you get when you run the program in Listing 12-1:

- The `for` statement in Listing 12-1 starts by setting the `count` variable equal to 0.
- Then the `for` statement tests to make sure that `count` is less than 10 (which it certainly is).
- Then the `for` statement dives ahead and executes the printing statements between the curly braces. At this early stage of the game, the computer prints `I've chewed 0 time(s).`
- Then the `for` statement executes `count++` — that last thing inside the `for` statement's parentheses. This last action adds 1 to the value of `count`.



```
I've chewed 2 time(s).
I've chewed 3 time(s).
I've chewed 4 time(s).
I've chewed 5 time(s).
I've chewed 6 time(s).
I've chewed 7 time(s).
I've chewed 8 time(s).
I've chewed 9 time(s).
10 times! Hooray!
I can swallow!
```

FIGURE 12-1: Chewing ten times.

This ends the first iteration of the `for` statement in Listing 12-1. Of course, this loop has more to it than just one iteration:

- With `count` now equal to 1, the `for` statement checks again to make sure that `count` is less than 10. (Yes, 1 is smaller than 10.)
- Because the test turns out okay, the `for` statement marches back into the curly-braced statements and prints `I've chewed 1 time(s)` on the screen.
- Then the `for` statement executes that last `count++` inside its parentheses. The statement adds 1 to the value of `count`, increasing the value of `count` to 2.

And so on. This whole process repeats over and over again until, after ten iterations, the value of `count` finally reaches 10. When this happens, the check for count being less than 10 fails, and the loop's execution ends. The computer jumps to whatever statement comes immediately after the for statement. In Listing 12-1, the computer prints 10 times! Hooray! I can swallow! The whole process is illustrated in Figure 12-2.

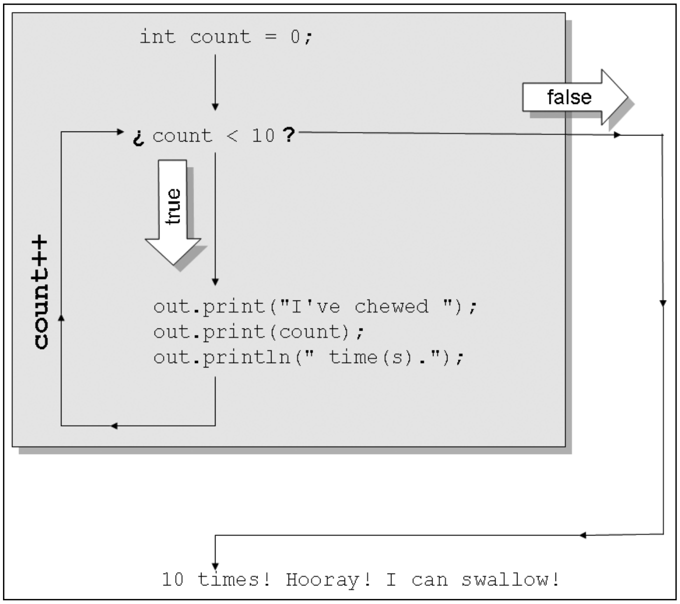

int count = 0;
false
¿ count < 10 ?
true
count++
out.print("I've chewed ");
out.print(count);
out.println(" time(s).");
10 times! Hooray! I can swallow!

FIGURE 12-2: The action of the for loop in Listing 12-1.

Esprit de for

A typical for statement looks like this:

```
for (Initialization; Condition; Update) {
    Statements
}
```

After the word for, you put three things in parentheses: an initialization, a condition, and an update.

Each of the three items in parentheses plays its own, distinct role:

- **Initialization:** The initialization is executed once, when the run of your program first reaches the for statement.
- **Condition:** The condition is tested several times (at the start of each iteration).
- **Update:** The update is also evaluated several times (at the end of each iteration).

If it helps, think of the loop as though its text is shifted all around:

```
//This is NOT real code
int count = 0
for count < 0 {
   out.print("I've chewed ");
   out.print(count);
   out.println(" time(s).");
   count++;
}
```

You can't write a real for statement this way. (The compiler would throw code like this right into the garbage can.) Even so, this is the order in which the parts of the for statement are executed.

WARNING

The first line of a for statement (the word for followed by stuff in parentheses) isn't a complete statement. So you almost never put a semicolon after the stuff in parentheses. If you make a mistake and type a semicolon, like this:

```
// DON'T DO THIS:
for (int count = 0; count < 10; count++); {
```

you usually put the computer into a do-nothing loop. The computer counts to itself from 0 to 9. After counting, the computer executes whatever statements come immediately after the open curly brace. (The loop ends at the semicolon, so the statements after the open curly brace aren't inside the loop.)

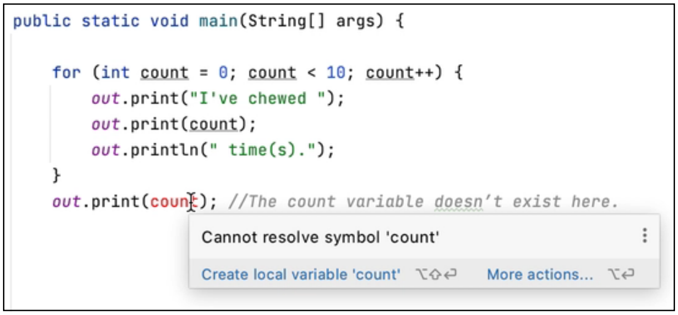
Initializing a for loop

Look at the first line of the for loop in Listing 12-1 and notice the declaration int count = 0. That's something new. When you create a for loop, you can declare a variable (like count) as part of the loop initialization.

If you declare a variable in the initialization of a for loop, you can't use that variable outside the loop. For example, in Listing 12-1, try putting out.println(count) after the end of the loop:

```
//This code does not compile.
for (int count = 0; count < 10; count++) {
     out.print("I've chewed ");
     out.print(count);
     out.println(" time(s).");
}

out.print(count); //The count variable doesn't exist here.
```

With this extra reference to the count variable, the compiler gives you an error message. You can see the message in Figure 12-3. If you're not experienced with for statements, the message may surprise you — "Whadaya mean Cannot resolve symbol 'count'? You can see a count variable declaration just four lines above that statement." Ah, yes. But the count variable is declared in the for loop's initialization. Outside the for loop, that count variable doesn't exist.



```
public static void main(String[] args) {

    for (int count = 0; count < 10; count++) {
        out.print("I've chewed ");
        out.print(count);
        out.println(" time(s).");
    }
    out.print(count); //The count variable doesn't exist here.
```

Cannot resolve symbol 'count'

FIGURE 12-3: What count variable? I don't see a count variable.

To use a variable outside of a for statement, you have to declare that variable outside the for statement. You can even do this with the for statement's counting variable. Listing 12-2 has an example.

LISTING 12-2: Using a Variable Declared Outside of a for Loop

```
import static java.lang.System.out;

public class AuntEdnaDoesItAgain {

    public static void main(String[] args) {
        int count;

        for (count = 0; count < 10; count++) {
            out.print("I've chewed ");
            out.print(count);
            out.println(" time(s).");
        }

        out.print(count);
        out.println(" times! Hooray!");
        out.println("I can swallow!");
    }
}
```

A run of the code in Listing 12-2 looks exactly like the run for Listing 12-1. The run is pictured in Figure 12-1. Unlike its predecessor, Listing 12-2 enjoys the luxury of using the `count` variable to display the number 10. It can do this because in Listing 12-2, the `count` variable belongs to the entire `main` method and not to the `for` loop alone.

Notice the words `for (count = 0` in Listing 12-2. Because `count` is declared before the `for` statement, you don't declare `count` again in the `for` statement's initialization. I tried declaring `count` twice, as in the following code:

```
//This does NOT work:
int count;

for (int count = 0; count < 10; count++) {
     ... etc.
```

And IntelliJ told me to clean up my act:

```
Variable 'count' is already defined in the scope
```

VERSATILE LOOPING STATEMENTS

If you were stuck on a desert island with only one kind of loop, what kind would you want to have? The answer is, you can get along with any kind of loop. The choice between a `while` loop and a `for` loop is about the code's style and efficiency. It's not about necessity.

Anything you can do with a `for` loop, you can do with a `while` loop as well. Consider, for example, the `for` loop in Listing 12-1. Here's how you can achieve the same effect with a `while` loop:

```
int count = 0;
while (count < 10) {
  out.print("I've chewed ");
  out.print(count);
  out.println(" time(s).");
  count++;
}
```

In the `while` loop, you have explicit statements to declare, initialize, and increment the `count` variable.

(continued)

(continued)

The same kind of trick works in reverse. Anything you can do with a `while` loop, you can do with a `for` loop as well. But turning certain `while` loops into `for` loops seems strained and unnatural. Consider a `while` loop from Listing 11-2, in Chapter 11:

```
while (total < 21) {
    card = myRandom.nextInt(10) + 1;
    total += card;
    System.out.print(card);
    System.out.print(" ");
    System.out.println(total);
}
```

Turning this loop into a `for` loop means wasting most of the stuff inside the `for` loop's parentheses:

```
for ( ; total < 21 ; ) {
    card = myRandom.nextInt(10) + 1;
    total += card;
    System.out.print(card);
    System.out.print(" ");
    System.out.println(total);
}
```

The preceding `for` loop has a condition, but it has no initialization and no update. That's okay. Without an initialization, nothing special happens when the computer first enters the `for` loop. And without an update, nothing special happens at the end of each iteration. It's strange, but it works.

Usually, when you write a `for` statement, you're counting how many times to repeat something. But, in truth, you can do just about any kind of repetition with a `for` statement.

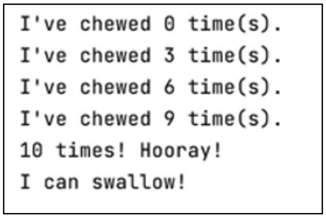
Shut up and chew!

Look again at Figure 12-1. Do you really want the code to report each and every time through the loop? Maybe not. Listing 12-3 honks its horn only after three chews in a row.

LISTING 12-3: Enjoying a Quieter Meal

```
import static java.lang.System.out;

public class LessTalking {

    public static void main(String[] args) {

        for (int count = 0; count < 10; count++) {
            if (count % 3 == 0) {
                out.print("I've chewed ");
                out.print(count);
                out.println(" time(s).");
            }
        }
        out.println("10 times! Hooray!");
        out.println("I can swallow!");
    }
}
```

Listing 12-3 contains an `if` statement inside of a loop. The effect is to print messages during certain loop iterations and to skip the printing during other iterations. Figure 12-4 shows the output of the code in Listing 12-3.



```
I've chewed 0 time(s).
I've chewed 3 time(s).
I've chewed 6 time(s).
I've chewed 9 time(s).
10 times! Hooray!
I can swallow!
```

FIGURE 12-4: A modest amount of information about chewing.

Chapter 7 tells you about Java's remainder operator (`%`). Here's how it works:

- **When `count` is 4, the `if` statement in Listing 12-3 finds `4 % 3` — the remainder when you divide 4 by 3.**

  That remainder is 1, not 0. So Java doesn't execute the code's `out.print` calls.

- **When `count` is 5, the `if` statement finds `5 % 3` — the remainder when you divide 5 by 3.**

  That remainder is 2, not 0. So Java doesn't execute the code's `out.print` calls.

- **When count is 6, the if statement finds 6 % 3 — the remainder when you divide 6 by 3.**

  That remainder is 0. So Java executes the code's out.print calls. The same reasoning works for all the numbers from 0 to 9.

TRY IT OUT

When it comes to writing code with loops, there's no such thing as having too much practice. Try these problems. Work slowly and don't get discouraged. Remember that solutions are available at http://beginprog.allmycode.com.

NARCISSIST'S CODE

Write a program that reads the user's name and a number (howMany) from the keyboard. The program uses a for loop to display the user's name howMany times on the screen.

BRITISH POUNDS TO US DOLLARS

In April 2017, one British pound was worth 1.25 US dollars. Write a program to create a simple currency conversion table. In your program, use a for loop to display the following table:

```
Pounds Dollars
1        1.25
2        2.5
3        3.75
4        5.0
5        6.25
6        7.5
7        8.75
8       10.0
9       11.25
```

Repeating Until You Get What You Need (Java do Statements)

I introduce Java's while loop in Chapter 11. When you create a while loop, you write the loop's condition first. After the condition, you write the code that gets repeatedly executed.

```
while (Condition) {
    Code that gets repeatedly executed
}
```

This way of writing a `while` statement is no accident. The look of the statement emphasizes an important point — that the computer always checks the condition before executing any of the repeated code.

If the loop's condition is never true, the stuff inside the loop is never executed — not even once. In fact, you can easily cook up a `while` loop whose statements are never executed (although I can't think of a reason why you would ever want to do it):

```
//This code doesn't print anything:
int twoPlusTwo = 2 + 2;
while (twoPlusTwo == 5) {
    System.out.println("""
            Are you kidding? 2+2 doesn't equal 5.
            Everyone knows that 2+2 equals 3.""");
}
```

In spite of this silly `twoPlusTwo` example, the `while` statement turns out to be the most useful of Java's looping constructs. In particular, the `while` loop is good for situations in which you must look before you leap. For example: "While money is in my account, write a mortgage check every month." When you first encounter this statement, if your account has a zero balance, you don't want to write a mortgage check — not even one check.

But at times (not many), you want to leap before you look. In a situation when you're asking the user for a response, maybe the user's response makes sense, but maybe it doesn't. Maybe the user's finger slipped, or perhaps the user didn't understand the question. In many situations, it's important to correctly interpret the user's response. If the user's response doesn't make sense, you must ask again.

Holding out for a trustworthy response

Consider a program that deletes a file. Before deleting the file, the program asks for confirmation from the user. If the user types **Y**, delete; if the user types **N**, don't delete. Of course, deleting a file is serious stuff. Mistaking a bad keystroke for a "yes" answer can delete the company's records. (And mistaking a bad keystroke for a "no" answer can preserve the company's incriminating evidence.) If there's any doubt about the user's response, the program should ask the user to respond again.

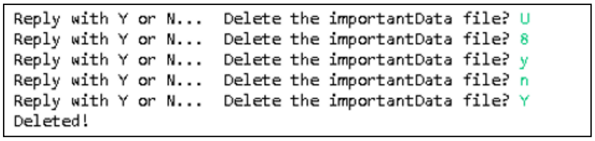
Pause a moment to think about the flow of actions — what should and shouldn't happen when the computer executes the loop. A loop of this kind doesn't need to check anything before getting the user's first response. Indeed, before the user gives the first response, the loop has nothing to check. The loop shouldn't start with, "as long as the user's response is invalid, get another response from the user." Instead, the loop should just leap ahead, get a response from the user, and then check the response to see whether it made sense. The code to do all of this is in Listing 12-4.

LISTING 12-4: Repeat Before You Delete

```
/*
 * DISCLAIMER: Neither the author nor John Wiley & Sons,
 * Inc., nor anyone else even remotely connected with the
 * creation of this book, assumes any responsibility
 * for any damage of any kind due to the use of this code,
 * or the use of any work derived from this code,
 * including any work created partially or in full by
 * the reader.
 *
 * Sign here:_______________________________
 */

import java.io.File;
import java.util.Scanner;

public class IHopeYouKnowWhatYoureDoing {

    public static void main(String[] args) {

        var keyboard = new Scanner(System.in);
        char reply;

        do {

            System.out.print("Reply with Y or N...");
            System.out.print(" Delete the importantData file? ");
            reply = keyboard.findWithinHorizon(".", 0).charAt(0);

        } while (reply != 'Y' && reply != 'N');

        if (reply == 'Y') {
            new File("importantData.txt").delete();
            System.out.println("Deleted!");
        } else {
            System.out.println("No harm in asking!");
        }

        keyboard.close();
    }
}
```

Deleting a file

A run of the program in Listing 12-4 is shown in Figure 12-5. Before deleting a file, the program asks the user whether it's okay to make the deletion. If the user gives one of the two expected answers (Y or N), the program proceeds according to the user's direction. But if the user enters any other letter (or any digit, punctuation symbol, or whatever), the program asks the user for another response.



```
Reply with Y or N...  Delete the importantData file? U
Reply with Y or N...  Delete the importantData file? 8
Reply with Y or N...  Delete the importantData file? y
Reply with Y or N...  Delete the importantData file? n
Reply with Y or N...  Delete the importantData file? Y
Deleted!
```

FIGURE 12-5: No! Don't do it!

In Figure 12-4, the user hems and haws for a while, first with the letter U, and then with the digit 8, and then with lowercase letters. Finally, the user enters Y and the program deletes the importantData.txt file. If you compare the files on your hard drive (before and after the run of the program), you see that the program trashes the file named importantData.txt.

If you use IntelliJ IDEA, here's how you can tell that a file is being deleted:

1. **Create a Java project containing the code in Listing 12-4.**
2. **In the Project tool window, right-click the tree's topmost branch. (See Figure 12-6.)**

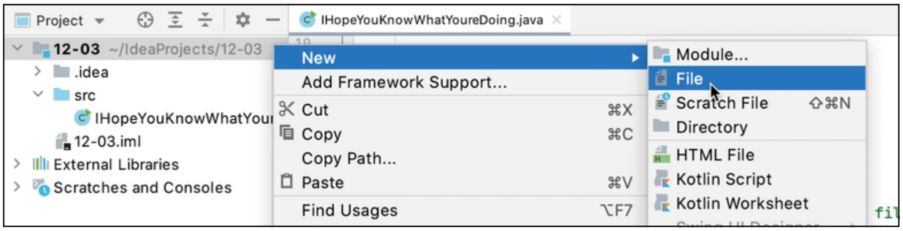


FIGURE 12-6: Adding a file to your project.

Figure 12-6 shows part of IntelliJ's main window. This window describes a project named 12–03. Notice the label 12–03 at the top of the Project tool window's tree. The 12–03 project's files live primarily in a folder named 12–03 on the computer's hard drive. That 12–03 folder is called the project's *root folder*. The root folder has subfolders named .idea and src along with some other useful goodies.

To the right of the `12-03` label, IntelliJ faintly displays the characters `~/IdeaProjects/12-03`. So, on the computer's hard drive, the project's `12-03` root folder is a subfolder of a folder named `IdeaProjects`.

REMEMBER

Don't right-click any of the project's subfolders. (For example, don't right-click the project's `src` folder.) Instead, right-click the project's root.

TECHNICAL STUFF

The tree in IntelliJ's Project tool window is a useful summary of a project's resources, but it isn't quite the same as the tree in your computer's File Explorer or Finder. Some items on the Project tool window's tree are neither files nor folders, and some items on your hard drive don't appear on IntelliJ's tree.

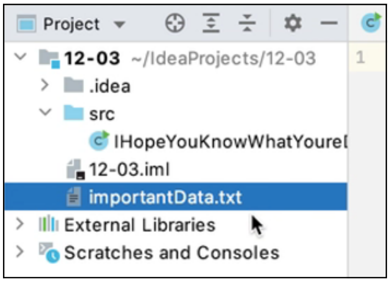
3. **On the resulting context menu, choose New ➪ File.**

   IntelliJ's New File dialog box appears.

4. **In the dialog box's text field, type the name of your new file.**

   Type **importantData.txt**.

5. **With the cursor still in the dialog box's text field, press Enter.**
6. **Observe that the file's name appears in IntelliJ's Project tool window. (See Figure 12-7.)**



```
12-03 ~/IdeaProjects/12-03
  .idea
  src
    IHopeYouKnowWhatYoure[
  12-03.iml
  importantData.txt
External Libraries
Scratches and Consoles
```

FIGURE 12-7: Your new file hangs on a branch of the Project tool window's tree.

The name is in the project's root directory. If you're unsure about this, collapse the tree's `src` branch. When you do, the project's `IHopeYouKnowWhatYoureDoing` branch disappears, but the `importantData.txt` branch doesn't. (See Figure 12-8.)

You put your new file in the root directory because, in Listing 12-4, the name `importantData.txt` (with no slashes or backslashes) refers only to a name in the project's root directory. The program's run has no effect on any files outside of the root directory, even if any of those files has the name `importantData.txt`.

CROSS REFERENCE

To find out how to refer to files outside of the project's root directory, see Chapter 16.

For this experiment, you don't have to add any text to the file. The file exists only to be deleted. (What a shame!)

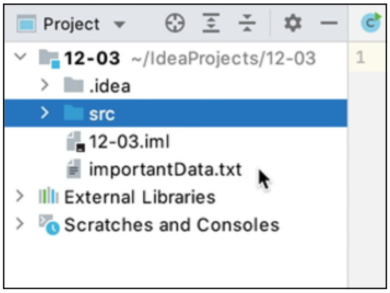


FIGURE 12-8: The `important Data.txt` file isn't inside the `src` branch.

7. **Run the program.**

   When the program runs, type **Y** to delete the `importantData.txt` file.

8. **After running the program, look for `importantData.txt` in the Project tool window.**

   Sure enough, you can't find an `importantData.txt` file. You've deleted it!

In Listing 12-4, the statement

```
new File("importantData.txt").delete();
```

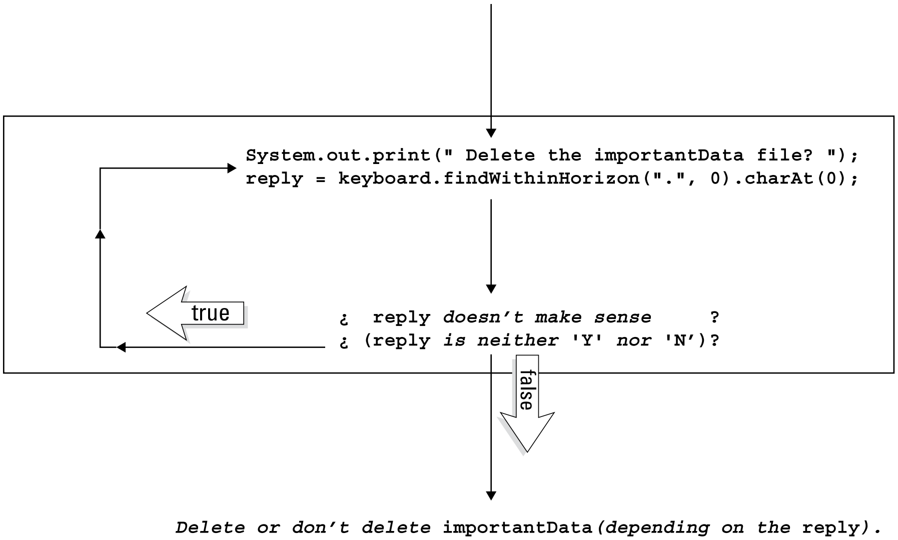
is tricky. At first glance, you seem to be creating a new file, only to delete that file in the same line of code! But in reality, the words `new File` create only a representation of a file inside your program. To be more precise, the words `new File` create, inside your program, a representation of a disk file that may or may not already exist on your computer's hard drive. Here's what the `new File` statement really means:

> "Let `new File("importantData.txt")` refer to a file named `importantData.txt`. If such a file exists, then delete it."

Yes, the devil is in the details. But smiles are in the subtleties, and nobility is in the nuance.

Taming of the do

To write the program in Listing 12-4, you need a loop — a loop that repeatedly asks the user whether the `importantData.txt` file should be deleted. (The action of the loop in Listing 12-4 is illustrated in Figure 12-9.) The loop continues to ask until the user gives a meaningful response. The loop tests its condition at the end of each iteration, after each of the user's responses.



```
System.out.print(" Delete the importantData file? ");
reply = keyboard.findWithinHorizon(".", 0).charAt(0);
```

true
¿ reply doesn't make sense ?
¿ (reply is neither 'Y' nor 'N')?
false
Delete or don't delete importantData (depending on the reply).

FIGURE 12-9: Here we go loop, do loop.

That's why the program in Listing 12-4 has a *do* loop (also known as a `do ... while` loop). With a `do` loop, the program jumps right in, executes some statements, and then checks a condition. If the condition is true, the program goes back to the top of the loop for another go-around. If the condition is false, the computer leaves the loop (and jumps to whatever code comes immediately after the loop).

Repeat performance

The format of a `do` loop is

```
do {
    Statements
} while (Condition)
```

Writing the *`Condition`* at the end of the loop reminds me that the computer executes the *`Statements`* inside the loop first. After the computer executes the *`Statements,`* it goes on to check the *`Condition`*. If the *`Condition`* is true, the computer goes back for another iteration of the *`Statements`*.

With a `do` loop, the computer always executes the statements inside the loop at least once:

```
//This code prints something:
int twoPlusTwo = 2 + 2;
do {
    System.out.println("""
            Are you kidding? 2+2 doesn't equal 5.
            Everyone knows that 2+2 equals 3.""");
} while (twoPlusTwo == 5);
```

This code displays `Are you kidding? 2+2 doesn't equal 5 . . .` *and so on* and then tests the condition `twoPlusTwo == 5`. Because `twoPlusTwo == 5` is false, the computer doesn't go back for another iteration. Instead, the computer jumps to whatever code comes immediately after the loop.

TRY IT OUT

Get some practice using Java's `do` statement.

DO I HEAR AN ECHO?

In a `do` statement, repeatedly read numbers from the keyboard. Display each number back to the user on the screen. After displaying a number, ask whether the user wants to continue entering numbers. When the user replies with the letter `n`, stop.

Here's a sample run of the program:

```
Enter a number: 5
5
Continue? (y/n) y

Enter a number: 81
81
Continue? (y/n) y

Enter a number: 29
29
Continue? (y/n) n

Done!
```

TALLYHO!

In a `do` statement, repeatedly read `int` values from the keyboard and keep track of the running total. The user says, "I want to stop entering values" by typing one final `int` value — the value 0. At that point, the program displays the total of all values that the user entered.

4
The Inside ScOOP

IN THIS PART . . .

Understanding object-oriented programming (and not just pretending to understand)

Dividing your code into manageable chunks

Creating recipes to use over and over again

IN THIS CHAPTER

- » Programming with class (and with style and finesse)
- » Making objects from classes
- » Joining the exclusive "I understand classes and objects" society

Chapter 13

Programming with Objects and Classes

Chapters 7 and 8 introduce Java's primitive types — things like `int`, `double`, `char`, and `boolean`. That's great, but how often does a real-world problem deal exclusively with such simple values? Consider an exchange between a merchant and a customer. The customer makes a purchase, which can involve item names, model numbers, credit card info, sales tax rates, and lots of other stuff. A purchase is more complicated than an `int` value. It's more complicated than a `double` value. How do you represent an entire purchase in a Java program?

In older computer programming languages, you treat an entire purchase like a big pile of unbundled laundry. Imagine a mound of socks, shirts, and other items of clothing. You have no basket, so you grab as much as you can handle. As you walk to the washer, you drop a few things — a sock here and a washcloth there. This is like the older way of storing the values in a purchase. In older languages, there's no purchase. There are only `double` values, `char` values, and other loose items. You put the purchase amount in one variable, the customer's name in another, and the sales tax data somewhere else. But that's awful. You tend to drop things on your way to the compiler. With small errors in a program, you can easily drop an amount here and a customer's name there.

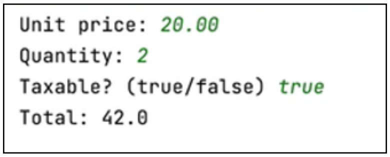
With laundry and computer programming, you're better off if you have a basket. The newer programming languages, like Java, allow you to combine values and make new, more useful kinds of values. For example, in Java, you can combine a `double` value, an `int` value, a `boolean` value, and maybe other kinds of values to create something that you call a `Purchase`. Because your purchase info is all in one big bundle, keeping track of the purchase's pieces is easier. That's the start of an important computer programming concept: the notion of *object-oriented programming.*

The Class Is Always Cleaner

I start with a "traditional" example. The program in Listing 13-1 processes simple purchase data. A run of the program is shown in Figure 13-1.

LISTING 13-1: Doing It the Old-Fashioned Way

```
import java.util.Scanner;

public class ProcessData {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        double unitPrice;
        int quantity;
        boolean taxable;

        System.out.print("Unit price: ");
        unitPrice = keyboard.nextDouble();
        System.out.print("Quantity: ");
        quantity = keyboard.nextInt();
        System.out.print("Taxable? (true/false) ");
        taxable = keyboard.nextBoolean();

        double total = unitPrice * quantity;
        if (taxable) {
            total = total * 1.05;
        }

        System.out.print("Total: ");
        System.out.println(total);

        keyboard.close();
    }
}
```



```
Unit price: 20.00
Quantity: 2
Taxable? (true/false) true
Total: 42.0
```

FIGURE 13-1: Processing a customer's purchase.

If the output in Figure 13-1 looks funny, it's because I do nothing in the code to control the number of digits beyond the decimal point. So in the output, the value $42.00 looks like `42.0`. That's okay. I show you how to fix the problem in Chapter 14.

Reference types and Java classes

The code in Listing 13-1 involves a few simple values: `unitPrice`, `quantity`, and `taxable`. So here's the main point of this chapter: By combining several simple values, you can get a single, more useful value. That's the way it works — you take some of Java's primitive types, whip them together to make a primitive type stew, and what do you get? You get a more useful type called a *reference type.* Listing 13-2 has an example.

LISTING 13-2: **What It Means to Be a Purchase**

```
public class Purchase {
    double unitPrice;
    int quantity;
    boolean taxable;
}
```

The code in Listing 13-2 has no `main` method, so you can't tell IntelliJ to start running that code. If you right-click the editor's `Purchase` tab or the Project tool window's `Purchase` branch, the resulting context menu has no Run item. If you click the little green Run button at the top of IntelliJ's main window, IntelliJ may start running some other listing's code, but it won't start running the code in Listing 13-2. Having a method named `main` is the only way to tell Java to "start running code here."

REMEMBER

The code in Listing 13-2 is quite useful, but Listing 13-2 isn't a complete, runnable application. Don't ask IntelliJ to start running the code in Listing 13-2. IntelliJ can't do that.

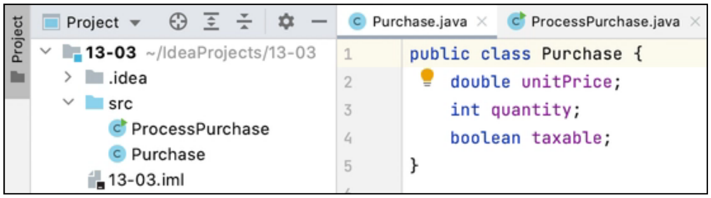
How to use a newly defined class

To do something useful with the code in Listing 13-2, you need a `main` method. That `main` method belongs in a separate file. Listing 13-3 shows you such a file.

LISTING 13-3: Using Your Purchase Class

```
import java.util.Scanner;

public class ProcessPurchase {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        Purchase onePurchase = new Purchase();

        System.out.print("Unit price: ");
        onePurchase.unitPrice = keyboard.nextDouble();
        System.out.print("Quantity: ");
        onePurchase.quantity = keyboard.nextInt();
        System.out.print("Taxable? (true/false) ");
        onePurchase.taxable = keyboard.nextBoolean();

        double total = onePurchase.unitPrice * onePurchase.quantity;
        if (onePurchase.taxable) {
            total = total * 1.05;
        }

        System.out.print("Total: ");
        System.out.println(total);

        keyboard.close();
    }
}
```

To compile and run this section's example, you *must* have two files in your IntelliJ project's `src` folder. One file, named `ProcessPurchase.java`, contains the code in Listing 13-3. The other file, named `Purchase.java` contains the code in Listing 13-2. Figure 13-2 drives this point home.



```
public class Purchase {
    double unitPrice;
    int quantity;
    boolean taxable;
}
```

FIGURE 13-2: Two for the price of two.

With both `ProcessPurchase.java` and `Purchase.java` in your project's `src` folder, right-click either the `ProcessPurchase` branch in the Project tool window's tree or the `ProcessPurchase.java` tab at the top of the Editor. In either case, IntelliJ displays a context menu. On the context menu, select `Run ProcessPurchase` or `Run ProcessPurchase.main()`.

REMEMBER

The code to run is the `ProcessPurchase` code. Right-clicking the `Purchase` branch or the `Purchase.java` tab won't do any harm, but it won't help you run this section's example.

WARNING

Java looks carefully at the names of the files on your computer's hard drive. If you mistakenly put `public class ProcessPurchase` in a file named `Purchase.java` or `FileWithMainMethod.java`, IntelliJ tells you to rethink your choices.

What's going on here?

The best way to understand the code in Listing 13-3 is to compare it, line by line, with the code in Listing 13-1. In fact, there's a mechanical formula for turning the code in Listing 13-1 into the code in Listing 13-3. Table 13-1 describes the formula.

TABLE 13-1: Converting Your Code to Use a Class

In Listing 13-1	In Listing 13-3

`double unitPrice;`
`int quantity;`
`boolean taxable;`
	`Purchase onePurchase = new Purchase();`

`unitPrice`	`onePurchase.unitPrice`
`quantity`	`onePurchase.quantity`
`taxable`	`onePurchase.taxable`

The two programs (in Listings 13-1 and 13-3) do essentially the same thing, but one uses primitive variables, and the other leans on the `Purchase` code from Listing 13-2. Both programs have runs like the one shown earlier, in Figure 13-1.

Why bother?

On the surface, the code in Listing 13-3 is longer, more complicated, and harder to read. But think about a big pile of laundry: It may take time to find a basket and to shovel socks into the basket, but when you have clothes in the basket, the clothes are much easier to carry. It's the same way with the code in Listing 13-3. When you have your data in a `Purchase` basket, it's much easier to do complicated things with purchases.

AN APOLOGY OF SORTS (ALONG WITH AN EXCUSE!)

As far as most professional programmers are concerned, the code in Listings 13-2 and 13-3 is quite simple. In fact, it's too simple. Java programmers don't like expressions such as `onePurchase.unitPrice`, and they hate declarations such as `double unitPrice`. Instead, they prefer `private double unitPrice` along with expressions such as `onePurchase.getUnitPrice()`. These programmers will tell you that "the code in Burd's examples isn't safe," "Burd's examples promote bad programming practices," and "In the summertime, Burd wears colors that don't match." In fact, these programmers are absolutely correct.

But in my opinion, you learn to walk before you learn to run. Classes and objects are difficult concepts, so I present them in as simple a form as I possibly can. Object-oriented programming is an art as well as a science. It's a way of thinking — a way whose mastery requires time and patience. I have students at the university (some outstanding students) who spend months wrestling with the fundamentals of classes and objects.

In this book, I don't present the most time-honored, expert-approved way of dealing with classes and objects. Instead, I follow the simplest path to help you get classes and objects into your consciousness. You can learn the better way of doing things after you've become comfortable dividing problems into their object-oriented parts.

And besides, for all the nasty things someone might say about this chapter's simple examples, the examples aren't fake. They run, they produce output, and they illustrate the separating of a class from the main program's code. At the core, that's what object-oriented programming is all about.

So there!

From Classes Come Objects

The code in Listing 13-2 defines a class. A *class* is a design plan; it describes the way in which you intend to combine and use pieces of data. For example, the code in Listing 13-2 announces your intention to combine `double`, `int`, and `boolean` values to make new `Purchase` values.

Classes are central to all Java programming. But Java is called an *object-oriented* language. Java isn't called a class-oriented language. In fact, no one uses the term *class-oriented language.* Why not?

Well, you can't put your arms around a class. A class isn't real. A class without an object is like a day without chocolate. If you're sitting in a room right now, glance at all the chairs in the room. How many chairs are in the room? Two? Five? Twenty? In a room with five chairs, you have five chair objects. Each chair (each object) is something real, something you can use, something you can sit on.

A language like Java has classes and objects. What's the difference between a class and an object?

- An object is a thing.
- A class is a design plan for things of that kind.

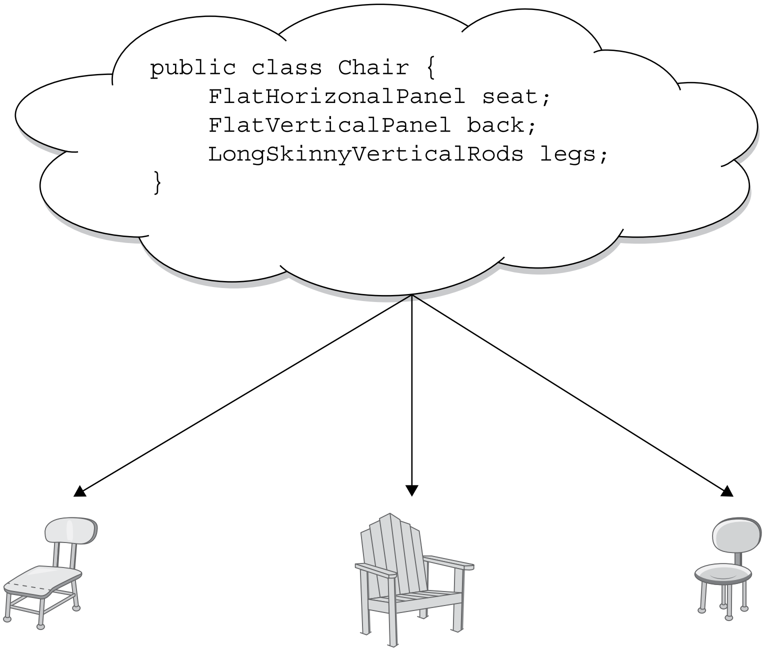
For example, how would you describe what a chair is? Well, a chair has a seat, a back, and legs. In Java, you may write the stuff in Listing 13-4.

LISTING 13-4: What It Means to Be a Chair

```
/*
* This is real Java code, but this code
* cannot be compiled on its own:
*/
public class Chair {
    FlatHorizonalPanel seat;
    FlatVerticalPanel back;
    LongSkinnyVerticalRods legs;
}
```

The code in Listing 13-4 is a design plan for chairs. The code tells you that each chair has three parts. The code names the parts (`seat`, `back`, and `legs`) and tells you a little bit about each part. (For example, a seat is a `FlatHorizontalPanel`.) In the same way, the code in Listing 13-2 tells you that each purchase has three components. The code names the components (unitPrice, quantity, and taxable) and tells you the primitive type of each component.

Imagine some grand factory at the edge of the universe. While you sleep each night, this factory stamps out tangible objects — objects that you'll encounter during the next waking day. Tomorrow you'll go for an interview at the Sloshy Shoes Company. So tonight the factory builds chairs for the company's offices. The factory builds chair objects, as shown in Figure 13-3, from the almost-real code in Listing 13-4.



```
public class Chair {
    FlatHorizonalPanel seat;
    FlatVerticalPanel back;
    LongSkinnyVerticalRods legs;
}
```

FIGURE 13-3: Objects from the Chair class.

In Listing 13-3, the line

```
Purchase onePurchase = new Purchase();
```

behaves like that grand factory at the edge of the universe. Rather than create chair objects, that line in Listing 13-3 creates a Purchase object, as shown in Figure 13-4. That particular line in Listing 13-3 is a declaration with an initialization. Just as the line

```
int count = 0;
```

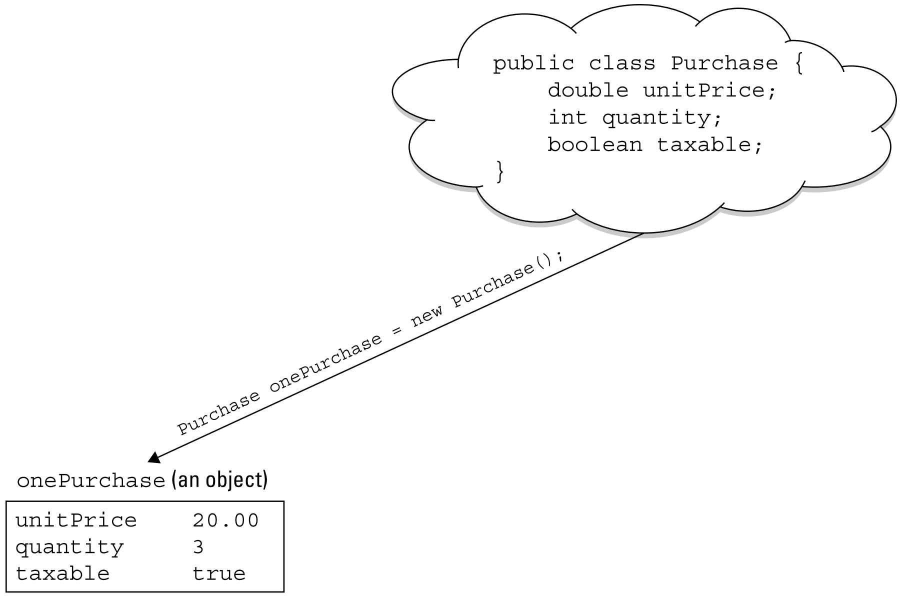


FIGURE 13-4: An object created from the Purchase class.

declares the `count` variable and sets `count` to 0, the line in Listing 13-3 declares the `onePurchase` variable and makes `onePurchase` point to a brand-new object. That new object contains three parts: a `unitPrice` part, a `quantity` part, and a `taxable` part.

TECHNICAL STUFF

If you want to be picky, there's a difference between the stuff in Figure 13-4 and the action of the statement `Purchase onePurchase = new Purchase()` from Listing 13-3. Figure 13-4 shows an object with the values `20.00`, `2`, and `true` stored in it. The statement `Purchase onePurchase = new Purchase()` creates a new object, but it doesn't fill the object with useful values. Getting values comes later in Listing 13-3.

Understanding (or ignoring) the subtleties

Sometimes, when you refer to a particular object, you want to emphasize which class the object came from. Well, subtle differences in emphasis call for big differences in terminology. Here's how Java programmers use the terminology:

- In Listing 13-3, the statement `Purchase onePurchase = new Purchase()` creates a new *object*.
- In Listing 13-3, the statement `Purchase onePurchase = new Purchase()` creates a new *instance* of the `Purchase` class.

The words *object* and *instance* are almost synonymous, but Java programmers never say "object of the Purchase class" (or if they do, they feel funny).

By the way, if you mess up this terminology and say something like "object of the Purchase class," no one jumps down your throat. Everyone understands what you mean, and life goes on as usual. In fact, I often use a phrase like "Purchase object" to describe an instance of the Purchase class. The difference between object and instance isn't terribly important. But it's important to remember that the words *object* and *instance* have the same meaning. (Okay! They have *nearly* the same meaning.)

Making reference to an object's parts

In the Purchase code of Listing 13-2, I declare three variables: a unitPrice variable, a quantity variable, and a taxable variable. In real life, you might say that each purchase has three parts: the unitPrice of an item being purchased, the quantity of items purchased, and the fact that the purchase is or isn't taxable.

When you create a Java class, each of these variables is called a *field.* The Purchase class has three fields — namely, the unitPrice field, the quantity field, and the taxable field.

After you've created an object, you use dots to refer to the object's fields. For example, in Listing 13-3, I put a value into the onePurchase object's unitPrice field with the following code:

```
onePurchase.unitPrice = keyboard.nextDouble();
```

Later in Listing 13-3, I get the unitPrice field's value with the following code:

```
double total = onePurchase.unitPrice * onePurchase.quantity;
```

This dot business may look cumbersome, but it really helps programmers when they're trying to organize the code. In Listing 13-1, each variable is a separate entity. But in Listing 13-3, each use of the word unitPrice is inextricably linked to the notion of a purchase. That's good.

TECHNICAL STUFF

Every field is a variable, but not every variable is a field. For example, in the following code, the amount variable isn't a field because it's declared inside of the main method.

```
    public static void main(String[] args) {
        double amount;
        amount = 5.95;
        // ... Etc.
```

One way or another, I don't want you to get bogged down thinking about the words *field* and *variable.* A *field* is simply a variable that has a special role inside of a class. When I want to emphasize that special role, I use the word *field.* When I don't want to emphasize that special role, I use the word *variable.* I may even switch back and forth between *field* and *variable* in the same sentence. Who knows? I might call something a *fariable* or a *vield.*

If you care about which word I use and when I use it, good for you. If you don't care, don't worry about it.

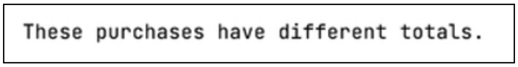
Creating several objects

After you've created a `Purchase` class, you can create as many purchase objects as you want. For example, in Listing 13-5, I create two purchase objects.

LISTING 13-5: Processing Purchases

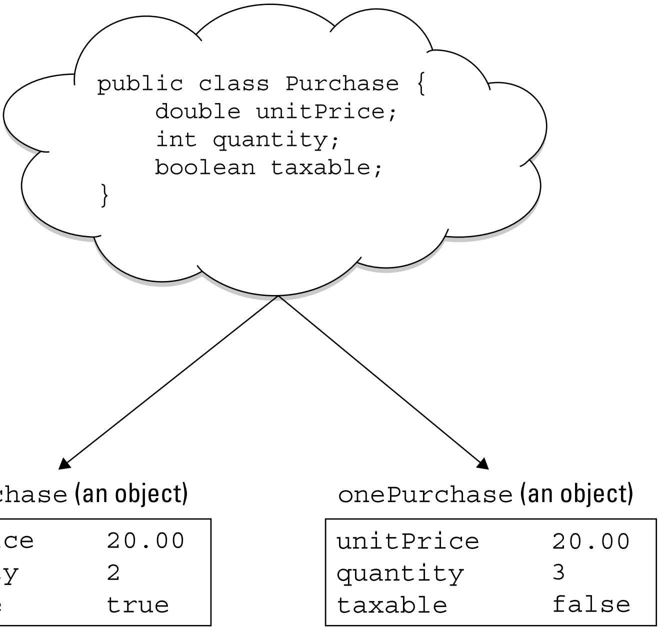
```
public class ProcessPurchases {

    public static void main(String[] args) {
        var purchase1 = new Purchase();
        purchase1.unitPrice = 20.00;
        purchase1.quantity = 3;
        purchase1.taxable = true;

        var purchase2 = new Purchase();
        purchase2.unitPrice = 20.00;
        purchase2.quantity = 3;
        purchase2.taxable = false;

        double purchase1Total = purchase1.unitPrice * purchase1.quantity;
        if (purchase1.taxable) {
            purchase1Total *= 1.05;
        }

        double purchase2Total = purchase2.unitPrice * purchase2.quantity;
        if (purchase2.taxable) {
            purchase2Total *= 1.05;
        }

        if (purchase1Total == purchase2Total) {
            System.out.println("No difference");
        } else {
            System.out.println("These purchases have different totals.");
        }
    }
}
```

Figure 13-5 has a run of the code in Listing 13-5, and Figure 13-6 illustrates the concept.



```
These purchases have different totals.
```

FIGURE 13-5: Running the code in Listing 13-5.



```
public class Purchase {
    double unitPrice;
    int quantity;
    boolean taxable;
}
```

onePurchase (an object)

```
unitPrice      20.00
quantity       2
taxable        true
```

onePurchase (an object)

```
unitPrice      20.00
quantity       3
taxable        false
```

FIGURE 13-6: From one class come two objects.

REMEMBER

To compile the code in Listing 13-5, you must have a copy of the `Purchase` class in the same project. (The `Purchase` class is in Listing 13-2.)

Listing 13-5 has two purchase objects because the code

```
new Purchase();
```

is executed two times.

TIP

Just as you can separate an `int` variable's assignment from the variable's declaration

```
int count;
count = 0;
```

you can also separate a `Purchase` variable's assignment from the variable's declaration:

```
Purchase purchase1;
purchase1 = new Purchase();
```

After you've created the code in Listing 13-2, the word `Purchase` stands for a brand-new type — a reference type. Java has eight built-in primitive types and has as many reference types as people can define during your lifetime. In Listing 13-2, I define the `Purchase` reference type, and you can define reference types, too.

Table 13-2 has a brief comparison of primitive types and reference types.

TABLE 13-2: Java Types

	Primitive Type	Reference Type
How it's created	Built into the language	Defined as a Java class
How many are there	Eight	Indefinitely many
Examples		
Variable declaration	`int count;`	`Purchase aPurchase;`
Assignment	`count = 0;`	`aPurchase = new Purchase();`

Variable declaration with initialization	`int count = 0;`
	or
	`var count = 0;`
		`Purchase aPurchase = new Purchase();`
		or
		`var aPurchase = new Purchase();`

Assigning a value to one of its parts	(Not applicable. A primitive type has no parts.)	`aPurchase.unitPrice = 20.00;`

If it looks like a Purchase and smells like a Purchase . . .

Chapter 6 introduces the word `var` and, from Chapter 7 onward, I use `var` to avoid repeating the word `Scanner` in the examples' declarations. In Listing 13-5, I use `var` to avoid repeating the word `Purchase`. That is, I write

```
var purchase1 = new Purchase();

var purchase2 = new Purchase();
```

instead of writing

```
Purchase purchase1 = new Purchase();

Purchase purchase2 = new Purchase();
```

Whether you start a declaration with `var` or with `Purchase`, the code works correctly. But remember that you must not separate a `var` from its initialization. The declarations of `purchase1` and `purchase2` in Listing 13-5 are fine, but the following code, with assignments instead of initializations, would make alarms go off:

```
// Don't do this:

var purchase1;
purchase1 = new Purchase();

var purchase2;
purchase2 = new Purchase();
```

A line like `var  purchase1;` contains no mention of the `Purchase` class that you declared in Listing 13-2, so Java can't deal with it.

The word `var` comes with another noteworthy restriction. Imagine this variation on the `Purchase` class from Listing 13-2:

```
public class PurchaseWithDefault {
    double unitPrice;
    int quantity = 100;
    boolean taxable;
}
```

In this scenario, you're not interested in selling fewer than 100 units at a time. Most of your customers buy the minimum of 100 units but, occasionally. someone wants to buy more. In that case you change `quantity` to the larger number.

```
int userInput = keyboard.nextInt();
if (userInput > 100) {
    onePurchase.quantity = userInput;
}
```

In the `PurchaseWithDefault` declaration, you give the `quantity` field an initial value, so you may be tempted to change `int quantity` to `var quantity`. But that doesn't work.

```
public class PurchaseWithDefault {
    double unitPrice;
    var quantity = 100; // No! No! No! Don't do this!
    boolean taxable;
}
```

Sorry! You can't use `var` to declare a class's field.

Another Way to Think about Classes

When you start learning object-oriented programming, you may think that this class idea is a big hoax. Some geeks in Silicon Valley had nothing better to do, so they went to a bar and made up some confusing gibberish about classes. They don't know what it means, but they have fun watching people struggle to understand it.

Well, that's not what classes are all about. Classes are serious stuff. What's more, classes are useful. Many reputable studies have shown that classes and object-oriented programming save time and money.

Even so, the notion of a class can be elusive. Even experienced programmers — the ones who are new to object-oriented programming — have trouble understanding how an object differs from a class.

Classes, objects, and tables

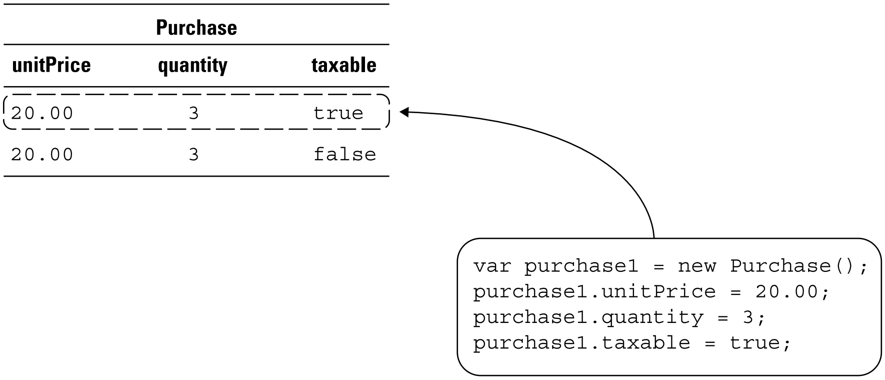
Because classes can be mysterious, I'll expand your understanding with another analogy. Figure 13-7 has a table containing three purchases. The table's title consists of one word (the word *Purchase*), and the table has three column headings: the words *unitPrice, quantity,* and *taxable.* Well, the code in Listing 13-2 has the same stuff — `Purchase`, `unitPrice`, `quantity`, and `taxable`. So, in Figure 13-7, think of the top part of the table (the title and column headings) as a class. Like the code in Listing 13-2, this top part of the table tells you what it means to be a `Purchase`. (It means having a `unitPrice` value, a `quantity` value, and a `taxable` value.)

```
public class Purchase {
    double unitPrice;
    int quantity;
    boolean taxable;
}
```

	Purchase	
unitPrice	quantity	taxable
20.00	3	true
20.00	3	false

FIGURE 13-7: A table of purchases.

A class is like the top part of a table. And what about an object? Well, an object is like a row of a table. For example, with the code in Listing 13-5, I create two objects (two instances of the `Purchase` class). The first object has `unitPrice` value `20.00`, `quantity` value `3`, and `taxable` value `true`. In the corresponding table, the first row has these three values — `20.00`, `3`, and `true`, as shown in Figure 13-8.



	Purchase	
unitPrice	quantity	taxable
20.00	3	true
20.00	3	false

```
var purchase1 = new Purchase();
purchase1.unitPrice = 20.00;
purchase1.quantity = 3;
purchase1.taxable = true;
```

FIGURE 13-8: A purchase corresponds to a row of the table.

Some questions and answers

Here's the world's briefest object-oriented programming FAQ:

- **Can I have an object without having a class?**

  No, you can't. In Java, every object is an instance of a class.

- **Can I have a class without having an object?**

  Yes, you can. In fact, almost every program in this book creates a class without an object. Take Listing 13-5, for example. The code in Listing 13-5 defines a class named ProcessPurchases. And nowhere in Listing 13-5 (or anywhere else) do I create an instance of the ProcessPurchases class. I have a class with no objects. That's just fine. It's business as usual.

- **After I've created a class and its instances, can I add more instances to the class?**

  Yes, you can. In Listing 13-5, I create one instance and then another. In a for loop, I could create a dozen instances and I'd have a dozen rows in the table of Figure 13-8. With no objects, three objects, four objects, or more objects, I still have the same old Purchase class.

- **Can an object come from more than one class?**

  Bite your tongue! Maybe other object-oriented languages allow this nasty class cross-breeding, but in Java, it's strictly forbidden. That's one thing that distinguishes Java from some of the languages that preceded it: Java is cleaner, more uniform, and easier to understand.

What's Next?

Listing 13-5 contains some code that makes me nervous. In Listing 13-5, the statements that compute a purchase total appear once, and then appear a second time with only one tiny change:

```
double purchase1Total = purchase1.unitPrice * purchase1.quantity;
if (purchase1.taxable) {
    purchase1Total *= 1.05;
}

double purchase2Total = purchase2.unitPrice * purchase2.quantity;
if (purchase2.taxable) {
    purchase2Total *= 1.05;
}
```

To me, this repetition seems silly. Aren't computers supposed to save us from mundane burdens such as repetitive typing? What if I type these statements correctly the first time but make a mistake the second time? Maybe I'll copy the bad version a third and fourth time and end up with code that's all messed up!

Repetitive code is error-prone. It's inconvenient and unnecessary. Rather than repeat code, you should be able to summarize your code and then refer to that summarized code repeatedly. With Java's methods, you have precisely that capability. You've used other peoples' methods in many examples throughout this book, and in Chapter 15, you create methods of your own.

TRY IT OUT

This is your chance to write some exceedingly classy code.

WHAT'S YOUR BMI?

A person's body mass index (BMI) is the person's weight (in kilograms) divided by the square of the person's height (in meters). For example, a 65 kg person who's 1.65 meters tall has a BMI of 23.875.

- Create a `Person` class. Each `Person` has a weight and a height. In other words, the `Person` class has two fields: a `weight` field (a `double` value) and a `height` field (another `double` value).
- Create another class containing a `main` method. In the `main` method, create a `Person` object. Assign values to the `Person` object's `weight` and `height` fields by reading input from the keyboard. Use the `Person` object's `weight` and `height` fields to calculate the person's BMI. Then output the person's BMI.
- Modify the `main` method that you wrote for the previous bullet so that it creates three `Person` objects and calculates each object's BMI.

A BIT OF MACROECONOMICS

A country's debt-to-GDP ratio is the amount of the government's debt divided by the country's gross domestic product (GDP). For example, a country with $14.3 trillion of debt and $18.5 trillion GDP has a debt-to-GDP ratio of approximately 77 percent. A low debt-to-GDP ratio means that a country can pay off its debts without having to incur even more debt.

Try writing the following code:

- Create a `Country` class. Each `Country` has a `debt` and a gdp. In other words, the `Country` class has two fields: a `debt` field (a `double` value) and a gdp field (another `double` value).
- Create another class containing a `main` method. In the `main` method, create a `Country` object. Assign values to the `Country` object's `debt` and gdp fields by reading input from the keyboard.

  Also in the `main` method, ask the user for an acceptable debt-to-GDP ratio. If your `Country` object's debt-to-GDP ratio is lower than the acceptable value, output the words `That's acceptable`. Otherwise, output `That's not acceptable`.

- Modify the `main` method you wrote for the previous bullet so that it creates three `Country` objects and displays `That's acceptable` or `That's not acceptable` for each object.

NOTHING IN PARTICULAR

This program isn't about a `Purchase` class, a `Person` class, a `Country` class, or a class with any obvious real-world relevance:

- Create a `Thing` class. The `Thing` class has two fields: a `value1` field (an `int` value) and a `value2` field (another `int` value).
- Create another class containing a `main` method. In the `main` method, create a `Thing` object. Assign values to the `Thing` object's `value1` and `value2` fields by reading input from the keyboard. Use the `Thing` object's `value1` and `value2` fields to display a sentence about the object. A typical sentence might be

  ```
  This thing has values 42 and 91.
  ```

- Modify the `main` method you wrote for the previous bullet so that it creates three `Thing` objects and displays a sentence about each of them.

MAKE A HIT RECORD

Newer versions of Java have a fancy feature called *record classes*. For an introduction to these beauties, name two files `PurchaseRecord.java` and `ProcessPurchaseRecord.java`. Put the following code in these files and then give the code a spin.

```
// PurchaseRecord.java

public record PurchaseRecord(double unitPrice,
                             int quantity,
                             boolean taxable) {
}

// ProcessPurchaseRecord.java

import java.util.Scanner;

public class ProcessPurchaseRecord {

    public static void main(String[] args) {
```

```
        var keyboard = new Scanner(System.in);

        System.out.print("Unit price: ");
        var unitPrice = keyboard.nextDouble();
        System.out.print("Quantity: ");
        var quantity = keyboard.nextInt();
        System.out.print("Taxable? (true/false) ");
        var taxable = keyboard.nextBoolean();

        var onePurchase = new PurchaseRecord(unitPrice, quantity, taxable);

        double total = onePurchase.unitPrice() * onePurchase.quantity();
        if (onePurchase.taxable()) {
            total = total * 1.05;
        }

        System.out.print("Total: ");
        System.out.println(total);

        keyboard.close();
    }
}
```

IN THIS CHAPTER

- » Using Java's `String` class
- » Calling methods
- » Understanding static and nonstatic methods and fields
- » Making numbers look good

Chapter 14
Using Methods and Fields from a Java Class

I hope you didn't read Chapter 13, because I tell a big lie at the beginning of that chapter. Actually, it's not a lie. It's an exaggeration.

Actually, it's not an exaggeration. It's a careful choice of wording. In Chapter 13, I write that the gathering of data into a class is the start of object-oriented programming. Well, that's true. Except that some programming languages had data-gathering features before object-oriented programming became popular. Pascal had *records.* C had *structs.*

To be painfully precise, the grouping of data into usable chunks is only a prerequisite to object-oriented programming. You're not really doing object-oriented programming until you combine both data and methods in your classes.

This chapter starts the data-and-methods ball rolling, and Chapter 15 rounds out the picture.

Long Live the String!

The String class is declared in the Java API. This means that somewhere in the stuff you download from https://adoptopenjdk.net is a file named String.java. If you hunt down this String.java file and peek at the file's code, you find some familiar-looking stuff:

```
public class String {
  // ... And so on.
```

In your own code, you can use this String class without ever seeing what's inside the String.java file. That's one of the great things about object-oriented programming.

A simple example

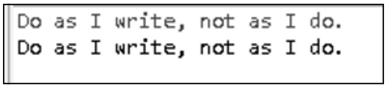
A String is a bunch of characters. It's like having several char values in a row. You can declare a variable to be of type String and store several letters in the variable. Listing 14-1 has a tiny example.

LISTING 14-1: I'm Repeating Myself Again (Again)

```
import java.util.Scanner;

public class JazzyEchoLine {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        String lineIn;

        lineIn = keyboard.nextLine();
        System.out.println(lineIn);

        keyboard.close();
    }
}
```

A run of Listing 14-1 is shown in Figure 14-1. This run bears an uncanny resemblance to runs in Listing 5-1 (in Chapter 5). That's because Listing 14-1 is a reprise of the effort in Listing 5-1.

The new idea in Listing 14-1 is the role that `String` plays. In Listing 5-1, I have no variable to store the user's input. But in Listing 14-1, I create the `lineIn` variable. This variable stores a bunch of letters, like the letters `Do as I write, not as I do`.



```
Do as I write, not as I do.
Do as I write, not as I do.
```

FIGURE 14-1: Running the code in Listing 14-1.

Putting String variables to good use

The program in Listing 14-1 takes the user's input and echoes it back on the screen. This is a wonderful program, but (like many college administrators that I know) it doesn't seem to be particularly useful.

Take a look at a more useful application of Java's `String` type. A nice one is in Listing 14-2.

LISTING 14-2: Putting a Name in a String Variable

```
import java.util.Scanner;
import static java.lang.System.out;

public class ProcessMoreData {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        String fullName;
        double amount;
        boolean taxable;
        double total;

        out.print("Customer's full name: ");
        fullName = keyboard.nextLine();
        out.print("Amount: ");
        amount = keyboard.nextDouble();
        out.print("Taxable? (true/false) ");
        taxable = keyboard.nextBoolean();
        if (taxable) {
            total = amount * 1.05;
        } else {
            total = amount;
        }
```

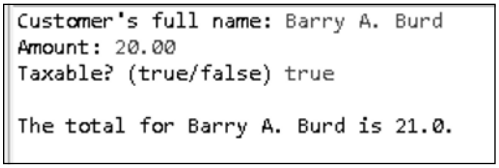
(continued)

LISTING 14-2: ***(continued)***

```
        out.println();
        out.print("The total for ");
        out.print(fullName);
        out.print(" is ");
        out.print(total);
        out.println(".");

        keyboard.close();
    }
}
```

A run of the code in Listing 14-2 is shown in Figure 14-2. The code stores `Barry A. Burd` in a variable called `fullName` and displays the `fullName` variable's content as part of the output. To make this program work, you have to store `Barry A. Burd` somewhere. After all, the program follows a certain outline:

```
Get a name.
Get some other stuff.
Compute the total.
Display the name (along with some other stuff).
```



```
Customer's full name: Barry A. Burd
Amount: 20.00
Taxable? (true/false) true

The total for Barry A. Burd is 21.0.
```

FIGURE 14-2: Making a purchase.

If you don't have the program store the name somewhere, by the time it finishes getting other stuff and computing the total, it forgets the name (so the program can't display the name).

Reading and writing strings

To read a `String` value from the keyboard, you can call either `next` or `nextLine`:

- **The method `next` reads up to the next blank space.**

  For example, with the input `Barry A. Burd`, the statements

  ```
  String firstName = keyboard.next();
  String middleInit = keyboard.next();
  String lastName = keyboard.next();
  ```

- assign Barry to firstName, A. to middleInit, and Burd to lastName.

  The method nextLine reads to the end of the current line.

  For example, with input Barry A. Burd, the statement

  ```
  String fullName = keyboard.nextLine();
  ```

  assigns Barry A. Burd to the variable fullName. (Hey, being an author has some hidden perks.)

To display a String value, you can call one of your old friends — System.out.print or System.out.println. In fact, most of the programs in this book display String values. In Listing 14-2, a statement like

```
out.print("Customer's full name: ");
```

displays the String value "Customer's full name: ".

TIP

You can use print and println to write String values to a disk file. For details, see Chapter 16.

Chapter 4 introduces a bunch of characters, enclosed in double quote marks:

```
"Chocolate, royalties, sleep"
```

In Chapter 4, I call this a *literal* of some kind. (It's a literal because, unlike a variable, it looks just like the stuff it represents.) Well, in this chapter, I can continue the story about Java's literals:

- In Listing 14-2, amount and total are double variables, and 1.05 is a double literal.
- In Listing 14-2, fullName is a String variable, and things like "Customer's full name: " are String literals.

REMEMBER

In a Java program, you surround the letters in a String literal with double quote marks.

TRY IT OUT

With enough practice using Java `String` values, you'll never get tied up in knots.

PLUS-SIZE PRINTS

Using Java's plus sign (+), replace all statements from `out.print("The total for ")` to `out.println(".")` with a single `out.println` call.

FILL IN THE BLANKS

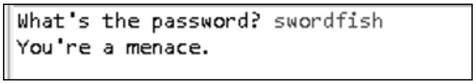
Linguist Noam Chomsky once wrote that "Colorless green ideas sleep furiously." Chomsky wasn't crazy. He was showing that a grammatically correct sentence can be completely meaningless. Write a program that prompts the user for five words: two adjectives, a plural noun, a verb, and an adverb. In the end, the program displays the sentence containing those five words. Red big toads marry nightmares. Electric international oceans eat cheese.

FILL IN MORE BLANKS

Expand your fill-in-the-blanks code so that it has a more varied sentence structure. A deliberate mistake makes houses into prunes. All dashboards fly when you buy an atom.

Using an Object's Methods

If you're not too concerned about classes and reference types, the use of the type `String` in Listing 14-2 is no big deal. Almost everything you can do with a primitive type seems to work with the `String` type as well. But danger lies around the next curve. Take a look at the code in Listing 14-3 and the run of the code shown in Figure 14-3.



```
What's the password? swordfish
You're a menace.
```

FIGURE 14-3: But I typed the correct password!

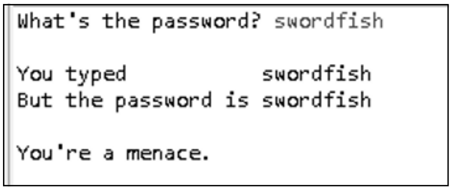
LISTING 14-3: A Faulty Password Checker

```
/*
 * This code does not work:
 */

import java.util.Scanner;

import static java.lang.System.out;

public class TryToCheckPassword {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        String password = "swordfish";
        String userInput;

        out.print("What's the password? ");
        userInput = keyboard.next();

        if (password == userInput) {
            out.println("You're okay!");
        } else {
            out.println("You're a menace.");
        }

        keyboard.close();
    }
}
```

Here are the facts as they appear in this example:

- According to the code in Listing 14-3, the value of `password` is `"swordfish"`.
- In Figure 14-3, in response to the program's prompt, the user types `swordfish`. So, in the code, the value of `userInput` is `"swordfish"`.
- The `if` statement checks the condition `password == userInput`. Because both variables have the value `"swordfish"`, the condition *should* be true, but . . .
- . . . the condition is *not* true because the program's output is `You're a menace`.

What's going on here? I try beefing up the code to see whether I can find any clues. An enhanced version of the password-checking program is in Listing 14-4, with a run of the new version shown in Figure 14-4.



```
What's the password? swordfish

You typed           swordfish
But the password is swordfish

You're a menace.
```

FIGURE 14-4: This looks even worse.

LISTING 14-4: An Attempt to Debug the Code in Listing 14-3

```
import java.util.Scanner;

import static java.lang.System.out;

public class DebugCheckPassword {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        String password = "swordfish";
        String userInput;

        out.print("What's the password? ");
        userInput = keyboard.next();

        out.println();
        out.print("You typed ");
        out.println(userInput);
        out.print("But the password is ");
        out.println(password);
        out.println();

        if (password == userInput) {
            out.println("You're okay!");
        } else {
            out.println("You're a menace.");
        }

        keyboard.close();
    }
}
```

Ouch! I'm stumped this time. The run in Figure 14-4 shows that both the `userInput` and `password` variables have value `swordfish`. Why doesn't the program accept the user's input?

When you compare two things with a double equal sign, reference types and primitive types don't behave the same way. Consider, for example, `int` versus `String`:

- You can compare two `int` values with a double equal sign. When you do, things work exactly as you would expect. For example, the condition in the following code is true:

  ```
  int apples = 7;
  int oranges = 7;
  if (apples == oranges) {
      System.out.println("They're equal.");
  }
  ```

- When you compare two `String` values with the double equal sign, things don't work the way you expect. The computer doesn't check to see whether the two `String` values contain the same letters. Instead, the computer checks some esoteric property of the way variables are stored in memory.

REMEMBER

For your purposes, the term *reference type* is just a fancy name for a class. Because `String` is defined to be a class in the Java API, I call `String` a reference type. This terminology highlights the parallel between primitive types (such as `int`) and classes (that is, reference types, such as `String`).

Comparing strings

In the bullets in the previous section, the difference between `int` and `String` is mighty interesting. But if the double equal sign doesn't work for `String` values, how do you check to see if Joe User enters the correct password? You do it with the code in Listing 14-5.

LISTING 14-5: Calling an Object's Method

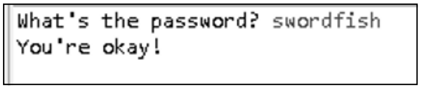
```
/*
 * This program works!
 */

import java.util.Scanner;

import static java.lang.System.out;

public class CheckPassword {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        String password = "swordfish";
        String userInput;
```

(continued)

LISTING 14-5: ***(continued)***

```
        out.print("What's the password? ");
        userInput = keyboard.next();

        if (password.equals(userInput)) {
            out.println("You're okay!");
        } else {
            out.println("You're a menace.");
        }

        keyboard.close();
    }
}
```

A run of the new password-checking code is shown in Figure 14-5, and let me tell you, it's a big relief! The code in Listing 14-5 actually works! When the user types `swordfish`, the `if` statement's condition is true.



```
What's the password? swordfish
You're okay!
```

FIGURE 14-5: At last, Joe User can log in.

The truth about classes and methods

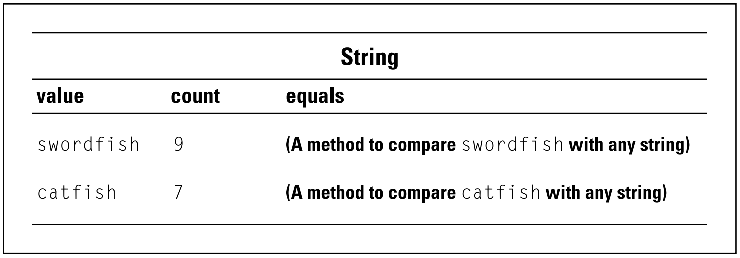
The magic in Listing 14-5 is the use of a method named `equals`. I have two ways to explain the `equals` method: a simple way and a more detailed way. First, here's the simple way: The `equals` method compares the characters in one string with the characters in another. If the characters are the same, the condition inside the `if` statement is true. That's all there is to it.

REMEMBER

Don't use a double equal sign to compare two `String` objects. Instead, use one of the object's `equals` methods.

For a more detailed understanding of the `equals` method, flip to Chapter 13 and take a look at Figures 13-6 and 13-7. Those figures illustrate the similarities between classes, objects, and the parts of a table. In the figures, each row represents a purchase, and each column represents a feature that purchases possess.

You can observe the same similarities for any class, including Java's `String` class. In fact, what Figure 13-6 does for purchases, Figure 14-6 does for strings.



String

value	count	equals
swordfish	9	**(A method to compare** swordfish **with any string)**
catfish	7	**(A method to compare** catfish **with any string)**

FIGURE 14-6: Viewing the String class and String objects as parts of a table.

The stuff shown in Figure 14-6 is much simpler than the real String class story. But Figure 14-6 makes a good point. Like the purchases in Figure 13-6, each string has its own features. For example, each string has a value (the actual characters stored in the string), and each string has a count (the number of characters stored in the string). You can't really write the following line of code because the stuff in Figure 14-6 omits a few subtle details:

```
//This code does NOT work:
System.out.println(password.count);
```

Anyway, each row in Figure 14-6 has three items: a value, a count, and an equals method. So each row of the table contains more than just data. Each row contains an equals method, a way of doing something useful with the data. It's as though each object (each instance of the String class) has three things:

- A bunch of characters (the object's value)
- A number (the object's count)
- A way of being compared with other strings (the object's equals method)

That's the essence of object-oriented programming. Each string has its own, personal copy of the equals method. For example, in Listing 14-5, the password string has its own equals method. When you call the password string's equals method and put the userInput string in the method's parentheses, the method compares the two strings to see whether those strings contain the same characters.

The userInput string in Listing 14-5 has an equals method, too. I could use the userInput string's equals method to compare this string with the password string. But I don't. In fact, in Listing 14-5, I don't use the userInput string's equals method at all. (To compare the userInput with the password, I had to use either the password string's equals method or the userInput string's equals method. I made an arbitrary choice: I chose the password string's method.)

Calling an object's methods

In Chapter 13, I create a `Purchase` class:

```
public class Purchase {
    double unitPrice;
    int quantity;
    boolean taxable;
}
```

I refer to the `unitPrice` variable, the `quantity` variable, and the `taxable` variable as the `Purchase` class's *fields.*

Calling a string's `equals` method is like getting a purchase's `unitPrice`. With both `equals` and `unitPrice`, you use your old friend, the dot. For example, in Listing 13-3 (in Chapter 13), you write

```
onePurchase.unitPrice = keyboard.nextDouble();
```

and in Listing 14-5, you write

```
if (password.equals(userInput))
```

A dot works the same way for an object's fields and its methods. In either case, a dot takes the object and picks out one of the object's parts. It works whether that part is a field (as in `onePurchase.unitPrice`) or a method (as in `password.equals`).

In fact, fields and methods are similar in so many ways that it's handy to have one word to describe both fields and methods. The word is *members.* In Chapter 13, the `Purchase` class has three members: `unitPrice`, `quantity`, and `taxable`. And in the Java API, the `String` class has about 60 members, one of which is `equals`.

Combining and using data

At this point in the chapter, I can finally say, "I told you so." Here's a quotation from Chapter 13:

> A class is a design plan; it describes the way in which you intend to *combine* and *use* pieces of data.

A class can define the way you *use* data. How do you use a password and a user's input? You check to see whether they're the same. That's why Java's `String` class defines an `equals` method.

REMEMBER

An object can be more than just a bunch of data. With object-oriented programming, each object possesses copies of methods for using that object.

TRY IT OUT

Java's `String` class is impressive, with nearly 60 methods. But that's nothing compared with Java's `JFrame` class. The `JFrame` class has hundreds of methods.

SHOW A FRAME

Identify the method calls in the following code:

```
import javax.swing.JFrame;

public class Main {

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300, 300);
        frame.setTitle("This is a frame");
        frame.setVisible(true);
    }
}
```

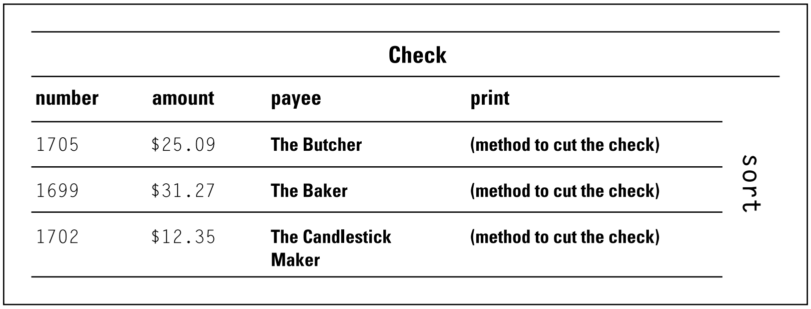
Run the code to find out what it does. (End the run of the code by clicking the little red square near the upper-right corner of IntelliJ's main window. For details, refer to Chapter 3.)

CHECK THE DOCUMENTATION

Visit `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html` to read about the methods belonging to Java's `JFrame` class. In particular, read about methods named `setSize`, `setTitle`, and `setVisible`.

How to Achieve Static Equilibrium

You have a fistful of checks. Each check has a number, an amount, and a payee. You print checks like these with your very own printer. To print the checks, you use a Java class. Each object made from the `Check` class has three fields (`number`, `amount`, and `payee`). And each object has one method (a `print` method). You can see all of this in Figure 14-7.



Check			
number	amount	payee	print
1705	$25.09	The Butcher	(method to cut the check)
1699	$31.27	The Baker	(method to cut the check)
1702	$12.35	The Candlestick Maker	(method to cut the check)

FIGURE 14-7: The Check class and some Check objects.

You'd like to print the checks in numerical order. So you need a method to *sort* the checks. If the checks in Figure 14-7 were sorted, the check with number 1699 would come first, and the check with number 1705 would come last.

The big question is, should each check have its own sort method? Does the check with number 1699 need to sort itself? And the answer is no. Some methods just shouldn't belong to the objects in a class.

Where do such methods belong? How can you have a sort method without creating a separate sort for each check?

Here's the answer: You make the sort method be *static.* Anything that's static belongs to a whole class, not to any particular instance of the class. If the sort method is static, the entire Check class has just one copy of the sort method. This copy stays with the entire Check class. No matter how many instances of the Check class you create — three, ten, or none — you have just one sort method.

For an illustration of this concept, refer to Figure 14-7. The whole class has just one sort method. So the sort method is static. No matter how you call the sort method, that method uses the same values to do its work. For the data in Figure 14-7, a single call to the sort method uses all three rows (the 1705, 1699, and 1702 rows).

Of course, each individual check (each object, each row of the table in Figure 14-7) still has its own number, its own amount, its own payee, and its own print method. When you print the first check, you get one amount, and when you print the second check, you get another. Because there's a number, an amount, a payee, and a print method for each object, I call these things *nonstatic.* I call them nonstatic because . . . well . . . because they're not static.

Calling static and nonstatic methods

In this book, my first use of the word static is in Listing 3-1. I use static as part of every main method (and this book's listings have lots of main methods). In Java, your main method has to be static. That's just the way it goes.

To call a static method, you use a class's name along with a dot. This is just slightly different from the way you call a nonstatic method:

- **To call an ordinary (nonstatic) method, you follow an object with a dot.**

  For example, a program to process the checks in Figure 14-7 may contain code of the following kind:

  ```
  Check firstCheck;
  firstCheck.number = 1705;
  firstCheck.amount = 25.09;
  firstCheck.payee = "The Butcher";
  firstCheck.print();
  ```

- **To call a class's static method, you follow the class name with a dot.**

  For example, to sort the checks in Figure 14-7, you may call

  ```
  Check.sort();
  ```

Turning strings into numbers

The code in Listing 14-5 introduces a nonstatic method named `equals`. To compare the `password` string with the `userInput` string, you preface `.equals` with either of the two string objects. In Listing 14-5, I preface `.equals` with the `password` object:

```
if (password.equals(userInput))
```

Each string object has an `equals` method of its own, so I can achieve the same effect by writing

```
if (userInput.equals(password))
```

But Java has another class named `Integer`, and the whole `Integer` class has a static method named `parseInt`. If someone hands you a string of characters and you want to turn that string into an `int` value, you can call the `Integer` class's `parseInt` method. Listing 14-6 has a small example.

WARNING

Don't confuse `Integer` with `int`. In Java, `int` is the name of a primitive type (a type that I use throughout this book). But `Integer` is the name of a class. Java's `Integer` class contains handy methods for dealing with `int` values. For example, in Listing 14-6, the `Integer` class's `parseInt` method makes an `int` value from a `string`. Java's `Integer` class is an example of a wrapper class. You can read more about wrapper classes in Chapter 19.

LISTING 14-6: More Chips, Please

```
import java.util.Scanner;

import static java.lang.System.out;

public class AddChips {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        String reply;
        int numberOfChips;

        out.print("""
                How many chips do you have?
                  (Type a number, or type 'Not playing')\s""");
        reply = keyboard.nextLine();

        if (!reply.equals("Not playing")) {
            numberOfChips = Integer.parseInt(reply);
            numberOfChips += 10;

            out.print("You now have ");
            out.print(numberOfChips);
            out.println(" chips.");
        }

        keyboard.close();
    }
}
```

Some runs of the code in Listing 14-6 are shown in Figure 14-8. You want to give each player ten chips. But some party poopers in the room aren't playing. So two people, each with no chips, may not get the same treatment. An empty-handed player gets ten chips, but an empty-handed party pooper gets none.

In Listing 14-6, you call the `Scanner` class's `nextLine` method, allowing a user to enter any characters at all — not just digits. If the user types `Not playing`, you don't give the killjoy any chips.

If the user types some digits, you're stuck holding these digits in the string variable named `reply`. You can't add ten to a string like `reply`. So you call the `Integer` class's `parseInt` method, which takes your string and hands you back a nice `int` value. From there, you can add ten to the `int` value.

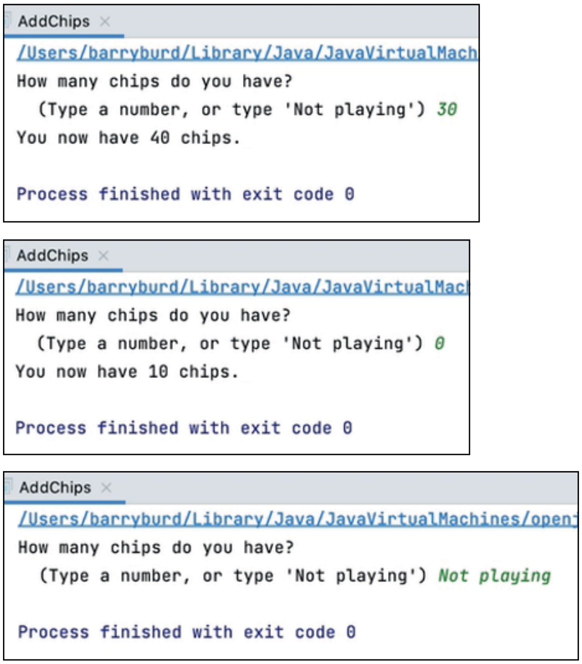
TIP

When you create a text block, Java takes liberties with the blank space in the text. For example, the text block in Listing 14-6 has many blank spaces (represented here by dots).

```
........out.print("""
................How many chips do you have?
..................(Type a number, or type 'Not playing')\s""");
```

But, in Figure 14-8, there are no blank spaces before `How many chips` and only two blank spaces before `(Type a number`. Text blocks also ignore blank spaces at the ends of lines. So you have to remind Java that you don't want this ugly business:

```
(Type a number, or type 'Not playing')10
```



```
/Users/barryburd/Library/Java/JavaVirtualMach
How many chips do you have?
  (Type a number, or type 'Not playing') 30
You now have 40 chips.

Process finished with exit code 0
```

```
/Users/barryburd/Library/Java/JavaVirtualMac
How many chips do you have?
  (Type a number, or type 'Not playing') 0
You now have 10 chips.

Process finished with exit code 0
```

```
/Users/barryburd/Library/Java/JavaVirtualMachines/open
How many chips do you have?
  (Type a number, or type 'Not playing') Not playing

Process finished with exit code 0
```

FIGURE 14-8: Running the code in Listing 14-6.

It's ugly because the user's response (the number `10`) butts right up against the program's prompt. In Listing 14-6, the escape sequence `\s` reminds Java to put a blank space after the program's prompt.

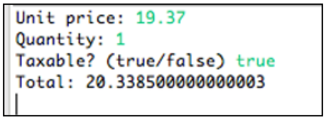
WARNING

Don't confuse Integer with int. In Java, int is the name of a primitive type (a type that I use throughout this book). But Integer is the name of a class. Java's Integer class contains handy methods for dealing with int values. For example, in Listing 14-6, the Integer class's parseInt method makes an int value from a string.

Turning numbers into strings

In Chapter 13, Listing 13-1 computes the price of a purchase. But a run of the code in Listing 13-1 has an anomaly. (Refer to Figure 13-1.) With 5 percent tax on 40 dollars, the program displays a total of 42.0. That's peculiar — where I come from, currency amounts aren't normally displayed with just one digit beyond the decimal point.

If you don't choose your purchase amount carefully, the situation is even worse. For example, in Figure 14-9, I run the same program (the code in Listing 13-1) with the purchase amount 19.37. The resulting display looks nasty.



```
Unit price: 19.37
Quantity: 1
Taxable? (true/false) true
Total: 20.338500000000003
```

FIGURE 14-9: Do you have change for 20.3385000 00000003?

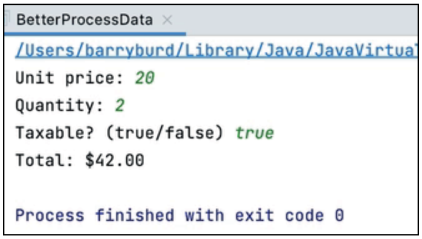
With its internal zeros and ones, the computer doesn't do arithmetic quite the way you and I are used to doing it. So, how do you fix this problem?

The Java API has a class named NumberFormat, and the NumberFormat class has a static method named getCurrencyInstance. When you call NumberFormat.getCurrencyInstance() with nothing inside the parentheses, you get an object that can mold numbers into US currency amounts. Listing 14-7 has an example.

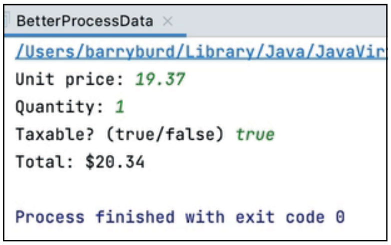
LISTING 14-7: **The Right Way to Display a Dollar Amount**

```
import java.text.NumberFormat;

import java.util.Scanner;

public class BetterProcessData {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        double unitPrice;
        int quantity;
        boolean taxable;

        NumberFormat currency = NumberFormat.getCurrencyInstance();

        System.out.print("Unit price: ");
        unitPrice = keyboard.nextDouble();
        System.out.print("Quantity: ");
        quantity = keyboard.nextInt();
        System.out.print("Taxable? (true/false) ");
        taxable = keyboard.nextBoolean();

        double total = unitPrice * quantity;
        if (taxable) {
            total = total * 1.05;
        }

        String niceTotal = currency.format(total);
        System.out.print("Total: ");
        System.out.println(niceTotal);

        keyboard.close();
    }
}
```

To see some beautiful runs of the code in Listing 14-7, check out Figure 14-10. Now at last, you see a total like $20.34, not 20.338500000000003. Ah! That's much better.



```
/Users/barryburd/Library/Java/JavaVirtua
Unit price: 20
Quantity: 2
Taxable? (true/false) true
Total: $42.00

Process finished with exit code 0
```



```
/Users/barryburd/Library/Java/JavaVir
Unit price: 19.37
Quantity: 1
Taxable? (true/false) true
Total: $20.34

Process finished with exit code 0
```

FIGURE 14-10: See the pretty numbers.

Turning numbers into nice looking strings

For my current purposes, the code in Listing 14-7 contains three interesting variables:

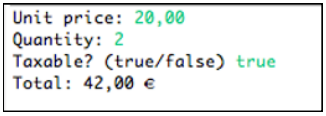
- The variable `total` stores a number, such as 42.0.
- The variable `currency` stores an object that can mold numbers into US currency amounts.
- The variable `niceTotal` is set up to store a bunch of characters.

The `currency` object has a `format` method. To get the appropriate bunch of characters into the `niceTotal` variable, you call the `currency` object's `format` method. You apply this `format` method to the variable `total`.

Your country; your currency

The code in Listing 14-7 works well in the United States. But in another country, the currency symbol might not be the dollar sign ($), and you might represent *twenty* with characters other than `20.00`.

Java shapes its input and output to match your computer's locale. Imagine, for example, that your computer runs the version of Windows sold in France. Then, as far as Java is concerned, your computer's locale is `Locale.FRANCE`, and a run of the code in Listing 14-7 looks like the run shown in Figure 14-11.



```
Unit price: 20,00
Quantity: 2
Taxable? (true/false) true
Total: 42,00 €
```

FIGURE 14-11: A run of Listing 14-7 on a computer in France.

In fact, you can customize your code for many countries, and you don't have to buy airplane tickets to do it! My computer is configured to run in the United States. But in Listing 14-8, I use Java's `Locale` class to get the run shown in Figure 14-11.

LISTING 14-8: Using a Java Locale

```
import java.text.NumberFormat;

import java.util.Locale;
import java.util.Scanner;
```

```
public class EvenBetterProcessData {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        keyboard.useLocale(Locale.FRANCE);

        double unitPrice;
        int quantity;
        boolean taxable;

        var currency = NumberFormat.getCurrencyInstance(Locale.FRANCE);

        System.out.print("Unit price: ");
        unitPrice = keyboard.nextDouble();
        System.out.print("Quantity: ");
        quantity = keyboard.nextInt();
        System.out.print("Taxable? (true/false) ");
        taxable = keyboard.nextBoolean();

        double total = unitPrice * quantity;
        if (taxable) {
            total = total * 1.05;
        }

        String niceTotal = currency.format(total);
        System.out.print("Total: ");
        System.out.println(niceTotal);

        keyboard.close();
    }
}
```

The View from On High

In this section, I answer some of the burning questions that I raise throughout this book: “What does `java.util` stand for?” “Why do I need the word `static` at certain points in the code?” “How can a degree in horticultural studies help you sort canceled checks?”

I also explain static in some unique and interesting ways. After all, static fields and methods aren’t easy to grasp. It helps to read about Java’s static feature from several points of view.

Unravelling Java's import declarations

In Java, you can group a bunch of classes into something called a *package.* In fact, the classes in Java's standard API are divided into about 200 packages. This book's examples make heavy use of three packages: the packages named java.util, java.lang, and java.io.

The class java.util.Scanner

The package java.util contains about 75 classes, including the useful Scanner class. Like most other classes, this Scanner class has two names: a *fully qualified name* and an abbreviated *simple name.* The class's fully qualified name is java.util.Scanner, and the class's simple name is Scanner. You get the fully qualified name by adding the package name to the class's simple name. (That is, you add the package name java.util to the simple name Scanner. You get java.util.Scanner.)

An import declaration lets you abbreviate a class's name. With the declaration

```
import java.util.Scanner;
```

the Java compiler figures out where to look for the Scanner class. So rather than write java.util.Scanner throughout your code, you can just write Scanner.

The class java.lang.System

The package java.lang contains about 100 classes, including the ever-popular System class. (The class's fully qualified name is java.lang.System, and the class's simple name is System.) Rather than write java.lang.System throughout your code, you can just write System. You don't even need an import declaration.

TECHNICAL STUFF

Among all of Java's packages, the java.lang package is special. With or without an import declaration, the compiler imports everything in the java.lang package. You can start your program with import java.lang.System. But if you don't, the compiler adds this declaration automatically.

The static System.out variable

What kind of importing must you do in order to abbreviate System.out.println? How can you shorten it to out.println? An import declaration lets you abbreviate a *class's* name. But in the expression System.out, the word out isn't a class. The word out is a static variable. (The out variable refers to the place where a Java program sends text output.) So you can't write

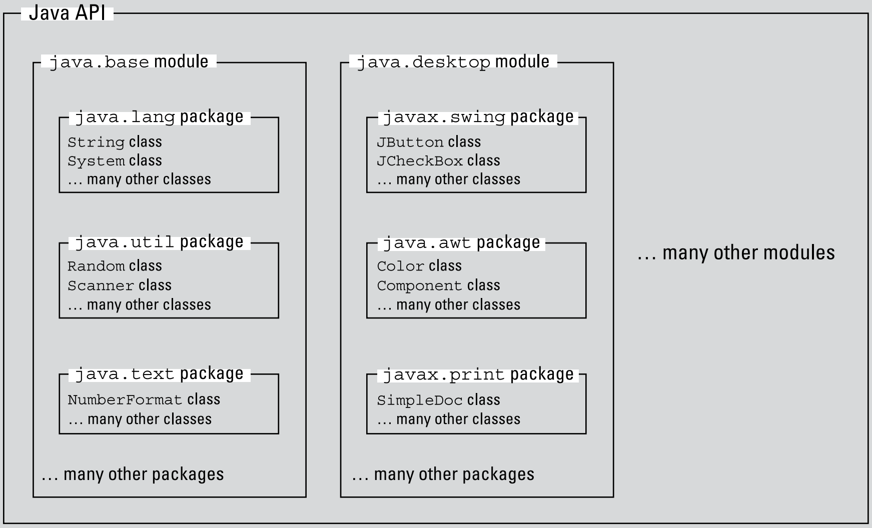
```
//This code is bogus. Don't use it:
import java.lang.System.out;
```

ALL YE NEED TO KNOW

I can summarize much of Java's complexity in only a few sentences:

- The Java API contains many packages.
- A package contains classes.
- From a class, you can create objects. Each such object is an *instance* of the class.
- An object has its own copy of each of the class's fields and methods (each of the class's members).
- A class has the one-and-only copy of each of the class's static fields and static methods (the class's static members).

If you care to know, Java groups its packages into even larger units called *modules*. For a quick peek at this concept, see the sidebar figure.



What do you do instead? You write

```
import static java.lang.System.out;
```

To find out more about the out variable's being a static variable, read the next section.

TECHNICAL STUFF

In this chapter, I refer to out as a static variable. That's okay. But a more descriptive way to refer to out is to call it a static field of Java's System class.

Shedding light on the static darkness

I love to quote myself. When I quote my own words, I don't need written permission. I don't have to think about copyright infringement, and I never hear from lawyers. Best of all, I can change and distort anything I say. When I paraphrase my own ideas, I can't be misquoted.

With all that in mind, here's a quote from an earlier section:

> Anything that's static belongs to a whole class, not to any particular instance of the class. [. . .] To call a static method, you use a class's name along with a dot.

How profound! In Listing 14-6, I introduce a static method named parseInt. Here's the same quotation applied to the static parseInt method:

> The static parseInt method belongs to the whole Integer class, not to any particular instance of the Integer class. [. . .] To call the static parseInt method, you use the Integer class's name along with a dot. You write something like Integer.parseInt(reply).

That's very nice! How about the System.out business that I introduce in Chapter 3? I can apply my quotation to that, too.

> The static out variable belongs to the whole System class, not to any particular instance of the System class. [. . .] To refer to the static out variable, you use the System class's name along with a dot. You write something like System.out.println().

If you think about what System.out means, this static business makes sense. After all, the name System.out refers to the place where a Java program sends text output. (When you use IntelliJ IDEA, the name System.out refers to IntelliJ's Run tool window.) A typical program has only one place to send its text output. So a Java program has only one out variable. No matter how many objects you create — three, ten, or none — you have just one out variable. And when you make something static, you ensure that the program has only one of those things.

All right, then! The out variable is static.

To abbreviate the name of a static variable (or a static method), you don't use an ordinary import declaration. Instead, you use a static import declaration. That's why, in Chapter 9 and beyond, I use the word static to import the out variable:

```
import static java.lang.System.out;
```

Barry makes good on an age-old promise

In Chapter 6, I pull a variable declaration outside of a main method. I go from code of the kind in Listing 14-9 to code of the kind that's in Listing 14-10.

LISTING 14-9: Declaring a Variable Inside the main Method

```
public class SnitSoft {

    public static void main(String[] args) {
        double amount = 5.95;

        amount = amount + 25.00;
        System.out.println(amount);
    }
}
```

LISTING 14-10: Pulling a Variable Outside the main Method

```
public class SnitSoft {
    static double amount = 5.95;

    public static void main(String[] args) {
        amount = amount + 25.00;
        System.out.println(amount);
    }
}
```

In Listing 14-9, amount is a variable belonging to the main method. But in Listing 14-10, amount is a static field belonging to the SnitSoft class.

In Chapter 6, I promise to explain why Listing 14-10 needs the extra word static (in static double amount = 5.95). Well, with all the fuss about static methods in this chapter, I can finally explain everything.

Refer to Figure 14-7. In that figure, you have checks, and you have a sort method. Each individual check has its own number, its own amount, and its own payee. But the entire Check class has just one sort method.

I don't know about you, but to sort my canceled checks, I hang them on my exotic *Yucca elephantipes* tree. I fasten the higher-numbered checks to the upper leaves and put the lower-numbered checks on the lower leaves. When I find a check whose number comes between two other checks, I select a free leaf (one that's between the upper and lower leaves).

A program to mimic my sorting method looks something like this:

```
public class Check {
    int number;
    double amount;
    String payee;

    static void sort() {
        Yucca tree;

        if (myCheck.number > 1700) {
            tree.attachHigh(myCheck);
        }
        // ... etc.
    }
}
```

Because of the word static, the Check class has only one sort method. And because I declare the tree variable inside the static sort method, this program has only one tree variable. (Indeed, I hang all my canceled checks on just one yucca tree.) I can move the tree variable's declaration outside of the sort method. But if I do, I may have too many yucca trees:

```
public class Check {
    int number;
    double amount;
    String payee;
    Yucca tree; //This is bad!
                //Each check has its own tree.

    static void sort() {
        if (myCheck.number > 5000) {
            tree.attachHigh(myCheck);
        }
        // ... etc.
    }
}
```

In this nasty code, each check has its own number, its own amount, its own payee, and its own tree. But that's ridiculous! I don't want to fasten each check to its own yucca tree. Everybody knows you're supposed to sort checks with just one yucca tree. (That's the way the big banks do it.)

When I move the `tree` variable's declaration outside of the `sort` method, I want to preserve the fact that I have only one tree. (To be more precise, I have only one tree for the entire `Check` class.) To make sure that I have only one tree, I declare the `tree` variable to be static:

```
public class Check {
    int number;
    double amount;
    String payee;
    static Yucca tree; //That's better!

    static void sort() {
        if (myCheck.number > 5000) {
            tree.attachHigh(myCheck);
        }
        // ... etc.
    }
}
```

For exactly the same reason, I write **`static`** `double amount` when I move from Listing 14-9 to 14-10.

TIP

To find out more about sorting, read *Algorithms For Dummies*, by John Paul Mueller and Luca Massaron. To learn more about bank checks, read *Managing Your Money Online For Dummies*, by Kathleen Sindell. To learn more about trees, read *Landscaping For Dummies*, by Phillip Giroux, Bob Beckstrom, and Lance Walheim (all published by Wiley).

TRY IT OUT

These experiments will help you understand static methods and static fields.

MORE MONEY

Run the following code and identify the static methods that are called in the code:

```
import java.text.NumberFormat;

import javax.swing.JOptionPane;

public class Main {

  public static void main(String[] args) {
    var currency = NumberFormat.getCurrencyInstance();
    String inputString = JOptionPane.showInputDialog("Enter an amount");
    double inputAmount = Double.parseDouble(inputString);
    double oneMore = inputAmount + 1;
    String oneMoreMoney = currency.format(oneMore);
    String message = "One more than that amount is " + oneMoreMoney + ".";
    JOptionPane.showMessageDialog(null, message);
  }
}
```

Hint: Most Java programmers begin the names of classes with capital letters. Any name that starts with a capital letter is probably the name of a class. If you're unsure about a particular name, you can look up that name in Java's API documentation. The documentation is online at `https://docs.oracle.com/en/java/javase/17/docs/api/index.html`.

BOOKS FOR DUMMIES

Have a gander at the following code. How many copies of the `author` field exist during a run of this code? How many copies of the `publisher` field exist during a run of the code? Why?

```
public class Book {
    String title;
    String author;
    static String publisher = "Wiley";
}

public class Main {

    public static void main(String[] args) {
        var javaForDummies = new Book();
        javaForDummies.title = "Java For Dummies";
        javaForDummies.author = "Barry Burd";

        var dosForDummies = new Book();
        dosForDummies.title = "DOS For Dummies";
        dosForDummies.author = "Dan Gookin";
    }
}
```

Remember that `public class Book` code must be in a file named `Book.java`, and the `public class Main` code must be in a different file — a file named `Main.java`.

Would the number of copies of the `publisher` field change if you added these two statements to the `main` method?

```
Book.publisher = "John Wiley & Sons, Inc.";
Book.publisher = "A publisher in the United States";
```

If so, why? If not, why not?

STATIC AND NONSTATIC FIELDS

Run the following code. Then examine the code to determine why it produces the output you see in the Run tool window:

```
public class IntegerHolder {
  int value;
  static int howMany = 0;
}

public class Main {

  public static void main(String[] args) {
    var holder1 = new IntegerHolder();
    holder1.value = 79;
    IntegerHolder.howMany++;

    var holder2 = new IntegerHolder();
    holder2.value = 443;
    IntegerHolder.howMany++;

    System.out.println(holder1.value);
    System.out.println(holder2.value);
    System.out.println(IntegerHolder.howMany);

    //System.out.println(IntegerHolder.value); Why is this statement illegal?

    System.out.println(holder1.howMany);      // This statement is legal
                                              // but the statement is
                                              // misleading. Why?
  }
}
```

SOMETHING'S WRONG HERE

What's wrong with the following code? How does the meaning of the word `static` make a difference? Would adding another `static` keyword fix the problem? Why?

```
import java.util.Scanner;

public class Main {
    Scanner keyboard = new Scanner(System.in);

    public static void main(String[] args) {
        int numberOfCats = keyboard.nextInt();
        System.out.println(numberOfCats);
    }
}
```

Remember that, in this example, you can't use `var keyboard` in the declaration of the `Scanner`. You can use `var` when you initialize a variable inside a method, but you can't use `var` when you declare a field. This prohibition applies to fields with initialization and fields without initialization. For details, refer to Chapter 13.

JUST THE FACTS

In the following code, why are the fields static?

```
public class Facts {
    static int numberOfPlanets = 8;
    static int numberOfMoons = 1;
    static int numberOfContinents = 7;
    static int numberOfOceans = 5;
}
```

You can improve on the `Facts` class even better by adding Java's `final` keyword to each field declaration:

```
public class BetterFacts {
    static final int numberOfPlanets = 8;
    static final int numberOfMoons = 1;
    static final int numberOfContinents = 7;
    static final int numberOfOceans = 5;
}
```

A *final* variable's value can't be changed. So, in a `main` method, the code

```
System.out.println(BetterFacts.numberOfPlanets);
```

is legal, but the code

```
BetterFacts.numberOfPlanets = 9; // Bad code
```

is not legal.

IN THIS CHAPTER

- Writing methods that work with existing values
- Building methods that modify existing values
- Making methods that return new values

Chapter 15

Creating New Java Methods

In Chapters 4 and 5, I introduce Java methods. I show you how to create a `main` method, how to call the `System.out.println` method, and how to use the `Scanner` class's `nextLine` method. Between Chapters 5 and 14, I make very little noise about methods. In Chapter 14, I introduce a bunch of new methods for you to call, but that's only half the story.

This chapter completes the circle. In this chapter, you create your own Java methods — not the tired old `main` method that you've been using all along, but rather some new, powerful Java methods.

Defining a Method within a Class

In Chapter 14, Figure 14-6 introduces an interesting notion — a notion that's at the core of object-oriented programming. Each Java string has its own `equals` method. That is, each string has, built within it, the functionality to compare itself to other strings. That's an important point. When you do object-oriented programming, you bundle data and functionality into a lump called a class. Just remember Barry's immortal words from Chapter 13:

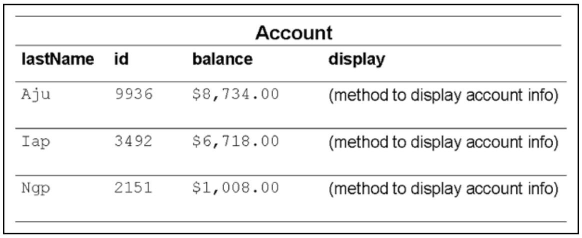
> A class . . . describes the way in which you intend to combine *and use* pieces of data.

And why are these words so important? They're important because, in object-oriented programming, chunks of data take responsibility for themselves. With object-oriented programming, everything you have to know about a string is located in the file `String.java`. So, if people have problems with the strings, they know just where to look for all the code. That's great!

This is the deal: Objects contain methods. Chapter 14 shows you how to use an object's methods, and this chapter shows you how to *create* an object's methods.

Making a method

Imagine a table containing the information about three accounts. (If you have trouble imagining such a thing, just look at Figure 15-1.) In the figure, each account has a last name, an identification number, and a balance. In addition (and here's the important part), each account knows how to display itself on the screen. Each row of the table has its own copy of a `display` method.



Account			
lastName	id	balance	display
Aju	9936	$8,734.00	(method to display account info)
Iap	3492	$6,718.00	(method to display account info)
Ngp	2151	$1,008.00	(method to display account info)

FIGURE 15-1: A table of accounts.

The last names in Figure 15-1 may seem strange to you. That's because I generated the table's data randomly. Each last name is a haphazard combination of three letters: one uppercase letter followed by two lowercase letters.

TECHNICAL STUFF

Though it may seem strange, generating account values at random is common practice. When you write new code, you want to test the code to find out whether it runs correctly. You can make up your own data (with values like `"Smith"`, `0000`, and `1000.00`). But to give your code a challenging workout, you should use some unexpected values. If you have values from some real-life case studies, you should use them. But if you don't have real data, randomly generated values are easy to create.

To find out how I randomly generate three-letter names, see this chapter's later sidebar, "Generating words randomly."

I need some code to implement the ideas in Figure 15-1. Fortunately, I have some code in Listing 15-1.

LISTING 15-1: An Account Class

```
import java.text.NumberFormat;

import static java.lang.System.out;

public class Account {
    String lastName;
    int id;
    double balance;

    void display() {
        var currency = NumberFormat.getCurrencyInstance();

        out.print("The account with last name ");
        out.print(lastName);
        out.print(" and ID number ");
        out.print(id);
        out.print(" has balance ");
        out.println(currency.format(balance));
    }
}
```

The Account class in Listing 15-1 defines four members: a lastName field, an id field, a balance field, and a display method. So each instance of Account class has its own lastName, its own id, its own balance, and its own way of doing display. These things match up with the four columns in Figure 15-1.

REMEMBER

You can't use var in the declaration of a field. This rule holds true even if you give the field an initial value. In Listing 15-1, the following declaration would be legal:

```
String lastName = "(UNKNOWN)";
```

But the use of var would be illegal:

```
// Java doesn't like this at all:
var lastName = "(UNKNOWN)";
```

Examining the method's header

Listing 15-1 contains the display method's declaration. Like a main method's declaration, the display declaration has a header and a body. (See Chapter 4.) The header has two words and some parentheses:

- **The word void tells the computer that, when the display method is called, the display method doesn't return anything to the place that called it.**

  Later in this chapter, a method does return something. For now, the display method returns nothing.

- **The word display is the method's name.**

  Every method must have a name. Otherwise, you don't have a way to call the method.

- **The parentheses contain all the things you're going to pass to the method when you call it.**

  When you call a method, you can pass information to that method on the fly. This display example, with its empty parentheses, looks strange. That's because no information is passed to the display method when you call it. That's okay. I give a meatier example later in this chapter.

Examining the method's body

The display method's body contains some print and println calls. The interesting thing here is that the body makes reference to the lastName, id, and balance fields. A method's body can do that. But with each object having its own lastName, id, and balance variables, what does a variable in the display method's body mean?

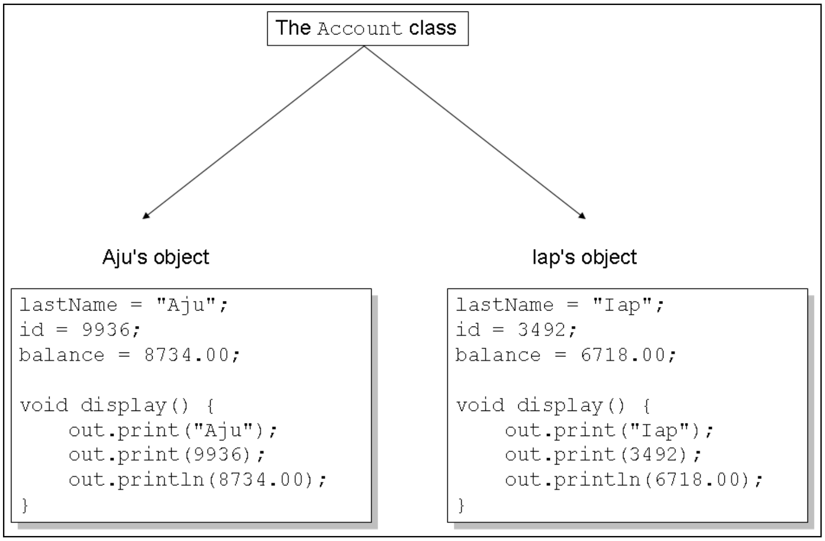
Well, when I use the Account class, I create little account objects. Maybe I create an object for each row of the table in Figure 15-1. Each object has its own values for the lastName, id, and balance variables, and each object has its own copy of the display method.

Take the first display method in Figure 15-1 — the method for Aju's account. The display method for that object behaves as though it had the code in Listing 15-2.

LISTING 15-2: How the `display` Method Behaves When No One's Looking

```
/*
* This is not real code:
*/
void display() {
    var currency = NumberFormat.getCurrencyInstance();


    out.print("The account with last name ");
    out.print("Aju");
    out.print(" and ID number ");
    out.print(9936);
    out.print(" has balance ");
    out.println(currency.format(8734.00));
}
```

In fact, each of the three `display` methods behaves as though its body has a slightly different code. Figure 15-2 illustrates this idea for two instances of the `Account` class.



```
lastName = "Aju";
id = 9936;
balance = 8734.00;

void display() {
    out.print("Aju");
    out.print(9936);
    out.println(8734.00);
}
```

```
lastName = "Iap";
id = 3492;
balance = 6718.00;

void display() {
    out.print("Iap");
    out.print(3492);
    out.println(6718.00);
}
```

FIGURE 15-2: Two objects, each with its own `display` method.

Calling the method

To put the previous section's ideas into action, you need more code. So the next listing (see Listing 15-3) creates instances of the `Account` class.

LISTING 15-3: Making Use of the Code in Listing 15-1

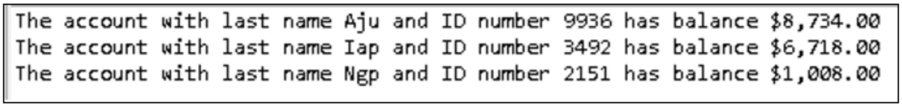
```
import java.util.Random;

public class ProcessAccounts {

    public static void main(String[] args) {

        var myRandom = new Random();
        Account anAccount;

        for (int i = 0; i < 3; i++) {
            anAccount = new Account();

            anAccount.lastName = "" +
                    (char) (myRandom.nextInt(26) + 'A') +
                    (char) (myRandom.nextInt(26) + 'a') +
                    (char) (myRandom.nextInt(26) + 'a');

            anAccount.id = myRandom.nextInt(10000);
            anAccount.balance = myRandom.nextInt(10000);
            anAccount.display();
        }
    }
}
```

Here's a summary of the action in Listing 15-3:

```
Do the following three times:
    Create a new object (an instance of the Account class).
    Randomly generate values for the object's lastName, id and balance.
    Call the object's display method.
```

The first of the three `display` calls prints the first object's `lastName`, `id`, and `balance` values. The second `display` call prints the second object's `lastName`, `id`, and `balance` values. And so on.

A run of the code from Listing 15-3 is shown in Figure 15-3.



```
The account with last name Aju and ID number 9936 has balance $8,734.00
The account with last name Iap and ID number 3492 has balance $6,718.00
The account with last name Ngp and ID number 2151 has balance $1,008.00
```

FIGURE 15-3: Running the code in Listing 15-3.

WARNING

Concerning the code in Listing 15-3, your mileage may vary. You don't see the same values as the ones in Figure 15-3. In fact, if you run Listing 15-3 more than once, you (almost certainly) get different three-letter names, different ID numbers, and different account balances each time. That's what happens when a program generates values randomly.

Following the flow

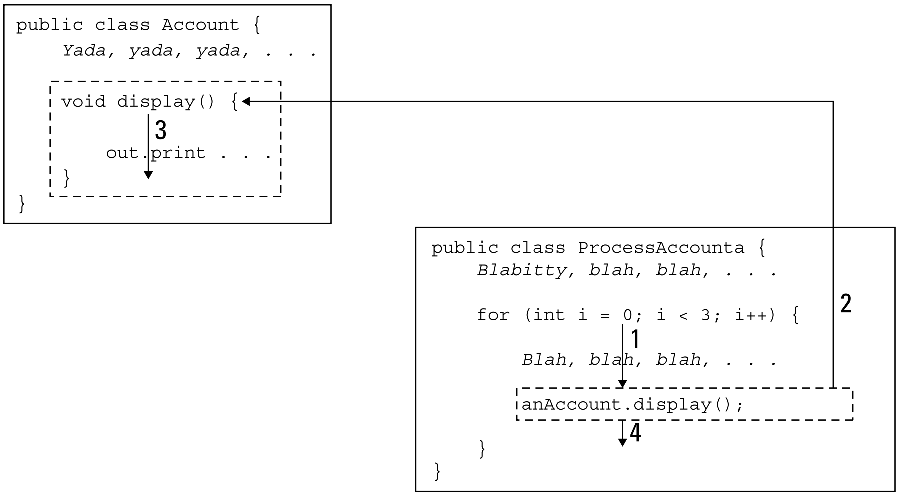
Suppose that you're running the code in Listing 15-3. The computer reaches the `display` method call:

```
anAccount.display();
```

At that point, the computer starts running the code inside the `display` method. In other words, the computer jumps to the middle of the `Account` class's code (the code in Listing 15-1).

After executing the `display` method's code (that forest of `print` and `println` calls), the computer returns to the point where it departed from in Listing 15-3. That is, the computer goes back to the `display` method call and continues on from there.

When you run the code in Listing 15-3, the flow of action in each loop iteration isn't exactly from the top to the bottom. Instead, the action goes from the `for` loop to the `display` method and then back to the `for` loop. The whole business is pictured in Figure 15-4.



```
public class Account {
     Yada, yada, yada, . . .

     void display() {

          out.print . . .
     }
}
```

```
public class ProcessAccounta {
     Blabitty, blah, blah, . . .

     for (int i = 0; i < 3; i++) {

          Blah, blah, blah, . . .

          anAccount.display();

     }
}
```

FIGURE 15-4: The flow of control between Listings 15-1 and 15-3.

Using punctuation

In Listing 15-3, notice the use of dots. To refer to the `lastName` stored in the `anAccount` object, you write

```
anAccount.lastName
```

To get the `anAccount` object to `display` itself, you write

```
anAccount.display();
```

That's great! When you refer to an object's field or call an object's method, the only difference is parentheses:

- To refer to an object's field, you don't use parentheses.
- To call an object's method, you use parentheses.

REMEMBER

When you call a method, you put parentheses after the method's name. You do this even if you have nothing to put inside the parentheses.

Combining characters

In Listing 15-3, the statement

```
anAccount.lastName = "" +
         (char) (myRandom.nextInt(26) + 'A') +
         (char) (myRandom.nextInt(26) + 'a') +
         (char) (myRandom.nextInt(26) + 'a');
```

has many plus signs, and each plus sign concatenates things together. The first thing is an empty string (`""`). This empty string contains no characters, so it's invisible. It doesn't get in the way of your seeing the second, third, and fourth things. The later "Generating words randomly" sidebar reveals the mysterious purpose of that invisible empty string.

Onto the empty string, the program concatenates a second thing. This second thing is the value of the expression `(char) (myRandom.nextInt(26) + 'A')`. The expression may look complicated, but it's really no big deal. This expression represents an uppercase letter (any uppercase letter, generated randomly). Once again, the later "Generating words randomly" sidebar tells a more complete story.

Onto the empty string and the uppercase letter, the program concatenates a third thing. This third thing is the value of the expression `(char) (myRandom.nextInt(26) + 'a')`. This expression represents a lowercase letter (any lowercase letter, generated randomly).

Onto all this stuff, the program concatenates another lowercase letter. So altogether you have a randomly generated three-letter name.

TECHNICAL STUFF

In Listing 15-3, the statement `anAccount.balance = myRandom.nextInt (10000)` assigns an `int` value to `balance`. But `balance` is a `double` variable, not an `int` variable. That's okay. In a rare case of permissiveness, Java allows you to assign an `int` value to a `double` variable. The result of the assignment is no big surprise. If you assign the `int` value 8734 to the `double` variable `balance`, the value of `balance` becomes 8734.00. The result is shown on the first line of Figure 15-3.

REMEMBER

Using the `double` type to store an amount of money is generally a bad idea. In this book, I use `double` to keep the examples as simple as possible. But the `int` type is better for money values, and the `BigDecimal` type is even better. For more details, see Chapter 7.

GENERATING WORDS RANDOMLY

Most programs don't work correctly the first time you run them, and some programs don't work without extensive trial-and-error. This section's code is a case in point.

To write this section's code, I needed a way to generate three-letter words randomly. After about a dozen attempts, I got the code to work. But I didn't stop there. I kept working for a few hours looking for a *simple* way to generate three-letter words randomly. In the end, I settled on the following code (in Listing 15-3):

```
anAccount.lastName = "" +
        (char) (myRandom.nextInt(26) + 'A') +
        (char) (myRandom.nextInt(26) + 'a') +
        (char) (myRandom.nextInt(26) + 'a');
```

This code isn't simple, but it's not nearly as bad as my original working version. Anyway, here's how the code works:

- **Each call to `myRandom.nextInt(26)` generates a number from 0 to 25.**
- **Adding `'A'` gives you a number from 65 to 90.**

  To store a letter `'A'`, the computer puts the number 65 in its memory. That's why adding `'A'` to 0 gives you 65 and why adding `'A'` to 25 gives you 90. (For more

(continued)

(continued)

information on letters being stored as numbers, see the discussion of Unicode characters at the end of Chapter 8.)

- **Applying `(char)` to a number turns the number into a `char` value.**

  To store the letters `'A'` through `'Z'`, the computer puts the numbers 65 through 90 in its memory. So, applying `(char)` to a number from 65 to 90 turns the number into an uppercase letter. For more information about applying things like `(char)`, see the discussion of casting in a sidebar in Chapter 7.

Pause for a brief summary. The expression `(char) (myRandom.nextInt(26) + 'A')` represents a randomly generated uppercase letter. In a similar way, `(char) (myRandom.nextInt(26) + 'a')` represents a randomly generated lowercase letter.

Watch out! The next couple of steps can be tricky:

- **Java doesn't allow you to assign a `char` value to a string variable.**

  In Listing 15-3, the following statement would lead to a compiler error:

  ```
  //Bad statement:
  anAccount.lastName = (char) (myRandom.nextInt(26) + 'A');
  ```

- **In Java, you can use a plus sign to add a `char` value to a string. When you do, the result is a string:**

  So, `"" + (char) (myRandom.nextInt(26) + 'A')` is a string containing one randomly generated uppercase character. And when you add `(char) (myRandom.nextInt(26) + 'a')` to the end of that string, you get another string — a string containing two randomly generated characters. Finally, when you add another `(char) (myRandom.nextInt(26) + 'a')` to the end of that string, you get a string containing three randomly generated characters. So you can assign that big string to `anAccount.lastName`. That's how the statement in Listing 15-3 works.

When you write a program like the one in Listing 15-3, you have to be careful with numbers, `char` values, and strings. I don't do this kind of programming every day of the week — before I got this section's example to work, I had many false starts. That's okay. I'm persistent.

Let the Objects Do the Work

When I was a young object, I wasn't as smart as the objects you have nowadays. Consider, for example, the object in Listing 15-4. This object not only displays itself, but it can also fill itself with values.

LISTING 15-4: A Class with Two Methods

```
import java.util.Random;
import java.text.NumberFormat;

import static java.lang.System.out;

public class BetterAccount {
    String lastName;
    int id;
    double balance;

    void fillWithData() {
        var myRandom = new Random();

        lastName = "" +
                (char) (myRandom.nextInt(26) + 'A') +
                (char) (myRandom.nextInt(26) + 'a') +
                (char) (myRandom.nextInt(26) + 'a');

        id = myRandom.nextInt(10000);
        balance = myRandom.nextInt(10000);
    }

    void display() {
        var currency = NumberFormat.getCurrencyInstance();

        out.print("The account with last name ");
        out.print(lastName);
        out.print(" and ID number ");
        out.print(id);
        out.print(" has balance ");
        out.println(currency.format(balance));
    }
}
```

I wrote some code to use the class in Listing 15-4. This new code is in Listing 15-5.

LISTING 15-5: This Is So Cool!

```
public class ProcessBetterAccounts {

    public static void main(String[] args) {

        BetterAccount anAccount;

        for (int i = 0; i < 3; i++) {
            anAccount = new BetterAccount();
            anAccount.fillWithData();
            anAccount.display();
        }
    }
}
```

Listing 15-5 is pretty slick. Because the code in Listing 15-4 is so darn smart, the new code in Listing 15-5 has very little work to do. This new code just creates a BetterAccount object and then calls the methods in Listing 15-4. When you run all this stuff, you get results like the ones in Figure 15-3.

Passing the Buck

Think about sending someone to the supermarket to buy bread. When you do this, you say, "Go to the supermarket and buy some bread." (Try it at home. You'll have a fresh loaf of bread in no time at all!) Of course, some other time, you send that same person to the supermarket to buy bananas. You say, "Go to the supermarket and buy some bananas." And what's the point of all of this? Well, you have a method, and you have some on-the-fly information that you pass to the method when you call it. The method is named "Go to the supermarket and buy some . . ." The on-the-fly information is either "bread" or "bananas," depending on your culinary needs. In Java, the method calls would look like this:

```
goToTheSupermarketAndBuySome(bread);
goToTheSupermarketAndBuySome(bananas);
```

The things in parentheses are called *parameters* or *parameter lists.* With parameters, your methods become much more versatile. Rather than get the same thing each time, you can send somebody to the supermarket to buy bread one time, bananas another time, and birdseed the third time. When you call your goToTheSupermarketAndBuySome method, you decide right there and then what you're going to ask your pal to buy.

These concepts are made more concrete in Listings 15-6 and 15-7.

LISTING 15-6: Adding Interest

```
import java.text.NumberFormat;

import static java.lang.System.out;

public class NiceAccount {
    String lastName;
    int id;
    double balance;

    void addInterest(double rate) {
        out.print("Adding ");
        out.print(rate);
        out.println(" percent...");

        balance += balance * (rate / 100.0);
    }

    void display() {

        var currency = NumberFormat.getCurrencyInstance();

        out.print("The account with last name ");
        out.print(lastName);
        out.print(" and ID number ");
        out.print(id);
        out.print(" has balance ");
        out.println(currency.format(balance));
    }
}
```

LISTING 15-7: Calling the `addInterest` Method

```
import java.util.Random;

public class ProcessNiceAccounts {

    public static void main(String[] args) {
        var myRandom = new Random();
        NiceAccount anAccount;
        double interestRate;
```

(continued)

LISTING 15-7: ***(continued)***

```
        for (int i = 0; i < 3; i++) {
            anAccount = new NiceAccount();

            anAccount.lastName = "" +
                    (char) (myRandom.nextInt(26) + 'A') +
                    (char) (myRandom.nextInt(26) + 'a') +
                    (char) (myRandom.nextInt(26) + 'a');
            anAccount.id = myRandom.nextInt(10000);
            anAccount.balance = myRandom.nextInt(10000);

            anAccount.display();

            interestRate = myRandom.nextInt(5);
            anAccount.addInterest(interestRate);

            anAccount.display();
            System.out.println();
        }
    }
}
```

In Listing 15-7, the line

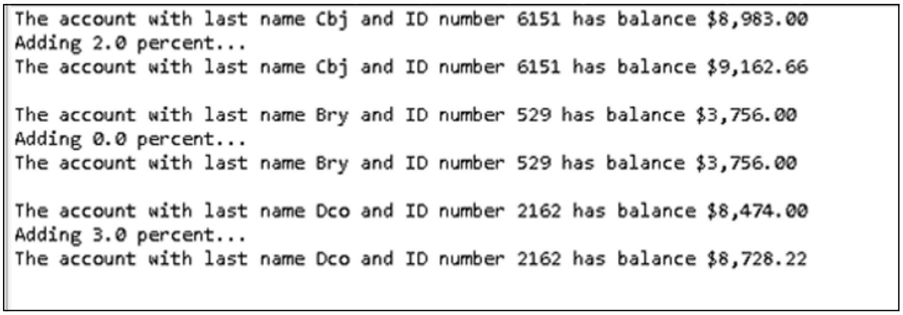
```
anAccount.addInterest(interestRate);
```

plays the same role as the line `goToTheSupermarketAndBuySome(bread)` in my little supermarket example. The word `addInterest` is a method name, and the word `interestRate` in parentheses is a parameter. Taken as a whole, this statement tells the code in Listing 15-6 to execute its `addInterest` method. This statement also tells Listing 15-6 to use a certain number (whatever value is stored in the `interestRate` variable) in the method's calculations. The value of `interestRate` can be 1.0, 2.0, or whatever other value you get by calling `myRandom.nextInt(5)`. In the same way, the `goToTheSupermarketAndBuySome` method works for bread, bananas, or whatever else you need from the market.

The next section has a detailed description of `addInterest` and its action. In the meantime, a run of the code in Listings 15-6 and 15-7 is shown in Figure 15-5.

REMEMBER

Java has strict rules about the use of types. For example, you can't assign a `double` value (like `3.14`) to an `int` variable. (The compiler simply refuses to chop off the `.14` part. You get an error message. So what else is new?) But Java isn't completely unreasonable about the use of types. Java allows you to assign an `int` value (like `myRandom.nextInt(5)`) to a `double` variable (like `interestRate`). If you assign the `int` value `2` to the `double` variable `interestRate`, then the value of `interestRate` becomes `2.0`. The result is shown on the second line of Figure 15-5.



```
The account with last name Cbj and ID number 6151 has balance $8,983.00
Adding 2.0 percent...
The account with last name Cbj and ID number 6151 has balance $9,162.66

The account with last name Bry and ID number 529 has balance $3,756.00
Adding 0.0 percent...
The account with last name Bry and ID number 529 has balance $3,756.00

The account with last name Dco and ID number 2162 has balance $8,474.00
Adding 3.0 percent...
The account with last name Dco and ID number 2162 has balance $8,728.22
```

FIGURE 15-5: Running the code in Listing 15-7.

Handing off a value

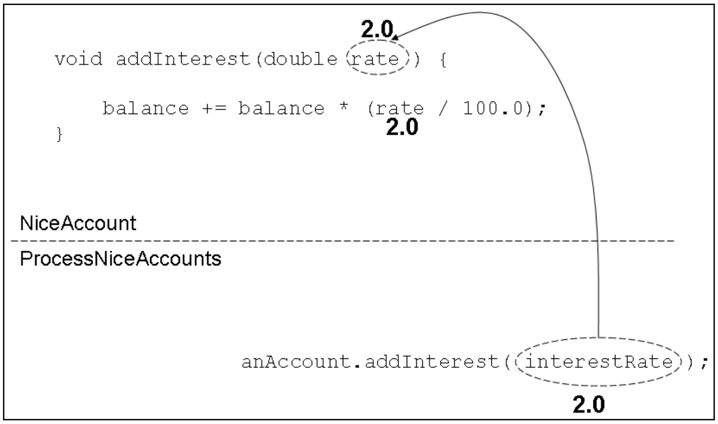
When you call a method, you can pass information to that method on the fly. This information is in the method's parameter list. Listing 15-7 has a call to the `addInterest` method:

```
anAccount.addInterest(interestRate);
```

The first time through the loop, the value of `interestRate` is `2.0`. (Remember, I'm using the data in Figure 15-5.) At that point in the program's run, the method call behaves as though it's the following statement:

```
anAccount.addInterest(2.0);
```

The computer is about to run the code inside the `addInterest` method (a method in Listing 15-6). But first, the computer *passes* the value `2.0` to the parameter in the `addInterest` method's header. Inside the `addInterest` method, the value of `rate` becomes `2.0`. For an illustration of this idea, see Figure 15-6.



```
void addInterest(double rate) {

    balance += balance * (rate / 100.0);
}
```

```
anAccount.addInterest(interestRate);
```

FIGURE 15-6: Passing a value to a method's parameter.

Here's something interesting. The parameter in the addInterest method's header is rate. But, inside the ProcessNiceAccounts class, the parameter in the method call is interestRate. That's okay. In fact, it's standard practice.

In Listings 15-6 and 15-7, the names of the parameters don't have to be the same. The only thing that matters is that both parameters (rate and interestRate) have the same type. In Listings 15-6 and 15-7, both of these parameters are of type double. So everything is fine.

Inside the addInterest method, the += assignment operator adds balance * (rate / 100.0) to the existing balance value. For some info about the += assignment operator, see Chapter 7.

Working with a method header

In the next few bullets, I make some observations about the addInterest method header (in Listing 15-6):

- **The word void tells the computer that when the addInterest method is called, the addInterest method doesn't send a value back to the place that called it.**

  The next section has an example in which a method sends a value back.

- **The word addInterest is the method's name.**

  That's the name you use to call the method when you're writing the code for the ProcessNiceAccounts class. (See Listing 15-7.)

- **The parentheses in the header contain placeholders for all the things you're going to pass to the method when you call it.**

  When you call a method, you can pass information to that method on the fly. This information is the method's parameter list. The addInterest method's header says that the addInterest method takes one piece of information, and that piece of information must be of type double:

  ```
  voidaddInterest(double rate)
  ```

  Sure enough, if you look at the call to addInterest (down in the ProcessNiceAccounts class's main method), that call has the variable interestRate in it. And interestRate is of type double. When I call addInterest, I'm giving the method a value of type double.

Using each object's field values

The addInterest method in Listing 15-6 is called three times from the main method in Listing 15-7. The actual account balances and interest rates are different each time:

- **In the first call of Figure 15-5, the balance is 8983.00, and the interest rate is 2.0.**

  When this call is made, the expression balance * (rate / 100.0) stands for 8983.00 * (2.0 / 100.00). See Figure 15-7.

- **In the second call of Figure 15-5, the balance is 3756.00, and the interest rate is 0.0.**

  When the call is made, the expression balance * (rate / 100.0) stands for 3756.00 * (0.0 / 100.00). Again, see Figure 15-7.

- **In the third call of Figure 15-5, the balance is 8474.00, and the interest rate is 3.0.**

  When the addInterest call is made, the expression balance * (rate / 100.0) stands for 8474.00 * (3.0 / 100.00).

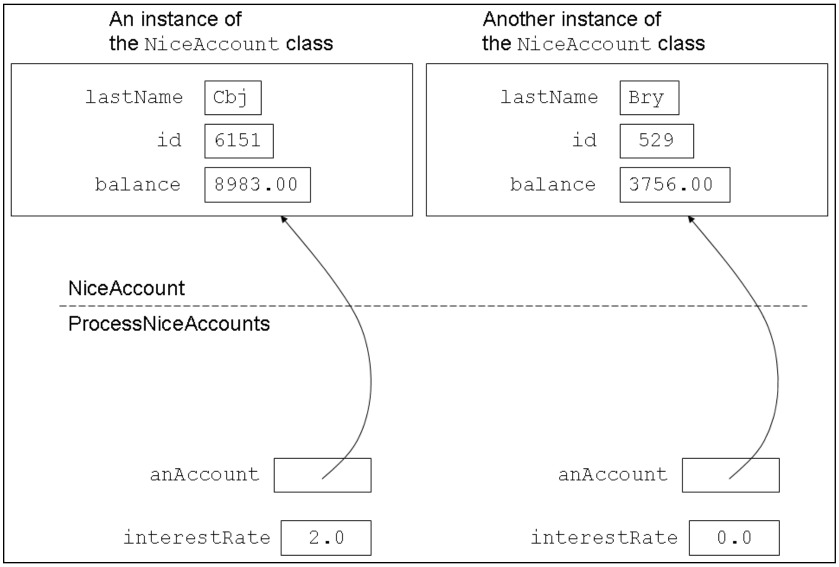


FIGURE 15-7: Cbj's account and Bry's account.

Passing more than one parameter

Take a look at Listings 15-6 and 15-7. In those listings, the `display` method has no parameters and the `addInterest` method has one parameter. Now consider the following code from Chapter 12:

```
char reply;
...
reply = keyboard.findWithinHorizon(".", 0).charAt(0);
```

In that code, the `findWithinHorizon` method call has two parameters: the `String` parameter `"."` and the `int` parameter `0`. That's not unusual. You can create methods with as many parameters as you like. The only restriction is this: When you call a method, the types of the parameters in the call must match up with the types of parameters in the method declaration's header. For example, in Java's API code, the first line of the `findWithinHorizon` method looks like this:

```
public String findWithinHorizon(String pattern, int horizon) {
```

And, in the method call `findWithinHorizon(".", 0)`, the first parameter `"."` is a `String`, and the second parameter `0` is an `int`.

Listings 15-8 and 15-9 are variations on the code in Listings 15-6 and 15-7. In the new listings, the `addInterest` method has two parameters: one for the interest rate and another for a number of years. When you call the `addInterest` method, the method repeatedly adds interest for the number of years you've specified.

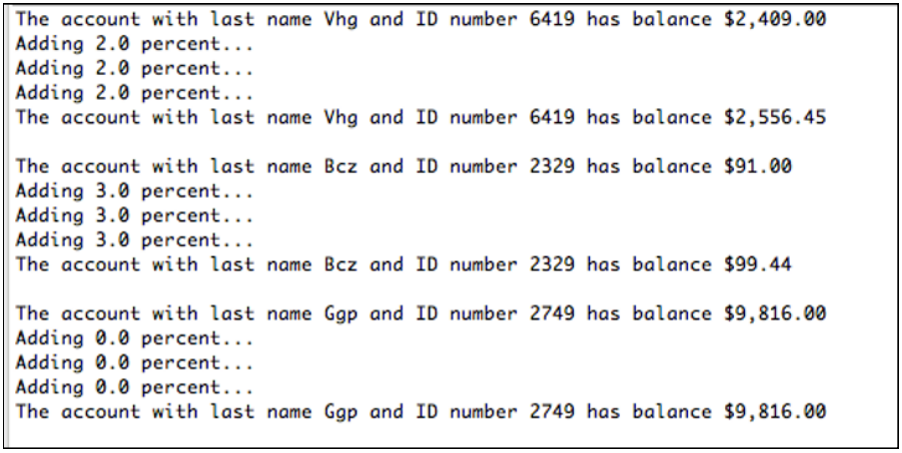
A run of the code in Listings 15-8 and 15-9 is shown in Figure 15-8.

LISTING 15-8: Adding Interest for a Certain Number of Years

```
import java.text.NumberFormat;

import static java.lang.System.out;

public class NiceAccount {
    String lastName;
    int id;
    double balance;

    void addInterest(double rate, int howManyYears) {
        for (int i = 1; i <= howManyYears; i++) {
            out.print("Adding ");
            out.print(rate);
            out.println(" percent...");
            balance += balance * (rate / 100.0);
        }
    }

    void display() {
        var currency = NumberFormat.getCurrencyInstance();

        out.print("The account with last name ");
        out.print(lastName);
        out.print(" and ID number ");
        out.print(id);
        out.print(" has balance ");
        out.println(currency.format(balance));
    }
}
```

LISTING 15-9: Calling the Beefed-Up addInterest Method

```
import java.util.Random;

public class ProcessNiceAccounts {

    public static void main(String[] args) {
        var myRandom = new Random();
        NiceAccount anAccount;
        double interestRate;

        for (int i = 0; i < 3; i++) {
            anAccount = new NiceAccount();

            anAccount.lastName = "" +
                    (char) (myRandom.nextInt(26) + 'A') +
                    (char) (myRandom.nextInt(26) + 'a') +
                    (char) (myRandom.nextInt(26) + 'a');
            anAccount.id = myRandom.nextInt(10000);
            anAccount.balance = myRandom.nextInt(10000);

            anAccount.display();

            interestRate = myRandom.nextInt(5);
            anAccount.addInterest(interestRate, 3);

            anAccount.display();
            System.out.println();
        }
    }
}
```



```
The account with last name Vhg and ID number 6419 has balance $2,409.00
Adding 2.0 percent...
Adding 2.0 percent...
Adding 2.0 percent...
The account with last name Vhg and ID number 6419 has balance $2,556.45

The account with last name Bcz and ID number 2329 has balance $91.00
Adding 3.0 percent...
Adding 3.0 percent...
Adding 3.0 percent...
The account with last name Bcz and ID number 2329 has balance $99.44

The account with last name Ggp and ID number 2749 has balance $9,816.00
Adding 0.0 percent...
Adding 0.0 percent...
Adding 0.0 percent...
The account with last name Ggp and ID number 2749 has balance $9,816.00
```

FIGURE 15-8: Running the code in Listing 15-9.

Getting a Value from a Method

Say that you're sending a friend to buy groceries. You make requests for groceries in the form of method calls. You issue calls such as

```
goToTheSupermarketAndBuySome(bread);
goToTheSupermarketAndBuySome(bananas);
```

The things in parentheses are parameters. Each time you call your `goToTheSupermarketAndBuySome` method, you put a different value in the method's parameter list.

Now, what happens when your friend returns from the supermarket? "Here's the bread you asked me to buy," says your friend. As a result of carrying out your wishes, your friend returns something to you. You made a method call, and the method returns information (or better yet, the method returns some food).

The thing returned to you is called the method's *return value,* and the type of thing returned to you is called the method's *return type.*

Return on an investment

To see how return values and return types work in a real Java program, check out the code in Listings 15-10 and 15-11.

LISTING 15-10: A Method That Returns a Value

```
import java.text.NumberFormat;

import static java.lang.System.out;

public class GoodAccount {
    String lastName;
    int id;
    double balance;

    double getInterest(double rate) {
        double interest;

        out.print("Adding ");
        out.print(rate);
        out.println(" percent...");

        interest = balance * (rate / 100.0);
        return interest;
    }

    void display() {
        var currency = NumberFormat.getCurrencyInstance();

        out.print("The account with last name ");
        out.print(lastName);
        out.print(" and ID number ");
        out.print(id);
        out.print(" has balance ");
        out.println(currency.format(balance));
    }
}
```

LISTING 15-11: Calling the Method in Listing 15-10

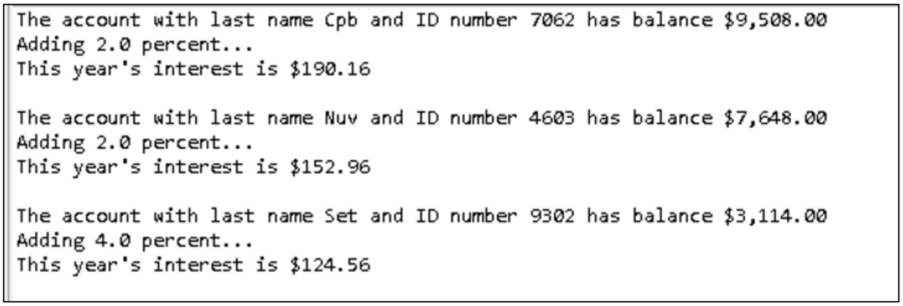
```
import java.util.Random;
import java.text.NumberFormat;

public class ProcessGoodAccounts {

    public static void main(String[] args) {
        var myRandom = new Random();
        var currency = NumberFormat.getCurrencyInstance();
        GoodAccount anAccount;
        double interestRate;
```

(continued)

LISTING 15-11: ***(continued)***

```
        double yearlyInterest;

        for (int i = 0; i < 3; i++) {
            anAccount = new GoodAccount();

            anAccount.lastName = "" +
                    (char) (myRandom.nextInt(26) + 'A') +
                    (char) (myRandom.nextInt(26) + 'a') +
                    (char) (myRandom.nextInt(26) + 'a');
            anAccount.id = myRandom.nextInt(10000);
            anAccount.balance = myRandom.nextInt(10000);

            anAccount.display();

            interestRate = myRandom.nextInt(5);
            yearlyInterest = anAccount.getInterest(interestRate);

            System.out.print("This year's interest is ");
            System.out.println(currency.format(yearlyInterest));
            System.out.println();
        }
    }
}
```

To see a run of code from Listings 15-10 and 15-11, take a look at Figure 15-9.



```
The account with last name Cpb and ID number 7062 has balance $9,508.00
Adding 2.0 percent...
This year's interest is $190.16

The account with last name Nuv and ID number 4603 has balance $7,648.00
Adding 2.0 percent...
This year's interest is $152.96

The account with last name Set and ID number 9302 has balance $3,114.00
Adding 4.0 percent...
This year's interest is $124.56
```

FIGURE 15-9: Running the code in Listing 15-11.

How return types and return values work

I want to trace a piece of the action in Listings 15-10 and 15-11. For input data, I use the first set of values in Figure 15-9.

Here's what happens when getInterest is called (you can follow along in Figure 15-10):

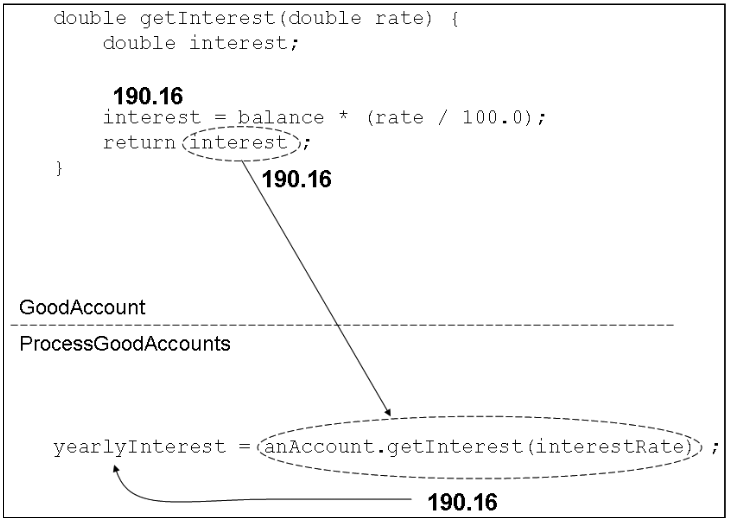
- The value of balance is 9508.00, and the value of rate is 2.0. So the value of balance * (rate / 100.0) is 190.16 — one hundred ninety dollars and sixteen cents.
- The value 190.16 gets assigned to the interest variable, so the statement

  ```
  return interest;
  ```

- has the same effect as

  ```
  return 190.16;
  ```

- The return statement sends this value 190.16 back to the code that called the method. *At that point in the process, the entire method call in Listing 15-11 —* anAccount.getInterest(interestRate) *— takes on the* value 190.16.
- Finally, the value 190.16 gets assigned to the variable yearlyInterest.



```
double getInterest(double rate) {
    double interest;

    interest = balance * (rate / 100.0);
    return interest;
}
```

```
yearlyInterest = anAccount.getInterest(interestRate);
```

FIGURE 15-10: A method call is an expression with a value.

REMEMBER

If a method returns anything, a call to the method is an expression with a value. That value can be printed, assigned to a variable, added to something else, or whatever. Anything you can do with any other kind of value, you can do with a method call.

Working with the method header (again)

When you create a method or a method call, you have to be careful to use Java's types consistently. Make sure that you check for the following:

- In Listing 15-10, the `getInterest` method's header starts with the word `double`. When the method is executed, it should send a `double` value back to the place that called it.
- Again in Listing 15-10, the last statement in the `getInterest` method is `return interest`. The method returns whatever value is stored in the `interest` variable, and the `interest` variable has type `double`. So far, so good.
- In Listing 15-11, the value returned by the call to `getInterest` is assigned to a variable named `yearlyInterest`. Sure enough, `yearlyInterest` is of type `double`.

That settles it! The use of types in the handling of method `getInterest` is consistent in Listings 15-10 and 15-11. I'm thrilled!

TRY IT OUT

Write a few programs to learn about creating Java methods.

ONE, TWO, THREE

In the code that follows, replace the comments with statements that do what the comments suggest:

```
public class Counter {
    int count = 0;

    void increment() {
        // Add 1 to the value of count
    }
}


public class Main {

    public static void main(String[] args) {
        var counter = new Counter();

        // Call the counter object's increment method

        System.out.println(counter.count);
    }
}
```

TWO, SEVEN, NINETEEN

Modify the code in the first paragraph of the previous section, "One, two, three," so that

- The `increment` method has a parameter.
- The `main` method calls `increment` several times, each time with a different parameter value.

The `increment` method uses its parameter to decide how much to increase the `count` value.

MOVE THINGS AROUND

Change the code in Listing 15-3 so that the `main` method begins with these lines:

```
public static void main(String[] args) {

    var myRandom = new Random();

    for (int i = 0; i < 3; i++) {
        var anAccount = new Account();
```

Does the new code run? If so, does it work correctly? Why, or why not?

PROCESS PURCHASES

At the end of Chapter 13, I complain about the repetitive code in Listing 13-5. When you see repetitive code, you can think about creating a method. You replace each repetition with a call to that method. The code's logic lives in one place (the method declaration), and the method calls repeatedly refer to that place. Et voilà! The repetition problem is solved!

Modify the code in Chapter 13 as follows:

- Add a `getTotal` method to the `Purchase` class code in Listing 13-2. The `getTotal` method takes no parameters and returns a `double` value.
- Replace the repetitive code in Listing 13-5 with two `getTotal` method calls.

More specifically:

- The getTotal method declares its own total variable.
- The getTotal method multiplies the Purchase object's unitPrice by the object's quantity and assigns the result to the total variable.
- Depending on the object's taxable value, the getTotal method either increases or doesn't increase the total variable's value.
- The getTotal method returns the value of the total variable.

HAS YOUR BMI CHANGED SINCE CHAPTER 13?

In the "What's your BMI?" experiment at the end of Chapter 13, you create a Person class, and your main method calculates a Person object's body mass index (BMI). Improve on that code so that the Person class contains its own getBmi method. The getBmi method calculates the Person object's body mass index from the values of the object's own weight and height fields.

In a separate class, the main method creates three Person objects, assigns values to each Person object's weight and height fields, and calls each Person object's getBmi method.

TWO TIMES NOTHING IS STILL NOTHING

In the "Nothing in particular" experiment at the end of Chapter 13, you create a Thing class, and your main method displays a sentence about a Thing object. Improve on that code so that the Thing class contains its own display method. The display method prints a Thing object's sentence based on the values of the object's own value1 and value2 fields.

In a separate class, the main method creates three Thing objects, assigns values to each Thing object's value1 and value2 fields, and calls each Thing object's display method.

MORE MACROECONOMICS

In the "A bit of macroeconomics" experiment at the end of Chapter 13, you create a Country class. Your main method asks the user for an acceptable debt-to-GDP ratio and reports That's acceptable or That's not acceptable after comparing a country's ratio to the acceptable ratio.

Improve on that code so that the Country class contains its own hasAcceptableRatio method. The hasAcceptableRatio method has one double parameter.

That parameter represents the debt-to-GDP ratio that the user has signified is acceptable. The hasAcceptableRatio method calculates the Country object's debt-to-GDP ratio from the values of object's own debt and gdp fields. The hasAcceptableRatio method returns a boolean value: true if the Country object's ratio is acceptable and false otherwise.

In a separate class, the main method creates three Country objects, assigns values to each Country object's debt and gdp fields, and calls each Country object's hasAcceptableRatio method.

EASY-PEASY

The "Make a hit record" experiment at the end of Chapter 13 introduces record classes. Why not try another example of Java's record class feature?

Put this code in a file named Journey.java:

```
public record Journey(int miles, double hours) {

    public double getSpeed() {
        return miles / hours;
    }
}
```

Also, put this code in a file named TakeAJourney.java:

```
public class TakeAJourney {

    public static void main(String[] args) {
        var journey = new Journey(400, 8);
        System.out.println(journey.getSpeed() + " miles per hour");
    }
}
```

And finally, run the TakeAJourney program.

What Next?

Object-oriented programming (OOP) is a pillar of modern programming, and every introduction to Java must include mention of it. But OOP isn't easy to master. Learning OOP presents special challenges for novice programmers.

This book covers only some shavings off the tip of the OOP iceberg. With that in mind, I heartily recommend a book written by my evil twin (and published by Wiley). *Java For Dummies* starts from scratch, but it includes about twice as much material as the book you're reading right now. In particular, *Java For Dummies* has more coverage of object-oriented programming.

Pick up the latest edition of *Java For Dummies* at your local bookstore. And, if you happen to make contact with the book's author, please tell him that I recommended his book.

5 Smart Java Techniques

IN THIS PART . . .

Getting input from your computer's hard drive

Doing more with decisions and loops

Working with bunches of things

Creating windows and other beautiful things

IN THIS CHAPTER

- Using data on your hard drive
- Writing code to access the hard drive
- Troubleshooting input/output behavior

Chapter 16

Piles of Files: Dealing with Information Overload

Consider these scenarios:

- **You're a business owner who handles hundreds of invoices each day.** You store invoice data in a file on your hard drive. You need customized code to sort and classify the invoices.
- **You're an astronomer with data from scans of the night sky.** When you're ready to analyze a chunk of data, you load the chunk onto your computer's hard drive.
- **You're the author of a popular self-help book.** Last year's fad was the Self-Mirroring Method. This year's craze is the Make Your Cake System. You can't modify your manuscript without converting to the publisher's new specifications. You need software to make the task bearable.

Each situation calls for a new computer program, and each program reads from a large data file. On top of all of that, each program creates a brand-new file containing bright, shiny results.

In previous chapters, the examples get input from the keyboard and send output to IntelliJ's Run tool window. That's fine for small tasks, but you can't have the computer prompt you for each bit of night sky data. For big problems, you need lots of data, and the best place to store the data is on a computer's hard drive.

Running a Disk-Oriented Program

To deal with volumes of data, you need tools for reading from (and writing to) disk files. At the mere mention of disk files, some people's hearts start to palpitate with fear. After all, a disk file is elusive and invisible. It's stored somewhere inside your computer, with some magic magnetic process.

The truth is, getting data from a disk is much like getting data from the keyboard. And printing data to a disk is like printing data to the computer screen.

TECHNICAL STUFF

In this book, displaying a program's text output "on the computer screen" means displaying text in IntelliJ's Run tool window. If you shun IntelliJ in favor of a different IDE (such as Eclipse or NetBeans) or you shun all IDEs in favor of your system's command window, then, for you, "on the computer screen" means something slightly different. Please read between the lines as necessary. Also, I'm well aware that many computers have SSD drives with no honest-to-goodness disks inside them. So, terms like *disk-oriented* and *disk files* are showing signs of age. But let's face facts: A "record store" no longer focuses on vinyl records, and in US measurement units, 12 inches is no longer the length of the king's foot. Today's LCD screens no longer need saving. And, unlike the old mechanical car radios, a web page's radio buttons don't mark your favorite stations.

Consider the scenario when you run the code in earlier chapters. You type some stuff on the keyboard. The program takes this stuff and spits out some stuff of its own. The program sends this new stuff to the Run tool window. In effect, the flow of data goes from the keyboard to the computer's innards and then to the screen, as shown in Figure 16-1.

Of course, the goal in this chapter is illustrated in Figure 16-2. It shows a file containing data on a hard drive. The program takes data from the disk file and spits out some brand-new data. The program then sends the new data to another file on the hard drive. In effect, the flow of data goes from a disk file to the computer's innards and on to another disk file.

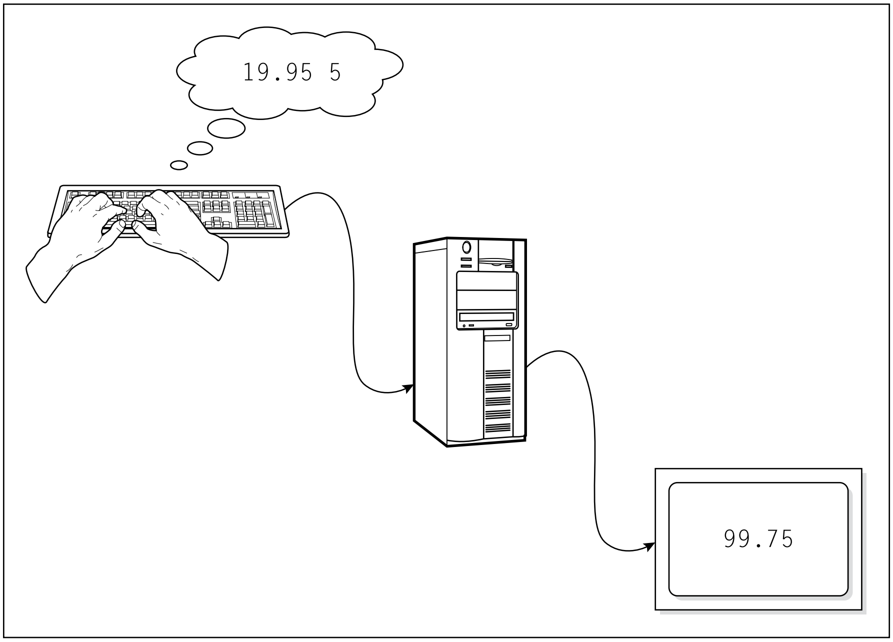


FIGURE 16-1: Using the keyboard and screen.

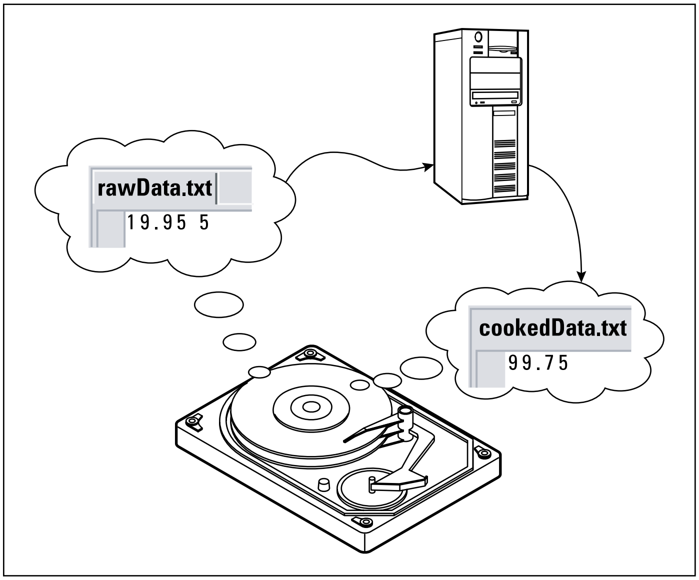


FIGURE 16-2: Using disk files.

The scenarios in Figures 16-1 and 16-2 are similar. In fact, it helps to remember these fundamental points:

- **The stuff in a disk file is no different from the stuff you type on a keyboard.**

  If a keyboard-reading program expects you to type `19.95 5`, then the corresponding disk-reading program expects a file containing those same characters, `19.95 5`. If a keyboard-reading program expects you to press Enter and type more characters, then the corresponding disk-reading program expects more characters on the next line in the file.

- **The stuff in a disk file is no different from the stuff you see in IntelliJ's Run tool window.**

  If a screen-printing program displays the number `99.75`, the corresponding disk-writing program writes the number `99.75` to a file. If a screen-printing program moves the cursor to the next line, then the corresponding disk-writing program creates a new line in the file.

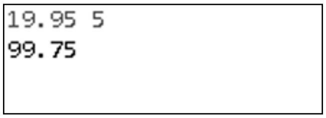
If you have trouble imagining what you have in a disk file, just imagine the text you would type on the keyboard or the text you would see on the computer screen (that is, in IntelliJ's Run tool window). That same text can appear in a file on your disk.

Reading and writing

Listing 16-1 contains a keyboard/screen program. The program multiplies unit price by quantity to produce the total price. A run of the code is shown in Figure 16-3.

LISTING 16-1: Using the Keyboard and the Screen

```
import java.util.Scanner;

public class ComputeTotal {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        double unitPrice, total;
        int quantity;

        unitPrice = keyboard.nextDouble();
        quantity = keyboard.nextInt();
        total = unitPrice * quantity;

        System.out.println(total);

        keyboard.close();
    }
}
```



```
19.95 5
99.75
```

FIGURE 16-3: Read from the keyboard; write to the screen.

REMEMBER

Grouping separators vary from one country to another. The run shown in Figure 16-3 works almost everywhere in the world. But if the unit price is 19-and-95-hundredths, you type `19.95` (with a dot) in some countries and `19,95` (with a comma) in others. When you install the computer's operating system, you tell it which country you live in. Java programs access this information and use it to customize the way the `nextDouble` method works.

The goal is to write a program like the one in Listing 16-1. But, rather than talk to your keyboard and screen, this new program talks to your hard drive. The new program reads unit price and quantity from your hard drive and writes the total back to your hard drive.

Java's API has everything you need for interacting with a hard drive. A nice example is in Listing 16-2.

LISTING 16-2: Using Input and Output Files

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

public class ReadAndWrite {

    public static void main(String[] args) throws FileNotFoundException {
        var diskScanner = new Scanner(new File("rawData.txt"));
        var diskWriter = new PrintStream("cookedData.txt");
        double unitPrice, total;
        int quantity;
```

(continued)

LISTING 16-2: ***(continued)***

```
        unitPrice = diskScanner.nextDouble();
        quantity = diskScanner.nextInt();

        total = unitPrice * quantity;

        diskWriter.println(total);

        diskScanner.close();
        diskWriter.close();
    }
}
```

For a guide to the care and feeding of the `rawData.txt` file (whose name appears in Listing 16-2), see the later section "Creating an input file."

Messing with files on your hard drive

> *I, ________ [print your name], agree to pay $_____ each month on the ___th day of the month.*

Fill in the blanks. That's all you have to do. Reading input from a disk can work the same way. Just fill in the blanks in Listing 16-3.

LISTING 16-3: **A Template to Read Data from a Disk File**

```
/*
 * Before IntelliJ can compile this code,
 * you must fill in the blanks.
 */

import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class ___________ {

    public static void main(String[] args) throws FileNotFoundException {

        var diskScanner = new Scanner(new File("_________"));

        ______ = diskScanner.nextInt();
        ______ = diskScanner.nextDouble();
        ______ = diskScanner.nextLine();
        ______ = diskScanner.findWithinHorizon(".",0).charAt(0);

        // Etc.

        diskScanner.close();
    }
}
```

To use Listing 16-3, make up a name for your class. Insert that name into the first blank space. Type the name of the input file in the second space (between the quotation marks). Then, to read a whole number from the input file, call `diskScanner.nextInt`. To read a number that has a decimal point, call `diskScanner.nextDouble`. You can call any of the `Scanner` methods in Chapter 5's Table 5-1 — the same methods you call when you get keystrokes from the keyboard.

The stuff in Listing 16-3 isn't a complete program. Instead, it's a *template* — a half-baked piece of code, with spaces for you to fill in. You can type Listing 16-3's lines into IntelliJ's editor and fill in the blanks at your leisure. But IntelliJ has a feature that deals specifically with templates. For more information, see this chapter's nearby sidebar, "Filling in the blanks."

FILLING IN THE BLANKS

When you right-click a project's `src` folder and choose New ➪ Java Class, IntelliJ creates a file with some code in it. The code looks something like this:

```
public class MyNewClass {
}
```

To create this code, IntelliJ uses a built-in template. You can see the template's inner workings. Here's how:

1. **Choose File ➪ Settings in Windows or IntelliJ IDEA ➪ Preferences on a Mac.**
2. **In the leftmost panel of the resulting dialog box, choose Editor ➪ File and Code Templates.**
3. **In the next-to-leftmost panel, select the Files tab.**

   A list of built-in templates appears.

(continued)

(continued)

4. **In the list of templates, select Class.**

   At that point, IntelliJ shows you some text that includes the following lines:

   ```
   public class ${NAME} {
   }
   ```

That's not Java code. It's an IntelliJ-specific description of a template for creating new class files.

You can create your own templates like the ones in Listings 16-3 and 16-4, but it's easier for you to scoop up templates that I've created for you. To do so, follow these instructions:

1. **On IntelliJ's main menu bar, choose File ➪ Manage IDE Settings ➪ Import Settings.**

   As a result, the Open dialog box appears.

2. **In the Open dialog box, navigate to the place on your hard drive where you downloaded this book's sample code.**
3. **In the sample code folder, drill down into the** `Chapter 16/16-03` **subfolder.**
4. **In the** `Chapter 16/16-03` **subfolder, double-click on the** `FileTemplates.zip` **file.**

   When you do, a dialog box asks you politely to check all components to import.

5. **Look for a check mark in the File Templates check box. Then click OK.**

   A new pop-up box asks whether you want to import and restart.

6. **Of course you do!**

   IntelliJ stops running and then begins its life anew. Now you can test the file handling templates.

7. **Right-click a project's `src` folder as though you're creating a new Java class.**
8. **Rather than choose New ➪ Java Class, choose New ➪ Dummies Read from Disk.**

   Pretty cool! Isn't it?

   A dialog box with the title New Dummies Read from Disk appears.

9. **In the File Name text field, type a new name for a Java class.**
10. **In the text field labeled Disk File Name Including the Extension, type the name of a file on your computer's hard drive — a name like `rawData.txt`.**
11. **With much fanfare and a sense of victory in your heart, click OK.**

    As a result, IntelliJ creates a brand-new Java class modeled after the code in Listing 16-3. Excellent!

With the template in Listing 16-3, you can input data from a disk file. With a similar template, you can write output to a file. The template is in Listing 16-4.

LISTING 16-4: A Template to Write Data to a Disk File

```
/*
 * Before IntelliJ can compile this code,
 * you must fill in the blanks.
 */

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

public class ___________ {

    public static void main(String[] args) throws FileNotFoundException {

        var diskWriter = new PrintStream("_________");

        diskWriter.print(_____);
        diskWriter.println(_____);

        // Etc.

        diskWriter.close();
    }
}
```

To use Listing 16-4, insert the name of your class into the first blank space. Type the name of the output file in the space between the quotation marks. Then, to write part of a line to the output file, call `diskWriter.print`. To write the remainder of a line to the output file, call `diskWriter.println`.

If your program gets input from one disk file and writes output to another, combine the stuff from Listings 16-3 and 16-4. When you do, you get a program like the one in Listing 16-2.

A QUICK LOOK AT JAVA'S DISK ACCESS FACILITIES

Templates like the ones in Listings 16-3 and 16-4 look very nice. But knowing how the templates work is even better. Here are a few tidbits describing the inner workings of Java's disk access code:

- **A `PrintStream` is something you can use for writing output.**

  A `PrintStream` is like a `Scanner`. The big difference is that a `Scanner` is for reading input, and a `PrintStream` is for writing output. To see what I mean, look at Listing 16-2. Notice the similarity between the statements that use `Scanner` and the statements that use `PrintStream`.

  The word `PrintStream` is defined in the Java API.

- **In Listing 16-2, the expression `new File("rawData.txt")` plays the same role that `System.in` plays in many other programs.**

  Just as `System.in` stands for the computer's keyboard, the expression `new File ("rawData.txt")` stands for a file on your computer's hard drive. When the computer calls `new File("rawData.txt")`, the computer creates something like `System.in` — something you can stuff inside the `new Scanner( )` parentheses.

  The word `File` is defined in the Java API.

- **A `FileNotFoundException` is something that may go wrong during an attempt to read input from a disk file (or an attempt to write output to a disk file).**

  Disk file access is loaded with pitfalls. Even the best programs run into disk access trouble occasionally. To brace against such pitfalls, Java insists on your adding some extra words to your code.

  In Listing 16-2, the added words `throws FileNotFoundException` form a throws clause. A *throws clause* is a kind of disclaimer. Putting a throws clause in your code is like saying, "I realize that this code can run into trouble."

  Of course, in the legal realm, you often have no choice about signing disclaimers. "If you don't sign this disclaimer, the surgeon won't operate on you." Okay then, I'll sign it. The same is true with a Java throws clause. If you put things `like new PrintStream("cookedData.txt")` in your code and you don't add something like `throws FileNotFoundException`, the Java compiler refuses to compile your code.

So, when do you need this throws FileNotFoundException clause, and when should you do without it? Well, having certain things in your code — things like new PrintStream("cookedData.txt") — forces you to create a throws clause. You can spend some time learning all about the kinds of things that demand throws clauses. But at this point, it's better to concentrate on other programming issues. Because you're a beginning Java programmer, the safest thing to do is to follow the templates in Listings 16-3 and 16-4.

The word FileNotFoundException is — you guessed it — defined in the Java API.

- **To create this chapter's code, I made up the names diskScanner and diskWriter.**

  The words diskScanner and diskWriter don't come from the Java API. In place of diskScanner and diskWriter, you can use any names you want. All you have to do is to use the names consistently within each of your Java programs.

- **A call to the close method ends the connection between your program and the file.**

  In many of this book's examples, you sever the connection between your program and the computer keyboard by calling keyboard.close(). The same is true when you call the close method for a disk file's scanner or a disk file's PrintStream instance. Calling the close method reminds Java to finish all pending read or write operations and to break the program's connection to the disk file or the keyboard or to whatever else holds data for the program.

  This book's examples are simple. If you omit a close method call in one of these examples, you might get a warning message from IntelliJ, but the world doesn't end. (That is, your program still runs correctly.) However, in a serious, make-it-or-break-it application, the proper placement of close calls is important. These close calls ensure the proper completion of the program's input and output actions and help free up disk resources for use by other running programs.

Running disk-oriented code

Testing the code in Listing 16-2 is a three-step process. Here's an outline of the three steps:

1. **Create the rawData.txt file.**
2. **Run the code in Listing 16-2.**
3. **View the contents of the newly created cookedData.txt file.**

The next few sections cover each step in detail.

Creating an input file

You can use any plain old text editor to create an input file for the code in Listing 16-2. In this section, I show you how to use IntelliJ's built-in editor.

TIP

If you don't want to create your own input file, you can skip this set of instructions. I've already created a `rawData.txt` file and included it in the material you download from this book's website. After uncompressing my `BeginProgJavaDummies6.zip` file, look for `rawData.txt` inside the `Chapter 16/16-02` folder.

To create an input file:

1. **Right-click the topmost branch of the tree in IntelliJ's Project tool window.**

   For example, if you've created a project named `16–02`, select the tree's `16–02` branch.

   REMEMBER

   In the Project tool window, select a branch whose label is the name of a project. Don't select an item within a project. (For example, don't select the `src` branch.)

2. **In the resulting context menu, choose New ➪ File.**

   IntelliJ's New File dialog box opens.

3. **In the Name field, type the name of your new data file.**

   You can type any name that your computer considers to be a valid filename. For this section's example, I used the name `rawData.txt`, but other names — such as `rawData.dat`, `rawData`, and `raw123.01.dataFile` — are fine. I try to avoid troublesome names (including short, uninformative names and names containing blank spaces), but the name you choose is entirely up to you (and your computer's operating system and your boss's whims and your customer's specifications).

4. **Press Enter.**

   What happens next depends on the name you typed.

5. **If IntelliJ shows you a dialog box complaining that the name you entered isn't associated with any particular file type, select Text from the list of types and then click OK.**

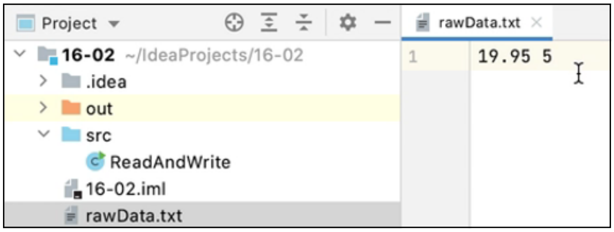
   At last, your new file's name appears in IntelliJ's Project tool window.

6. **If IntelliJ doesn't automatically open your file in the editor, double-click the file's branch in the Project tool window.**

   An empty editor (with the new file's name on its tab) appears in IntelliJ's editor.

7. **Type text in the editor.**

To create this section's example, I typed the text `19.95 5`, as shown in Figure 16-4. To create your own example, type whatever text your program needs during its run.



```
19.95 5
```

FIGURE 16-4: Editing an input file.

WARNING

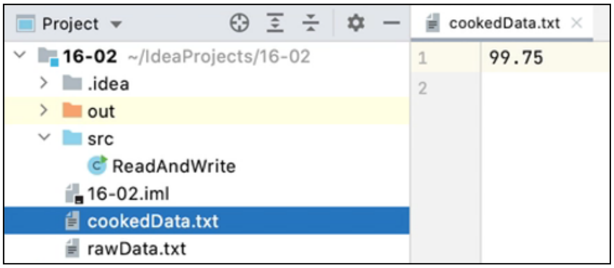
This section's steps apply whenever you use IntelliJ IDEA to create an input file. You can use other programs to create input files, such as Windows Notepad or Macintosh TextEdit. But if you do, you have to be careful about file formats and filename extensions. For example, to create a file named `raw123.01.dataFile` using Windows Notepad, type **"raw123.01.dataFile"** (with quotation marks) in the File Name field of the Save As dialog box. If you don't surround the name with quotation marks, Notepad might add `.txt` to the file's name (turning `raw123.01.dataFile` into `raw123.01.dataFile.txt`). A similar issue applies to the Macintosh's TextEdit program. By default, TextEdit adds the `.rtf` extension to each new file. To override the `.rtf` default for a particular file, choose Format➪Make Plain Text before saving the file. Then, whenever you save the file, TextEdit offers to add the `.txt` extension to the name of the file. In the Save As dialog box, if you don't want the file's name to end in `.txt`, uncheck the check box labeled If No Extension Is Provided, Use .txt.

Running the code

To have IntelliJ run the code, do the same thing you do with any other Java program. For example, to run the code in Listing 16-2, right-click the `ReadAndWrite` branch in the Project tool window. Then choose Run 'ReadAndWrite.main()'.

When you run the program in Listing 16-2, no text appears in IntelliJ's Run tool window. This total lack of any noticeable output gives some people the willies. The truth is, a program like the one in Listing 16-2 does all its work behind the scenes. The program has no statements that read from the keyboard and has no statements that print to the screen. So, if you have a very loud hard drive, you may hear a little chirping sound when you run the code, but you won't type any program input and you won't see any program output.

The program sends all its output to a file on your hard drive. To see the program's output, double-click the newly formed cookedData.txt branch of the Project tool window's tree. (See Figure 16-5.)

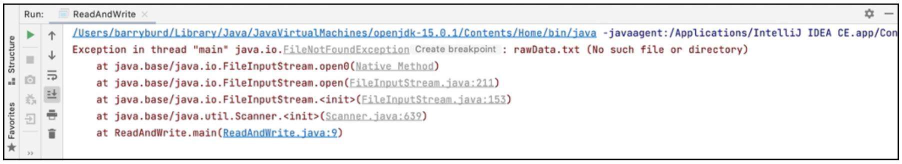


```
99.75
```

FIGURE 16-5: Viewing an output file.

File and error

When you run the code in Listing 16-2, the computer executes new Scanner(new File("rawData.txt")). If the Java virtual machine can't find the rawData.txt file, you see a message like the one shown in Figure 16-6. This error message can be frustrating. In many cases, you know darn well that there's a rawData.txt file on your hard drive. The stupid computer simply can't find it.



```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk-15.0.1/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Con
Exception in thread "main" java.io.FileNotFoundException Create breakpoint : rawData.txt (No such file or directory)
    at java.base/java.io.FileInputStream.open0(Native Method)
    at java.base/java.io.FileInputStream.open(FileInputStream.java:211)
    at java.base/java.io.FileInputStream.<init>(FileInputStream.java:153)
    at java.base/java.util.Scanner.<init>(Scanner.java:639)
    at ReadAndWrite.main(ReadAndWrite.java:9)
```

FIGURE 16-6: The computer can't find your file.

There's no quick, surefire way to fix this problem. But you should always check the following things first:

- **Check again for a file named rawData.txt.**

  Open Windows File Explorer or Macintosh Finder and poke around for a file with that name.

  TIP

  The filenames displayed in File Explorer and Finder can be misleading. You may see the name rawData, even though the file's real name is rawData.txt. To fix this problem once and for all, refer to Chapter 2.

- **Check the spelling of the file's name.**

  Make sure that the name in your program is exactly the same as the name of the file on your hard drive. Just one misplaced letter can keep the computer from finding a file.

- **If you use Linux (or a flavor of UNIX other than Mac OS X), check the capitalization of the file's name.**

  In Linux, and in many versions of UNIX, the difference between uppercase and lowercase can baffle the computer.

- **Ensure that the file is in the correct directory.**

  Sure, you have a file named `rawData.txt`. But don't expect your Java program to look in every folder on your hard drive to find the file. How do you know which folder should house files like `rawData.txt`?

  Here's how it works: Chapter 12 says that each IntelliJ project has its own root folder on your computer's hard drive. In Figure 16-5, you see the `16-02` project's root folder, the root folder's `src` subfolder, and some other stuff. Look carefully, and notice that the `rawData.txt` and `cookedData.txt` files are inside the `16-02` root folder. They're not inside the `src` subfolder, the `out` subfolder, or any other subfolder. If this were a family tree, `rawData.txt` and `cookedData.txt` would be children of `06-02`, not grandchildren.

  REMEMBER

  When you run this section's example, the `rawData.txt` file should be in the root of the `16-02` project folder on your hard drive. That's why, in Step 1 of the earlier "Creating an input file" section, I remind you to select the `16-02` project folder and not the project's `src` subfolder.

  Figure 16-5 shows input and output files in the root of their IntelliJ IDEA project. But in general, file locations can be tricky, especially if you switch from IntelliJ to an unfamiliar IDE. The general rule (about putting input and output files immediately inside a project directory) may not apply in other programming environments.

  Here's a trick you can use: Whatever IDE you use (or even if you create Java programs without an IDE), run this stripped-down version of the code in Listing 16-2:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

public class JustWrite {

    public static void main(String[] args) throws FileNotFoundException {

        var diskWriter = new PrintStream("cookedData.txt");
        diskWriter.println(99.75);

        diskWriter.close();
    }
}
```

This program has no need for a stinking `rawData.txt` file. If you run this code and see no error messages, search your hard drive for this program's output (the `cookedData.txt` file). Note the name of the folder that contains the `cookedData.txt` file. When you put `rawData.txt` in this same folder, any problem you had running the Listing 16-2 code should go away.

- **Check the `rawData.txt` file's content.**

  It never hurts to peek inside the `rawData.txt` file and make sure that the file contains the numbers `19.95 5`. Double-clicking the Project tool window's `rawData.txt` branch makes that file appear in IntelliJ's editor area.

REMEMBER

By default, Java's `Scanner` class looks for blank spaces between input values. So, this example's `rawData.txt` file should contain `19.95 5`, not `19.955` and not `19.95,5`.

TECHNICAL STUFF

The `Scanner` class looks for any kind of *white space* between the values. These white space characters may include blank spaces, tabs, and end-of-line markers. For example, the `rawData.txt` file may contain `19.95  5` (with several blank spaces between `19.95` and `5`), or it may have `19.95` and `5` on two separate lines.

Writing a Disk-Oriented Program

Listing 16-2 is much like Listing 16-1. In fact, you can go from Listing 16-1 to Listing 16-2 with some simple editing. Here's how:

- **Add the following import declarations to the beginning of your code:**

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
```

- **Add the following throws clause to the method header:**

```
throws FileNotFoundException
```

- **In the call to `new Scanner`, replace `System.in` with a call to `new File` as follows:**

```
var aVariableName = new Scanner(new File("inputFileName"))
```

REMEMBER

In a declaration of this kind, you can always use the word `Scanner` instead of `var`. In fact, if this declaration isn't inside of a method, you must use the word `Scanner` instead of `var`. For details, refer to Chapter 13.

- **Create a `PrintStream` for writing output to a disk file:**

  ```
  var anotherVariableName = new PrintStream("outputFileName");
  ```

- **Use the `Scanner` variable name in calls to `nextInt`, `nextLine`, and so on.**

  For example, to go from Listing 16-1 to Listing 16-2, I change

  ```
  unitPrice = keyboard.nextDouble();
  quantity = keyboard.nextInt();
  ```

  to

  ```
  unitPrice = diskScanner.nextDouble();
  quantity = diskScanner.nextInt();
  ```

- **Use the `PrintStream` variable name in calls to `print` and `println`.**

  For example, to go from Listing 16-1 to Listing 16-2, I change

  ```
  System.out.println(total);
  ```

  to

  ```
  diskWriter.println(total);
  ```

- **Use the `Scanner` variable name in the call to `close`.**

  For example, to go from Listing 16-1 to Listing 16-2, I change

  ```
  keyboard.close();
  ```

  to

  ```
  diskScanner.close();
  ```

- **Use the `PrintStream` variable name in a call to `close`.**

  For example, to go from Listing 16-1 to Listing 16-2, I add

  ```
  diskWriter.close();
  ```

  at the end of the `main` method.

Reading from a file

All the `Scanner` methods can read from existing disk files. For example, to read a word from a file named `mySpeech`, use code of the following kind:

```
var diskScanner = new Scanner(new File("mySpeech"));
String oneWord = diskScanner.next();
```

To read a character from a file named `letters.dat` and then display the character on the screen, you can do something like this:

```
var diskScanner = new Scanner(new File("letters.dat"));
System.out.println(diskScanner.findWithinHorizon(".", 0).charAt(0));
```

TECHNICAL STUFF

Notice how I read from a file named `mySpeech`, not `mySpeech.txt` or `mySpeech.doc`. Anything you put after the dot is called a *filename extension*, and for a file full of numbers and other data, the filename extension is optional. Sure, a Java program must be called *`something`*`.java`, but a data file can be named `mySpeech.txt`, `mySpeech.reallymine.allmine`, or just `mySpeech`. As long as the name in your `new File` call is the same as the filename on your computer's hard drive, everything is okay.

Writing to a file

The `print` and `println` methods can write to disk files. Here are some examples:

- During a run of the code in Listing 16-2, the variable `total` stores the number 99.75. To deposit 99.75 into the `cookedData.txt` file, you execute

  ```
  diskWriter.println(total);
  ```

  This `println` call writes to a disk file because of the following line in Listing 16-2:

  ```
  var diskWriter = new PrintStream("cookedData.txt");
  ```

- In another version of the program, you may decide not to use a `total` variable. To write 99.75 to the `cookedData.txt` file, you can call

  ```
  diskWriter.println(unitPrice * quantity);
  ```

- To display `OK` on the screen, you can make the following method call:

  ```
  System.out.print("OK");
  ```

  To write `OK` to a file named `approval.txt`, you can use the following code:

  ```
  var diskWriter = new PrintStream("approval.txt");
  diskWriter.print("OK");
  ```

- You may decide to write `OK` as two separate characters. To write to the screen, you can make the following calls:

  ```
  System.out.print('O');
  System.out.print('K');
  ```

And to write `OK` to the `approval.txt` file, you can use the following code:

```
var diskWriter = new PrintStream("approval.txt");

diskWriter.print('O');
diskWriter.print('K');
```

- Like their counterparts for `System.out`, the disk writing `print` and `println` methods differ in their end-of-line behaviors. Say you want to display the following text on the screen:

```
Hankees  Socks
7        3
```

To do this, you can make the following method calls:

```
System.out.print("Hankees  ");
System.out.println("Socks");
System.out.print(7);
System.out.print("        ");
System.out.println(3);
```

To plant the same text into a file named `scores.dat`, you can use the following code:

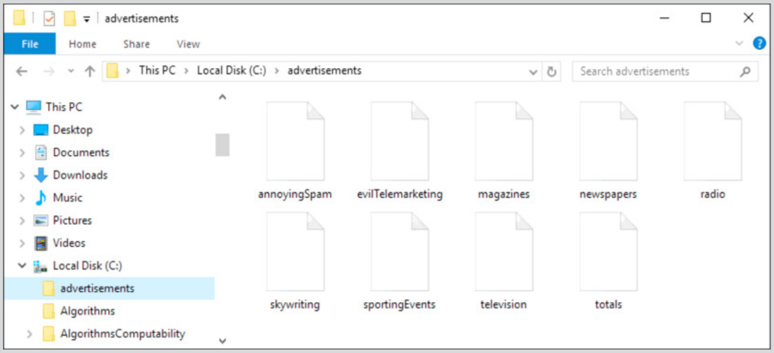
```
var diskWriter = new PrintStream("scores.dat");

diskWriter.print("Hankees  ");
diskWriter.println("Socks");
diskWriter.print(7);
diskWriter.print("        ");
diskWriter.println(3);
```

NAME THAT FILE

What if a file that contains data isn't in your program's project folder? If that's the case, when you call `new File`, the file's name must include folder names. For example, in Windows, your `TallyBudget.java` program might be in your `c:\Users\MyUserName\IdeaProjects\16-09` folder, and a file named `totals` might be in a folder named `c:\advertisements`. (See the following figure.)

(continued)

(continued)



Then, to refer to the `totals` file, you include the folder name, the filename, and (to be on the safe side) the drive letter:

```
var diskScanner = new Scanner(new File("c:\\advertisements\\totals"));
```

Notice that I use double backslashes to separate the drive letter, the folder name, and the filename. To find out why, look at the material on escape sequences in Chapters 10 and 11. The string `"\totals"` with a single backslash stands for a tab, followed by `otals`. But in this example, the file's name is `totals`, not `otals`. And with only one backslash, `"\advertisements"` makes no sense. IntelliJ would flag \a as an illegal escape character.

Inside quotation marks, you use the double backslash to indicate what would usually be a single backslash. So the string `"c:\\advertisements\\totals"` stands for `c:\advertisements\totals`. That's good because `c:\advertisements\totals` is the way you normally refer to a file in Windows.

If you want to sidestep all this backslash confusion, you can use forward slashes to specify each file's location. Windows responds exactly the same way to `new File(`**`"c:\\advertisements\\totals"`**`)` and to `new File(`**`"c:/advertisements/totals"`**`)`. And, if you use UNIX, Linux, or a Macintosh, the double backslash nonsense doesn't apply to you. Just write

```
var diskScanner = new Scanner (new File("/Users/me/advertisements/totals"));
```

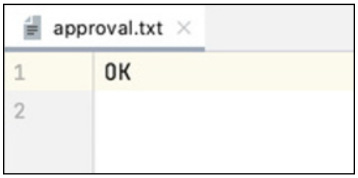
or something similar that reflects your system's directory structure.

Writing, Rewriting, and Re-Rewriting

Given my mischievous ways, I tried a little experiment. I asked myself what would happen if I ran the same file writing program more than once. So I created a tiny program (the program in Listing 16-5), and I ran the program twice. Then I examined the program's output file. The output file (shown in Figure 16-7) contains only two letters.

LISTING 16-5: A Little Experiment

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

public class WriteOK {

    public static void main(String[] args) throws FileNotFoundException {

        var diskWriter = new PrintStream(new File("approval.txt"));

        diskWriter.print  ('O');
        diskWriter.println('K');

        diskWriter.close();
    }
}
```



```
OK
```

FIGURE 16-7: Testing the waters.

Here's the sequence of events, from the start to the end of the experiment:

1. **Before I run the code in Listing 16-5, my computer's hard drive has no `approval.txt` file.**

   That's okay. Every experiment has to start somewhere.

2. **I run the code in Listing 16-5.**

   The call to `new PrintStream` in Listing 16-5 creates a file named `approval.txt`. Initially, the new `approval.txt` file contains no characters. Later in

Listing 16-5, calls to `print` and `println` put characters in the file. So, after I run the code, the `approval.txt` file contains two letters: the letters `OK`.

3. **I run the code from Listing 16-5 a second time.**

   At this point, I could imagine seeing `OKOK` in the `approval.txt` file. But that's not what I see in Figure 16-7. After running the code twice, the `approval.txt` file contains just one `OK`. Here's why:

   - The call to `new PrintStream` in Listing 16-5 deletes my existing `approval.txt` file. The call creates a new, empty `approval.txt` file.
   - After a new `approval.txt` file is created, the `print` method call drops the letter `O` into the new file.
   - The `println` method call adds the letter `K` to the same `approval.txt` file.

That's the story. Each time you run the program, it trashes whatever `approval.txt` file is already on the hard drive. Then the program adds data to a newly created `approval.txt` file.

TRY IT OUT

File handling can be tricky. If you run into trouble early on, it's easy to become frustrated. Fortunately, these experiments will get you started on the right track.

RUN BARRY'S CODE

Test the waters by downloading the code from this book's website (`http://beginprog.allmycode.com`). The `16-02` folder comes with its own `rawData.txt` file. Follow the instructions in this chapter's "Running disk-oriented code" section for running the code in Listing 16-2. After running the program, check IntelliJ's Project tool window to make sure that the run has created a `cookedData.txt` file.

WHERE'S MY FILE?

Create an IntelliJ project containing the following code:

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

public class ReadAndWrite {

    public static void main(String[] args) throws FileNotFoundException {
        var diskScanner = new Scanner(new File("data.txt"));
        var diskWriter = new PrintStream("data.txt");

        diskWriter.println("Hello");

        System.out.println(diskScanner.next());

        diskScanner.close();
        diskWriter.close();
    }
}
```

When you run the code, you see an error message in IntelliJ's Run tool window. Why?

WRITE AND THEN READ

Modify the code from the where's-my-file experiment so that the `var diskWriter` declaration comes before the `var diskScanner` declaration.

When you run the code, the word `Hello` should appear in IntelliJ's Run tool window. After running the code, check to make sure that your IntelliJ project contains a file named `data.txt`.

RANDOM NUMBERS IN A FILE

Create a program that writes ten randomly generated numbers in a disk file. After writing the numbers, the program reads the numbers from the file and displays them in IntelliJ's Run tool window.

IN THIS CHAPTER

- Dealing with many alternatives
- Jumping out from the middle of a statement
- Handling alternative assignments

Chapter 17
How to Flick a Virtual Switch

Imagine playing *Let's Make a Deal* with ten different doors: "Choose door number 1, door number 2, door number 3, door number 4 — wait! Let's break for a commercial. When we come back, I'll say the names of the other six doors."

What Wayne Brady (the show's host) needs is Java's `switch` statement.

Meet the switch Statement

The code in Listing 9-2 (refer to Chapter 9) simulates a fortune-telling toy — an electronic oracle. Ask the program a question and the program randomly generates a yes-or-no answer. But, as toys go, the code in Listing 9-2 isn't much fun. The code has only two possible answers. There's no variety. Even the earliest talking dolls could say about ten different sentences.

Suppose that you want to enhance the code of Listing 9-2. The call to `myRandom.nextInt(10) + 1` generates numbers from 1 to 10. So maybe you can display a different sentence for each of the ten numbers. A big pile of `if` statements should do the trick:

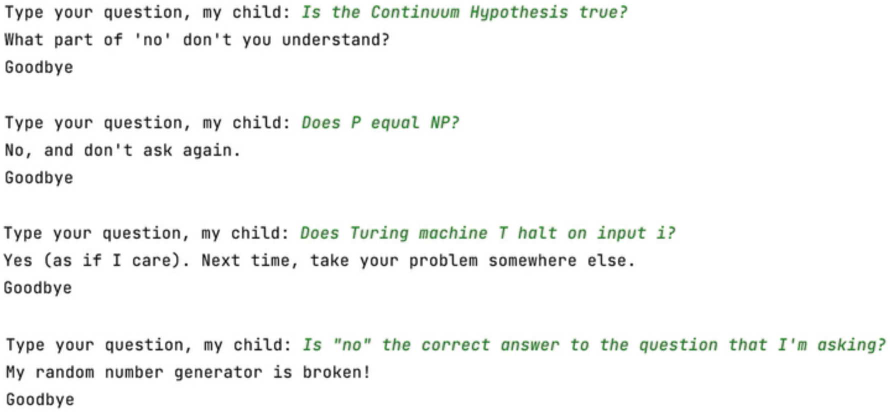
```
if (randomNumber == 1) {
    System.out.println("Yes. Isn't it obvious?");
}
if (randomNumber == 2) {
    System.out.println("What part of 'no' don't you understand?");
}
if (randomNumber == 3) {
    System.out.print("Yessir, yessir! Three bags full.");
}
if (randomNumber == 4)
    .
    .
    .
if (randomNumber < 1 || randomNumber > 10) {
    System.out.println("My random number generator is broken!");
}
```

But that approach seems wasteful. Why not create a statement that checks the value of `randomNumber` just once and then takes an action based on the value it finds? Fortunately, just such a statement exists: the *switch* statement. Listing 17-1 has an example of a `switch` statement.

LISTING 17-1: A Rude Answer for Every Occasion

```
import java.util.Random;
import java.util.Scanner;

import static java.lang.System.out;

public class TheOldSwitcheroo {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        var myRandom = new Random();
        int randomNumber;

        out.print("Type your question, my child: ");
        keyboard.nextLine();
        randomNumber = myRandom.nextInt(10) + 1;

        switch (randomNumber) {
            case 1 -> out.println("Yes. Isn't it obvious?");
            case 2 -> out.println("What part of 'no' don't you understand?");
            case 3 -> out.println("Yessir, yessir! Three bags full.");
            case 4, 5 -> out.println("No, and don't ask again.");
            case 6 -> out.println("Sure, whatever.");
            case 7 -> out.println("Yes, but only if you're nice to me.");
            case 8 -> {
                out.print("Yes (as if I care).");
                out.println(" Next time, take your problem somewhere else.");
            }
            case 9 -> out.println("No, not until Cromwell seizes Dover.");
            case 10 -> out.println("No, not until Nell squeezes Rover.");
            default -> out.println("My random number generator is broken!");
        }

        out.println("Goodbye");

        keyboard.close();
    }
}
```

Figure 17-1 shows four runs of the program in Listing 17-1.



```
Type your question, my child: Is the Continuum Hypothesis true?
What part of 'no' don't you understand?
Goodbye

Type your question, my child: Does P equal NP?
No, and don't ask again.
Goodbye

Type your question, my child: Does Turing machine T halt on input i?
Yes (as if I care). Next time, take your problem somewhere else.
Goodbye

Type your question, my child: Is "no" the correct answer to the question that I'm asking?
My random number generator is broken!
Goodbye
```

FIGURE 17-1: Running the code from Listing 17-1.

Anatomy of a switch statement

The overall idea behind the program in Listing 17-1 is illustrated in Figure 17-2.

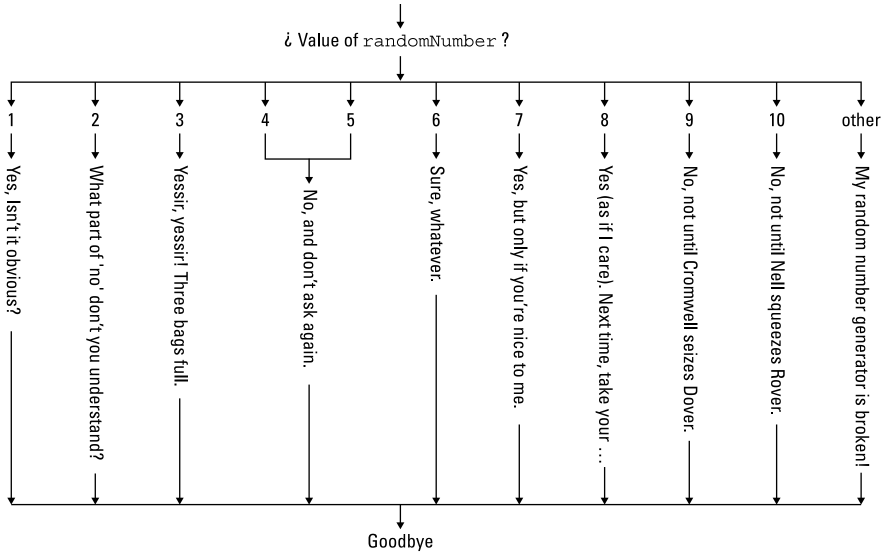

¿ Value of randomNumber ?
1
2
3
4
5
6
7
8
9
10
other
Yes, Isn't it obvious?
What part of 'no' don't you understand?
Yessir, yessir! Three bags full.
No, and don't ask again.
Sure, whatever.
Yes, but only if you're nice to me.
Yes (as if I care). Next time, take your . . .
No, not until Cromwell seizes Dover.
No, not until Nell squeezes Rover.
My random number generator is broken!
Goodbye

FIGURE 17-2: A fork with many prongs.

Here's what happens during the first run in Figure 17-1:

- The user types a heavy question, and the variable `randomNumber` gets a value. In the first run of Figure 17-1, this value is 2.
- Execution of the code in Listing 17-1 reaches the top of the `switch` statement, so the computer starts checking this statement's `case` clauses. The value 1 in the topmost `case` clause doesn't match the `randomNumber` value 2, so the computer moves on to the next `case` clause.
- The value in the next `case` clause (the number 2) matches the value of the `randomNumber` variable, so the computer executes the statement

  ```
  out.println("What part of 'no' don't you understand?");
  ```

  in this `case 2` clause. The computer displays `What part of 'no' don't you understand?` in IntelliJ's Run tool window. Then the computer skips right past `case 3`, `case 4`, and so on. The computer jumps to the statement just after the end of the `switch` statement.
- The computer displays `Goodbye` because that's what the statement after the `switch` statement tells the computer to do.

The second run in Figure 17-1 portrays a slightly different situation.

- In that run, the value of `randomNumber` is either 4 or 5.
- Execution of the code in Listing 17-1 reaches the top of the `switch` statement, so the computer starts checking this statement's `case` clauses. The values in

the topmost three case clauses don't match the randomNumber value, so the computer moves on to the case 4, 5 clause.

- One of the values in the case 4, 5 clause matches the value of the randomNumber variable, so the computer executes the statement in the case 4, 5 clause:

  ```
  out.println("No, and don't ask again.");
  ```

  Then the computer skips right past case 6, case 7, and so on. The computer jumps to the statement just after the end of the switch statement.

- The computer displays Goodbye because that's what the statement after the switch statement tells the computer to do.

For the third run in Figure 17-1, the value of randomNumber is 8. Nothing exciting happens during the run. But, in Listing 17-1, notice how curly braces surround the case 8 clause's statements. If a case clause contains more than one statement, curly braces are mandatory.

To create the fourth run in Figure 17-1, I cheated. I added the line

```
randomNumber = 11;
```

immediately before the switch statement. The computer responded by dropping past all the case clauses. Rather than land on a case clause, the computer jumped to the default clause. In the default clause, the computer displayed My random number generator is broken! and then moved out of the switch statement. When the computer was out of the switch statement, the computer displayed Goodbye.

Picky details about the switch statement

A switch statement can take the following form:

```
switch (Selector) {
    case FirstValue -> Statements
    case SecondValue -> MoreStatements

    // ... more cases ...

    default -> EvenMoreStatements
}
```

Here are some tidbits about `switch` statements:

- **The *Selector* doesn't have to have an `int` value. It can also be a `char`, `byte`, `short`, `String`, or `enum` value.**

  For example, the following code samples turn a `char` into a `String` and turn a `String` into a `char`:

  ```
  // char to String

  char letterGrade = keyboard.findWithinHorizon(".", 0).charAt(0);

  switch (letterGrade) {
      case 'A' -> out.println("Excellent");
      case 'B' -> out.println("Good");
      case 'C' -> out.println("Average");
      default  -> out.println("Other");
  }

  // String to char

  String description = keyboard.next();

  switch (description) {
      case "Excellent" -> out.println('A');
      case "Good" -> out.println('B');
      case "Average" -> out.println('C');
      default -> out.println("Other");
  }
  ```

  CROSS REFERENCE

  I introduce `enum` types in Chapter 10 and describe the `String` class with enthusiasm in Chapter 14.

- **The *Selector* doesn't have to be a single variable. It can have any value of type `char`, `byte`, `short`, `int`, `String`, or `enum`.**

  For example, you can simulate the rolling of two dice with the following code:

  ```
  int die1, die2;

  die1 = myRandom.nextInt(6) + 1;
  die2 = myRandom.nextInt(6) + 1;

  switch (die1 + die2) {
  // ... etc.
  ```

- **The `case` clauses in a `switch` statement don't have to be in order.**

  Here's some acceptable code:

  ```
  switch (randomNumber) {
      case 2 -> out.println("What part of 'no' don't you understand?");
      case 1 -> out.println("Yes. Isn't it obvious?");
      case 3 -> out.println("Yessir, yessir! Three bags full.");
  // ... etc.
  ```

  This mixing of `case` clauses may slow you down when you're trying to read and understand a program, but it's legal, nonetheless.

- **You don't need a `case` for each expected value of the *Selector*. You can leave some expected values to the `default`.**

  Here's an example:

  ```
  switch (randomNumber) {
      case 1 -> out.println("Yes. Isn't it obvious?");
      case 4, 5 -> out.println("No, and don't ask again.");
      case 10 -> out.println("No, not until Nell squeezes Rover.");
      default -> out.println("Sorry, I just can't decide.");
  }
  ```

- **The `default` clause is optional.**

  If you have no `default` clause and a value that's not covered by any of the cases comes up, the `switch` statement does nothing. The computer marches on and executes whatever statement comes immediately after the `switch` statement.

  What happens if you make one tiny change to the code in Listing 17-1? You remove the `switch` statement's `default` clause. The answer is that nothing happens. The value of `myRandom.nextInt(10) + 1` is always a number from 1 to 10. The program runs correctly, and all's right with the world. So, why bother with a `default` clause in the first place? The computer will never execute the `switch` statement's `default` clause. Right?

  No, that's not right. Most real-life programming problems are quite complicated. Determining that variables have the values you expect is never simple. What if another programmer on your team changes the way `randomNumber` gets its value? When you anticipate the things that can go wrong with your code, you build a more secure line of defense. It's always best to guard against possible problems — problems that you know can occur and problems that you think will never occur. Put a `default` clause in each of your `switch` statements.

- **In some situations, `if` statements are more versatile than `switch` statements.**

  For example, you can't use a condition in a `switch` statement's *Selector*:

  ```
  //You can't do this:
  switch (age >= 12 && age < 65)
  ```

  You can't use a condition as a `case` value, either:

  ```
  //You can't do this:
  switch (age) {
      case age <= 12 -> // ... etc.
  ```

TRY IT OUT

Here's where you gain some practice using `switch` statements.

DAYS OF THE WEEK

Write a program that reads a number (from 1 to 7) and displays the day of the week corresponding to that number. For example, in the United States, Sunday is counted as the first day of the week. So, if the user types `1`, my program displays `Sunday`. If the user types `2`, my program displays `Monday`. And so on.

TIME TO EAT

Write a program that asks the user what the current hour is, and uses a `switch` statement to inform the user about mealtime. If the hour is between 06:00 and 09:00, tell the user, "Breakfast is served." If the hour is between 11:00 and 13:00, tell the user, "Time for lunch." If the hour is between 17:00 and 20:00, tell the user, "It's dinnertime." For any other hours, tell the user, "Sorry. You'll have to wait or go find a snack."

TIME TO EAT IN THE UNITED STATES

Redo this section's Time To Eat program so that it works with a 12-hour clock. From 6 a.m. to 9 a.m., it's time for breakfast. From 11 a.m. to 1 p.m., lunch is available. And from 5 p.m. to 8 p.m., you can chow down on dinner.

A TINY CALCULATOR

Write a program that prompts the user for two numbers and a character. In a `switch` statement, the program applies the character's operation to the two numbers and displays the result. A run of the program might look like this:

```
First number: 21.0
Second number: 8.0
Operation (+ - * or /): +
29.0
```

The program accepts any one of the characters +, -, *, or /. For an additional challenge, enhance the program's output as follows:

```
First number: 31.0
Second number: 10.0
Operation (+ - * or /): *
31.0 * 10.0 = 310.0
```

Remember that you can't use Java's + sign to display a `char` value next to a numeric value. If you execute `System.out.println(5 + ' ' + '*')`, Java doesn't display `5 *`. Instead, Java displays `79`. (If you don't believe me, try it in JShell.)

COLOR BY NUMBERS

In the RGB color model, numeric values indicate amounts of red, green, and blue. If you mix red, green, and blue to show only eight colors, 0 is black, 1 is blue, 2 is green, 3 is cyan, 4 is red, 5 is magenta, 6 is yellow, and 7 is white. Write a program that reads a number from the keyboard and displays the name of that number's color.

A Switch in Time

In 2020, with the release of Java 14, the Java world "switched gears" (pun intended). The stewards of Java introduced a brand-new feature — namely, the *switch expression*. Here's the story:

If you look again at Listing 17-1, you wonder why someone took so much delight in typing the words `out.println`. Those words appear ten times in the `switch` statement, and the only difference is the choice of a snappy retort. This repeated use of `out.println` seems wasteful. Why not have only one call to `out.println` for all the different answers the program can display?

Java's `switch` expression addresses this issue. Listing 17-2 shows you how.

LISTING 17-2: Out with the Old out.println!

```
import java.util.Random;
import java.util.Scanner;

import static java.lang.System.out;

public class TheNewSwitcheroo {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        var myRandom = new Random();
        int randomNumber;
        String output;

        out.print("What's your question, my child? ");
        keyboard.nextLine();
        randomNumber = myRandom.nextInt(10) + 1;

        output = switch (randomNumber) {
            case 1 -> "Yes. Isn't it obvious?";
            case 2 -> "What part of 'no' don't you understand?";
            case 3 -> "Yessir, yessir! Three bags full.";
            case 4, 5 -> "No, and don't ask again.";
            case 6 -> "Sure, whatever.";
            case 7 -> "Yes, but only if you're nice to me.";
            case 8 -> """
                    Yes (as if I care).
                    Next time, take your problem somewhere else.""";
            case 9 -> "No, not until Cromwell seizes Dover.";
            case 10 -> "No, not until Nell squeezes Rover.";
            default -> "My random number generator is broken!";
        };

        out.println(output);

        keyboard.close();
    }
}
```

WARNING

If you're using an older version of Java, don't bother trying to run the code in Listing 17-2. It won't work.

Runs of the code in Listing 17-2 look exactly the same as the runs of Listing 17-1. (See Figure 17-1.) The big change is that Listing 17-2 consolidates most of the out.println calls. The change saves lots of typing. But, more importantly, Listing 17-2 is conceptually tighter than Listing 17-1. Needless repetition leads to errors, so Listing 17-2 is more likely to run correctly.

Dissecting the switch expression

In any computer language, the word *expression* refers to a bunch of code that stands for a value. In Listing 17-2, the entire `switch` expression stands for a string of characters. For example, when `randomNumber` is 2, the code

```
         switch (randomNumber) {
    case 1 -> "Yes. Isn't it obvious?";
    case 2 -> "What part of 'no' don't you understand?";
    // ...
    default -> "My random number generator is broken!";
};
```

stands for the string `"What part of 'no' don't you understand?"` So, in that scenario, Listing 17-2 does the same thing as this code:

```
output = "What part of 'no' don't you understand?";
out.println(output);
```

In this small snippet of code, notice how the first line ends with a semicolon. That's why there's a semicolon at the end of the big `switch` expression in Listing 17-2.

Can you switch between two kinds of switch?

Java's `switch` expression looks a lot like Java's `switch` statement, but with a few important differences:

- **In a `switch` expression, each `case` clause must have a value.**

  For example, the following code is illegal:

  ```
  // Bad code!
  output = switch (randomNumber) {
      case 1 -> out.println("Yes. Isn't it obvious?");
      case 2 -> "What part of 'no' don't you understand?";
      // Etc.
  ```

  The text `out.println("Yes. Isn't it obvious?")` can tell the computer to display something, but that text doesn't stand for a value of any kind. In the same way, Java won't accept the following assignment statement:

  ```
  // Bad code!
  output = out.println("Yes. Isn't it obvious?");
  ```

Despite appearances, `out.println("Yes. Isn't it obvious?")` doesn't stand for the string `"Yes. Isn't it obvious?"`. The `out.println` call and the `"Yes..."` string aren't interchangeable. It's the same way in English. An instruction to "Say *cheese*" isn't the same as saying only the word *cheese.* If it were, some photos would capture large groups of people with their mouths in the shape of the word *Say.*

CROSS REFERENCE

I can show you ways to squeeze `out.println` calls and other such things into `switch` expressions. For more info, see the next section.

- **A `switch` statement can stand on its own, but a `switch` expression can't.**

  In an effort to make Listing 17-2 look more like Listing 17-1, try changing the line

  ```
  output = switch (randomNumber) {
  ```

  by removing the `output =` part. When you do, IntelliJ displays a bunch of squiggly, red lines to tell you that your code is damaged goods. The problem is, a `switch` expression on its own doesn't tell the computer to do anything. Imagine having the following lines in a Java program:

  ```
  // Bad code!
  "Yes. Isn't it obvious?";
  out.println(output);
  ```

  It's like asking someone how to bake a cake and having them reply with nothing but the ingredients.

  TIP

  There are many ways to keep a `switch` expression from standing on its own. For example, in Listing 17-2, you can bypass the use of the `output` variable with code of the following kind:

  ```
  out.println(switch (randomNumber) {
      case 1 -> "Yes. Isn't it obvious?";
      case 2 -> "What part of 'no' don't you understand?";
      // ... Etc.
      default -> "My random number generator is broken!";
  });
  ```

- **In most situations, a `switch` expression's `default` clause isn't optional.**

  Go ahead: I dare you to remove the `default` clause from the code in Listing 17-2! In fact, I double-dare you, which means that I have to try it, too.

  In the end, neither of us is happy. A `switch` expression must account for every possible *Selector* value, even the values you think are impossible. The `switch` expression in Listing 17-2 must have a `default` clause just in case `randomNumber` is 11 or –17 or 2147483647. That's the rule.

TECHNICAL STUFF

The one exception to the rule about having a `default` clause is when the *Selector* is an `enum` value. For example, the following code is okay:

```
enum WhoWins {home, visitor, neither}
WhoWins who;

// After getting a value for the who variable ...

output = switch(who) {
    case home -> "Hankees wins.";
    case visitor -> "Socks wins.";
    case neither -> "It's a tie.";
};
```

TRY IT OUT

Redo the examples named Days of the Week and Time to Eat from the previous Try It Out section. This time, use `switch` expressions.

TAKE A FLYING LEAP

Every four years, my wife and I celebrate our wedding anniversary. We were married many years ago on February 29. In honor of that occasion, I present another `switch` statement example. This example reports the number of days in any particular month of the year.

The example comes in two forms — one with a `switch` statement and the other with a `switch` expression. Listing 17-3 has the `switch` statement.

LISTING 17-3: Thirty Days Hath September

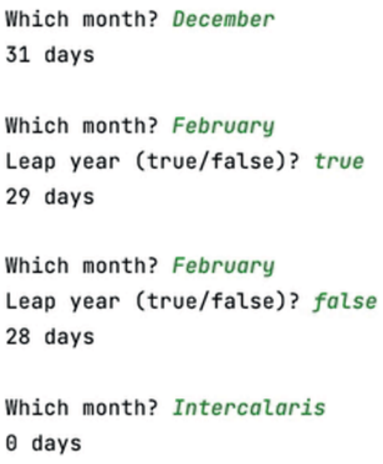
```
import java.util.Scanner;

import static java.lang.System.out;

public class StateYourTerms {

    public static void main(String[] args) {

        var keyboard = new Scanner(System.in);
        String month;
        int numberOfDays = 0;
        boolean isLeapYear;

        out.print("Which month? ");
        month = keyboard.next();

        switch (month) {
            case "January", "March", "May", "July",
```

(continued)

LISTING 17-3: ***(continued)***

```
                    "August", "October", "December" -> numberOfDays = 31;
            case "April", "June", "September", "November" -> numberOfDays = 30;
            case "February" -> {
                out.print("Leap year (true/false)? ");
                isLeapYear = keyboard.nextBoolean();
                if (isLeapYear) {
                    numberOfDays = 29;
                } else {
                    numberOfDays = 28;
                }
            }
        }

        out.print(numberOfDays);
        out.println(" days");

        keyboard.close();
    }
}
```

Figure 17-3 shows four runs of the code in Listing 17-3.



```
Which month? December
31 days

Which month? February
Leap year (true/false)? true
29 days

Which month? February
Leap year (true/false)? false
28 days

Which month? Intercalaris
0 days
```

FIGURE 17-3: What the heck is Intercalaris?

The `"February"` clause in Listing 17-3 has a life of its own. In Chapter 10, I nest `if` statements within other `if` statements. But in this `"February"` clause, I nest an `if` statement within a `switch` statement. That's cool.

Maybe the `switch` statement in Listing 17-3 doesn't make you giddy with glee. In that case, you can check out the `switch` expression in Listing 17-4. The code in

Listing 17-4 has exactly the same behavior as the program in Listing 17-3. You can see some runs earlier, in Figure 17-3.

LISTING 17-4: From Statements Come Values

```
import java.util.Scanner;

import static java.lang.System.out;

public class ExpressYourFeelings {

    public static void main(String[] args) {

        var keyboard = new Scanner(System.in);
        String month;
        int numberOfDays = 0;
        boolean isLeapYear;

        out.print("Which month? ");
        month = keyboard.next();

        out.println(switch (month) {
            case "January", "March", "May", "July",
                    "August", "October", "December" -> 31;
            case "April", "June", "September", "November" -> 30;
            case "February" -> {
                out.print("Leap year (true/false)? ");
                isLeapYear = keyboard.nextBoolean();
                if (isLeapYear) {
                    yield 29;
                } else {
                    yield 28;
                }
            }
            default -> 0;
        } + " days");

        keyboard.close();
    }
}
```

Here's a quote from a wise old man:

> "In a switch expression, each case clause must have a value."

The old man forgot to add that a switch expression's case clause may include statements that have no values. For example, in Listing 17-4, the line out.

print("Leap year (true/false)? ") has no value. How do you mix out.println calls and numeric values (29 or 28) in a single case clause?

The answer is, you use a yield statement. A yield statement tells Java to end the execution of any statements in the current case clause. In addition, the yield statement assigns a value (such as 29 or 28) to the entire case clause. When all is said and done, the value of the entire switch expression becomes either 29 or 28, and Java adds that value to the " days" string. It's as if the code for February 2022 looked like this:

```
            switch (month) {
    ...
    case "February" -> 28;
    ...
} + " days"
```

Java's newest switch facilities are quite versatile. But Java existed for 24 years without having these fancy switch features. For years to come, many programmers will continue to use the old form of the switch statement, so the next section covers that older form.

TRY IT OUT

Redo the examples named Days of the Week in the United States and A Tiny Calculator from the previous Try It Out section. This time, use switch expressions with yield statements.

Your Grandparents' switch Statement

In versions of Java before Java 14, the code in Listings 17-1 through 17-4 would have failed. That's because Java's switch statement came originally from the switch statement in the C/C++ language family, and the C/C++ switch statement was quite clunky. The code in Listing 17-5 does the same thing as the code in Listings 17-1 and 17-2, but the Listing 17-5 code works in all versions of Java — old and new.

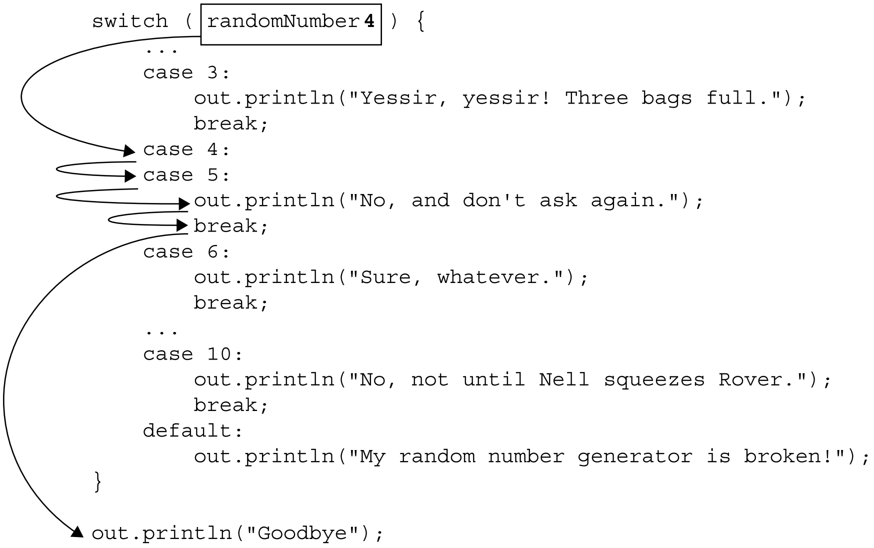
LISTING 17-5: A la recherche du temps perdu

```
import java.util.Random;
import java.util.Scanner;

import static java.lang.System.out;

public class TheVeryOldSwitcheroo {
    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        var myRandom = new Random();
        int randomNumber;

        out.print("Type your question, my child: ");
        keyboard.nextLine();
        randomNumber = myRandom.nextInt(10) + 1;

        switch (randomNumber) {
            case 1:
                out.println("Yes. Isn't it obvious?");
                break;
            case 2:
                out.println("What part of 'no' don't you understand?");
                break;
            case 3:
                out.println("Yessir, yessir! Three bags full.");
                break;
            case 4:
            case 5:
                out.println("No, and don't ask again.");
                break;
            case 6:
                out.println("Sure, whatever.");
                break;
            case 7:
                out.println("Yes, but only if you're nice to me.");
                break;
            case 8:
                out.print("Yes (as if I care).");
                out.println(" Next time, take your problem somewhere else.");
                break;
            case 9:
                out.println("No, not until Cromwell seizes Dover.");
                break;
            case 10:
                out.println("No, not until Nell squeezes Rover.");
                break;
            default:
                out.println("My random number generator is broken!");
        }

        out.println("Goodbye");

        keyboard.close();
    }
}
```

When you run the code in Listing 17-5, you see the same old stuff — the output in Figure 17-1. But compare Listing 17-5 with Listings 17-1 and 17-2. This new Listing 17-5 has much more baggage. To get the same results with `randomNumber` values 4 and 5, you have to write

```
case 4:
case 5:
    out.println("No, and don't ask again.");
    break;
```

The reason for this is a feature called *fall-through*. With this older form of the `switch` statement, execution falls through case after case until it encounters a `break` statement. For example, when `randomNumber` is 4, the computer follows the path shown in Figure 17-4.



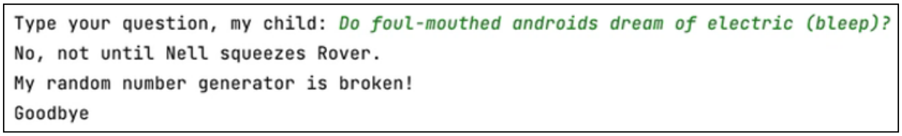
```
switch ( randomNumber 4 ) {
     ...
     case 3:
         out.println("Yessir, yessir! Three bags full.");
         break;
     case 4:
     case 5:
         out.println("No, and don't ask again.");
         break;
     case 6:
         out.println("Sure, whatever.");
         break;
     ...
     case 10:
         out.println("No, not until Nell squeezes Rover.");
         break;
     default:
         out.println("My random number generator is broken!");
}

out.println("Goodbye");
```

FIGURE 17-4: Follow the bouncing ball.

The `break` statement tells the computer, "Go directly to whatever statement comes after the end of the `switch` statement. Do not pass `default`. Do not print `My random number generator is broken!`"

You have to know about this older form of the `switch` statement because you find it in so many Java programs, but I don't recommend writing new code this way. Besides being horribly verbose, the use of `break` statements is an endless source of errors.

Even experienced Java programmers occasionally forget to include break statements. While I was preparing Listing 17-5 for inclusion in this book, I accidentally omitted the break statement in case 10. Figure 17-5 shows you a run of the buggy code.



```
Type your question, my child: Do foul-mouthed androids dream of electric (bleep)?
No, not until Nell squeezes Rover.
My random number generator is broken!
Goodbye
```

FIGURE 17-5: Is it 10? Is it some unexpected number? Please make up your mind.

The worst part about break statements is that they're easy to miss. When you omit a break statement, IntelliJ doesn't complain. You get no squiggly, red lines, no honking horns, no wagging fingers. The code runs just fine until you stumble on a case clause that's missing its break statement. If you blink and don't catch the error, that error stays in your code.

REMEMBER

If you use the older form of Java's switch statement, add break statements wherever you don't want fall-through to occur.

TIP

For the sake of overall tidiness, some programmers end their old switch statements with break statements. For example, they'd end the switch statement in Listing 17-5 like this:

```
    default:
        out.println("My random number generator is broken!");
        break;
}
```

If it makes you feel better, you can do it too.

TRY IT OUT

Get some practice using Ye Olde Java switch statement.

DON'T DO THIS AT HOME (OR ANYWHERE ELSE)

What's wrong with the following code, and how can you fix it?

```
switch (amount) {
case 1:
    System.out.println("US cent");
case 5:
    System.out.println("US nickel");
case 10:
```

```
            System.out.println("US dime");
        case 25:
            System.out.println("US quarter");
        case 50:
            System.out.println("US half dollar");
        case 100:
            System.out.println("US dollar");
        default:
            System.out.println("Not a US coin");
        }
```

ROUGHING IT

Rewrite Listing 17-3 so that it runs with an older version of Java — Java 11, for example.

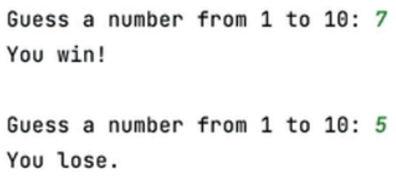
Using a Conditional Operator

Having read this chapter's description of the `switch` expression, you may ask, "Does Java have an *if* expression?" And the answer (whether you asked the question or not) is yes. But Java doesn't call it an *if* expression. Java calls it a *conditional operator*. The conditional operator works in every version of Java — old and new.

Listing 17-6 contains a tiny guessing game program, and Figure 17-6 shows two runs of the program.

LISTING 17-6: One Good Ternary Deserves Another

```
import java.util.Random;
import java.util.Scanner;

public class GuessingGame {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        var myRandom = new Random();
        int randomNumber;
        int guess;
        String reply;

        randomNumber = myRandom.nextInt(10) + 1;
        System.out.print("Guess a number from 1 to 10: ");
        guess = keyboard.nextInt();

        reply = (guess == randomNumber) ? "You win!" : "You lose.";
        System.out.println(reply);

        keyboard.close();
    }
```



```
Guess a number from 1 to 10: 7
You win!

Guess a number from 1 to 10: 5
You lose.
```

FIGURE 17-6: Everyone's a winner (almost).

```
}
```

Taken as a whole, `(guess == randomNumber) ? "You win!" : "You lose."` is an expression with a value. And what value does this expression have? Well, the value of `(guess == randomNumber) ? "You win!" : "You lose."` is either `"You win!"` or `"You lose."` It depends on whether `guess == randomNumber` is or isn't `true`.

That's how the conditional operator works:

- If the stuff before the question mark is `true`, the whole expression's value is whatever comes between the question mark and the colon.
- If the stuff before the question mark is `false`, the whole expression's value is whatever comes after the colon.

In Listing 17-6, the conditional operator's overall effect is as though the computer is executing

```
reply = "You win!";
```

or

```
reply = "You lose.";
```

To reinforce these ideas, Figure 17-7 provides a rare glimpse into the mind of a conditional operator as it figures out what its value is.

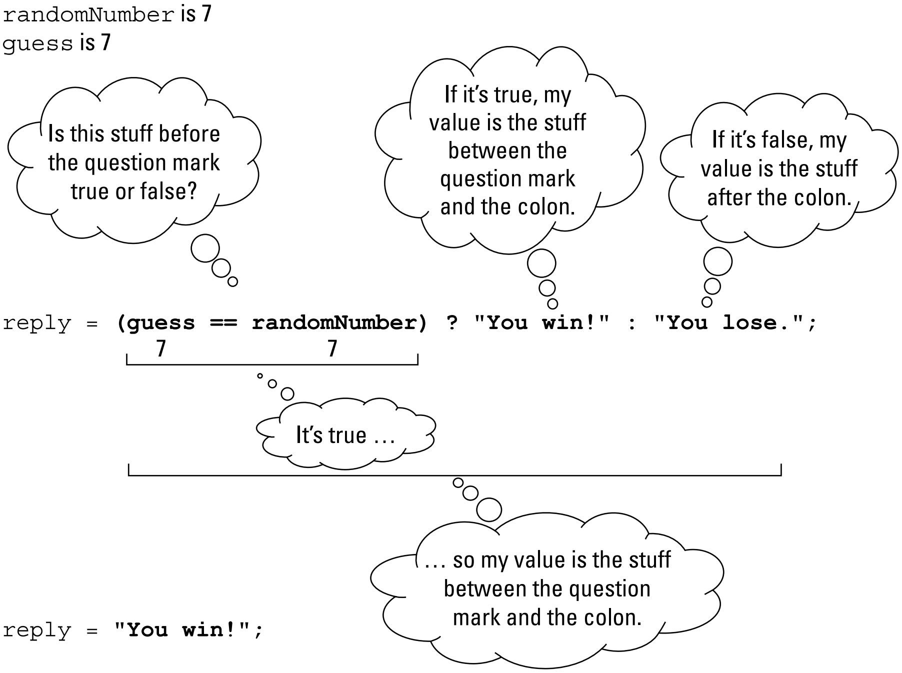
TRY IT OUT

Get some practice using Java's conditional operator.

DRESSED TO THE NINES

Listing 17-6 borrows an idea from one of the Try It Out programs in Chapter 9. Rewrite some of the chapter's other Try It Out programs using conditional operators:

- Ask the user if they want to see a smiley face. Display `:-)` if the user replies `Y`; display `:-(` otherwise.
- Read a number of meters from the keyboard. Then read a letter (either `c` or `m`) from the keyboard. If the letter is `c`, convert meters to centimeters and display the result. If the letter is `m`, convert meters to millimeters and display the result. (Assume that the user always types either the letter `c` or the letter `m`.)
- Read a number of meters from the keyboard. Then read a letter from the keyboard. If the letter is `c`, convert meters to centimeters and display the result. If the letter is `m`, convert meters to millimeters and display the result. For any other letters, display the original number of meters.



`randomNumber` is 7
`guess` is 7

```
reply = (guess == randomNumber) ? "You win!" : "You lose.";
```

```
reply = "You win!";
```

Is this stuff before the question mark true or false?
If it's true, my value is the stuff between the question mark and the colon.
If it's false, my value is the stuff after the colon.
7
7
It's true …
… so my value is the stuff between the question mark and the colon.

FIGURE 17-7: Have you ever seen an expression talking to itself?

IN THIS CHAPTER

- Analyzing loop strategies
- Diagnosing loop problems
- Creating nested loops

Chapter 18

Creating Loops within Loops

If you're an editor at Wiley Publishing, please don't read the next few paragraphs. In the next few paragraphs, I give away an important trade secret (something you really don't want me to do).

I'm about to describe a surefire process for writing a best-selling *For Dummies* book. Here's the process:

Write several words to create a sentence. Do this several times to create a paragraph:

```
Repeat the following to form a paragraph:
   Repeat the following to form a sentence:
      Write a word.
```

Repeat these instructions several times to make a section. Make several sections and then make several chapters:

```
Repeat the following to form a best-selling book in the For Dummies series:
   Repeat the following to form a chapter:
      Repeat the following to form a section:
         Repeat the following to form a paragraph:
```

```
        Repeat the following to form a sentence:
          Write a word.
```

This process involves a loop within a loop within a loop within a loop within a loop. It's like a verbal M.C. Escher print. Is it useful, or is it frivolous?

Well, in the world of computer programming, this kind of thing happens all the time. Most five-layered loops are hidden behind method calls, but two-layered loops within loops are everyday occurrences. So this chapter tells you how to compose a loop within a loop. It's very useful stuff.

By the way, if you're a Wiley editor, you can start reading again from this point onward.

Paying Your Old Code a Little Visit

The program in Listing 11-5 (over in Chapter 11) extracts a username from an email address. For example, the program reads

```
John@BurdBrain.com
```

from the keyboard and then writes

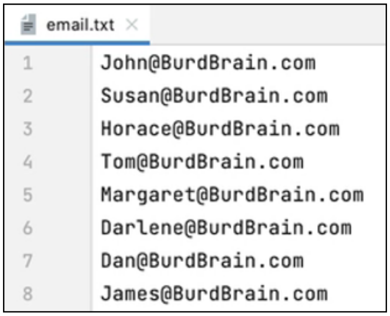
```
John
```

to the screen. Let me tell you, in this book I have some pretty lame excuses for writing programs, but this simple email example tops the list! Why would you want to type something on the keyboard, only to have the computer display *part* of what you typed? There must be a better use for code of this kind.

Sure enough, there is. The BurdBrain.com email system administrator has a list of 10,000 employees' email addresses. More precisely, the administrator's hard drive has a file named `email.txt`. This file contains 10,000 email addresses, with a single address on each line, as shown in Figure 18-1.

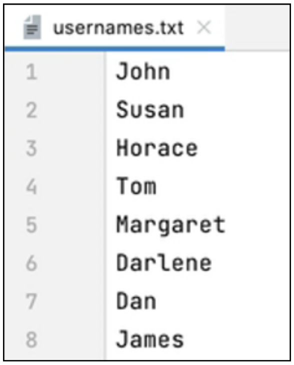
The company's email software has an interesting feature. To send email within the company, you don't need to type an entire email address. For example, to send email to John, you can type the username `John` instead of `John@BurdBrain.com`. (This `@BurdBrain.com` part is called the *hostname.*)

The company's email administrator wants to distill the content of the `email.txt` file. She wants a new file, `usernames.txt`, that contains usernames with no hostnames, as shown in Figure 18-2.



```
John@BurdBrain.com
Susan@BurdBrain.com
Horace@BurdBrain.com
Tom@BurdBrain.com
Margaret@BurdBrain.com
Darlene@BurdBrain.com
Dan@BurdBrain.com
James@BurdBrain.com
```

FIGURE 18-1: A list of email addresses.



```
John
Susan
Horace
Tom
Margaret
Darlene
Dan
James
```

FIGURE 18-2: Usernames extracted from the list of email addresses.

Reworking some existing code

To solve the administrator's problem, you need to modify the code in Listing 11-5. The new version gets an email address from a disk file and writes a username to another disk file. The new version is in Listing 18-1.

LISTING 18-1: From One File to Another

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.util.Scanner;

public class ListOneUsername {

    public static void main(String[] args) throws FileNotFoundException {

        var diskScanner = new Scanner(new File("email.txt"));
        var diskWriter = new PrintStream("usernames.txt");
        char symbol;
```

(continued)

LISTING 18-1: ***(continued)***

```
        symbol = diskScanner.findWithinHorizon(".", 0).charAt(0);

        while (symbol != '@') {
            diskWriter.print(symbol);
            symbol = diskScanner.findWithinHorizon(".", 0).charAt(0);
        }
        diskWriter.println();

        diskScanner.close();
        diskWriter.close();
    }
}
```

Listing 18-1 does almost the same thing as its forerunner in Listing 11-5. The only difference is that the code in Listing 18-1 doesn't interact with the user. Instead, the code in Listing 18-1 interacts with disk files.

Running your code

Here's how you run the code in Listing 18-1:

1. **Create a file named `email.txt` in your IntelliJ project directory.**

   In the `email.txt` file, put just one email address. Any address will do, as long as the address contains an at-sign (@).

2. **Put the `ListOneUsername.java` file (the code from Listing 18-1) in your project's `src` directory.**

3. **Run the code in Listing 18-1.**

   When you run the code, you see nothing interesting in the Run tool window. What a pity!

4. **View the contents of the `usernames.txt` file.**

   If your `email.txt` file contains `John@BurdBrain.com`, the `usernames.txt` file contains `John`.

For more details on any of these steps, refer to Chapter 16.

Nested Development

The previous section describes an email system administrator's problem — creating a file filled with usernames from a file filled with email addresses. The code in Listing 18-1 solves part of the problem — it extracts just one email address. That's a good start, but to get just one username, you don't need a computer program. A pencil and some paper do the trick.

Don't keep the email administrator waiting any longer. In this section, you develop a program that processes dozens, hundreds, and even thousands of email addresses from a file on your hard drive.

First, you need a strategy to create the program. Take the statements in Listing 18-1 and run them over and over again. Better yet, have the statements run themselves over and over again. Fortunately, you already know how to do something over and over again: You use a loop. (See Chapter 11 for the basics on loops.)

Here's the strategy: Take the statements in Listing 18-1 and enclose them in a larger loop:

```
while (not at the end of the email.txt file) {
    Execute the statements in Listing 18-1
}
```

Looking back at the code in Listing 18-1, you see that the statements in that code have a `while` loop of their own. So this strategy involves putting one loop inside another loop:

```
while (not at the end of the email.txt file) {
    //Blah-blah

    while (symbol != '@') {
        //Blah-blah-blah
    }

   //Blah-blah-blah-blah
}
```

Because one loop is inside the other, they're called *nested loops.* The old loop (the `symbol != '@'` loop) is the *inner loop.* The new loop (the end-of-file loop) is called the *outer loop.*

Checking for the end of a file

Now all you need is a way to test the loop's condition. How do you know when you're at the end of the `email.txt` file?

The answer comes from Java's `Scanner` class. This class's `hasNext` method answers `true` or `false` to the following question:

> Does the `email.txt` file have anything to read in it (beyond what you've already read)?

If the program's `findWithinHorizon` calls haven't gobbled up all the characters in the `email.txt` file, the value of `diskScanner.hasNext()` is `true`. So, to keep looping while you're not at the end of the `email.txt` file, you do the following:

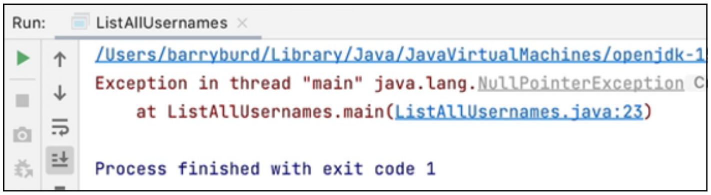
```
while (diskScanner.hasNext()) {
    Execute the statements in Listing 18-1
}
```

The first realization of this strategy is in Listing 18-2.

LISTING 18-2: The Mechanical Combining of Two Loops

```
/*
 * This code does NOT work (but you learn from your mistakes).
 */

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.util.Scanner;

public class ListAllUsernames {

    public static void main(String[] args) throws FileNotFoundException {

        var diskScanner = new Scanner(new File("email.txt"));
        var diskWriter = new PrintStream("usernames.txt");
        char symbol;

        while (diskScanner.hasNext()) {
            symbol = diskScanner.findWithinHorizon(".", 0).charAt(0);

            while (symbol != '@') {
                diskWriter.print(symbol);
                symbol = diskScanner.findWithinHorizon(".", 0).charAt(0);
            }
            diskWriter.println();
        }

        diskScanner.close();
        diskWriter.close();
    }
}
```

When you run the code in Listing 18-2, you get the disappointing response shown in Figure 18-3.



```
/Users/barryburd/Library/Java/JavaVirtualMachines/openjdk-1
Exception in thread "main" java.lang.NullPointerException C
    at ListAllUsernames.main(ListAllUsernames.java:23)

Process finished with exit code 1
```

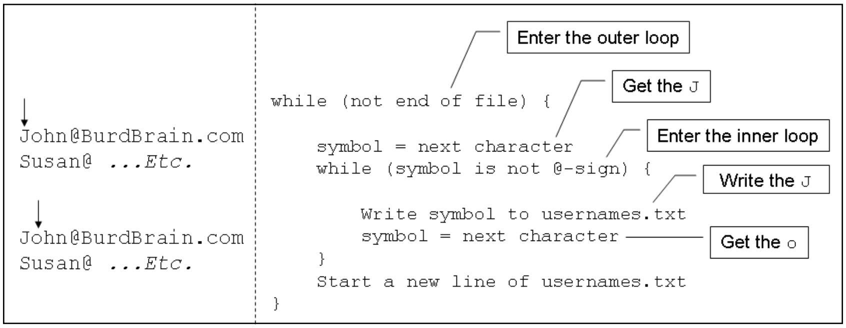
FIGURE 18-3: You goofed.

How it feels to be a computer

What's wrong with the code in Listing 18-2? To find out, I role-play the computer. "If I were a computer, what would I do when I execute the code in Listing 18-2?"

The first several things that I'd do are pictured in Figure 18-4. I would read the J in John, and then write the J in John, and then read the letter o (also in John).

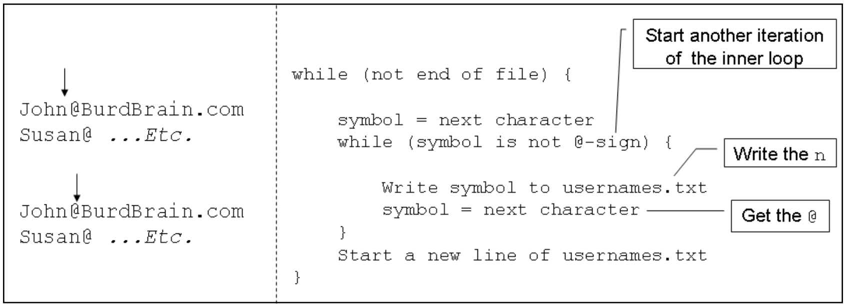
After a few trips through the inner loop, I'd get the @ sign in John@BurdBrain.com, as shown in Figure 18-5.



```
John@BurdBrain.com
Susan@ ...Etc.

John@BurdBrain.com
Susan@ ...Etc.
```

```
while (not end of file) {

    symbol = next character
    while (symbol is not @-sign) {

        Write symbol to usernames.txt
        symbol = next character
    }
    Start a new line of usernames.txt
}
```

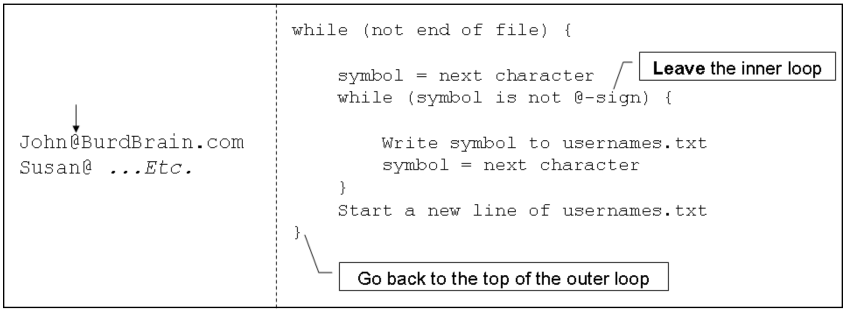
FIGURE 18-4: Role-playing the code in Listing 18-2.



```
while (not end of file) {

    symbol = next character
    while (symbol is not @-sign) {

        Write symbol to usernames.txt
        symbol = next character
    }
    Start a new line of usernames.txt
}
```

FIGURE 18-5: Reaching the end of the username.

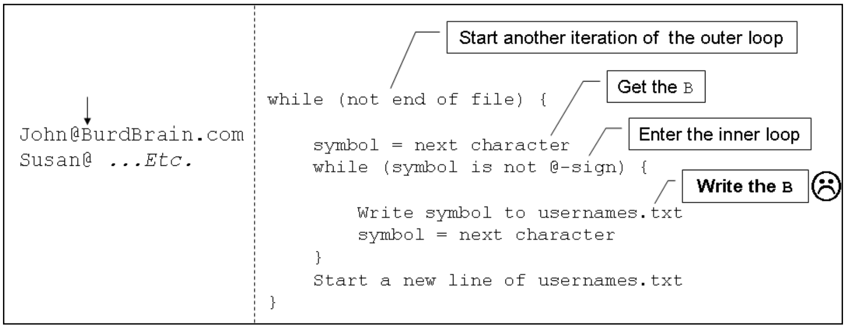
Finding this @ sign would jump me out of the inner loop and back to the top of the outer loop, as shown in Figure 18-6.



```
while (not end of file) {

    symbol = next character
    while (symbol is not @-sign) {

        Write symbol to usernames.txt
        symbol = next character
    }
    Start a new line of usernames.txt
}
```

FIGURE 18-6: Leaving the inner loop.

I'd get the B in BurdBrain and sail back into the inner loop. But then (horror of horrors!) I'd write that B to the usernames.txt file. (See Figure 18-7.)



```
while (not end of file) {

    symbol = next character
    while (symbol is not @-sign) {

        Write symbol to usernames.txt
        symbol = next character
    }
    Start a new line of usernames.txt
}
```

FIGURE 18-7: The error of my ways.

There's the error! You don't want to write hostnames to the usernames.txt file. When the computer found the @ sign, it should have skipped past the rest of John's email address.

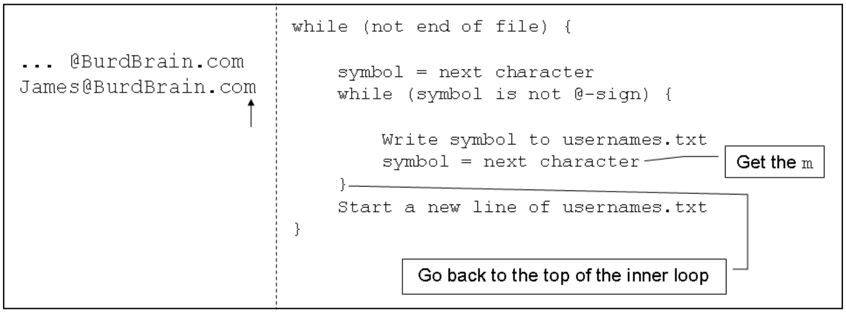
At this point, you have a choice: You can jump straight to the corrected code in Listing 18-3 (a couple of sections from here), or you can read on to find out about the error message in Figure 18-3.

Why the computer accidentally pushes past the end of the file

Ah! You're wondering why Figure 18-3 has that nasty error message.

I role-play the computer to help me figure out what's going wrong. Imagine that I've already role-played the steps in Figure 18-7. I shouldn't process the first letter `B` (let alone the entire `BurdBrain.com` hostname) with the inner loop. But, unfortunately, I do.

I keep running and processing more email addresses. When I get to the end of the last email address, I grab the `m` in `BurdBrain.com` and go back to test for an @ sign, as shown in Figure 18-8.



```
... @BurdBrain.com
James@BurdBrain.com
```

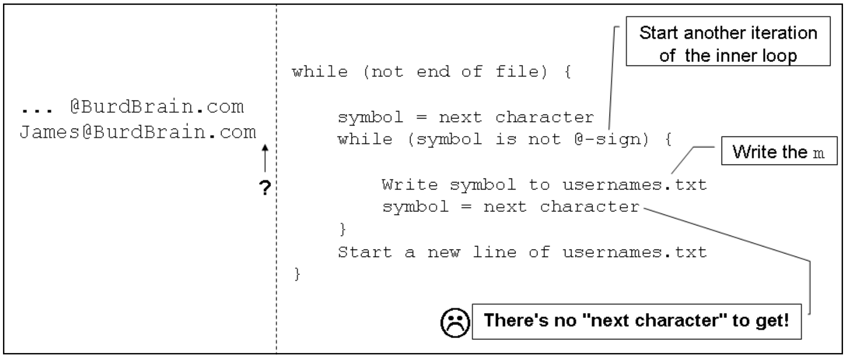
```
while (not end of file) {

    symbol = next character
    while (symbol is not @-sign) {

        Write symbol to usernames.txt
        symbol = next character
    }
    Start a new line of usernames.txt
}
```

FIGURE 18-8: The journey's last leg.

Now I'm in trouble. This last `m` certainly isn't an @ sign. So I jump into the inner loop and try to get yet another character. (See Figure 18-9.) The `email.txt` file has no more characters, so Java sends an error message to the computer screen. (Refer to the `NullPointerException` error message in Figure 18-3.)

TECHNICAL STUFF

Here's why I get a `NullPointerException`: The `email.txt` file has no more characters, so the call to `findWithinHorizon(".", 0)` comes up empty. (There's nothing to find.) In Java, a more precise way of describing that emptiness is with the word `null`. The call `findWithinHorizon(".", 0)` is `null`, so pointing to a character that was found (`charAt(0)`) is a fruitless endeavor. Thus, Java displays a `NullPointerException` message.



```
... @BurdBrain.com
James@BurdBrain.com
```

```
while (not end of file) {

    symbol = next character
    while (symbol is not @-sign) {

        Write symbol to usernames.txt
        symbol = next character
    }
    Start a new line of usernames.txt
}
```

?
Start another iteration of the inner loop
Write the m
There's no "next character" to get!

FIGURE 18-9: Trying to read past the end of the file.

Loop therapy

Listing 18-3 has the solution to the problem described with Figures 18-1 and 18-2. The code in this listing is almost identical to the code in Listing 18-2. The only difference is the added call to `nextLine`. When the computer reaches an @ sign, this `nextLine` call swallows the rest of the input line without actually tasting it. (The `nextLine` call gets the rest of the email address but doesn't output that part of the email address. The idea works because each email address is on its own, separate line.) After gulping down `@BurdBrain.com`, the computer moves gracefully to the next line of input.

LISTING 18-3: That's Much Better!

```
/*
 * This code is correct!!
 */

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.util.Scanner;

public class ListAllUsernames {

    public static void main(String[] args) throws FileNotFoundException {

        var diskScanner = new Scanner(new File("email.txt"));
        var diskWriter = new PrintStream("usernames.txt");
        char symbol;

        while (diskScanner.hasNext()) {
            symbol = diskScanner.findWithinHorizon(".", 0).charAt(0);
            while (symbol != '@') {
                diskWriter.print(symbol);
                symbol = diskScanner.findWithinHorizon(".", 0).charAt(0);
            }

            diskScanner.nextLine();
            diskWriter.println();
        }

        diskScanner.close();
        diskWriter.close();
    }
}
```

To run the code in Listing 18-3, you need an `email.txt` file — a file like the one shown earlier, in Figure 18-1. In the `email.txt` file, type several email addresses. Any addresses will do, as long as each address contains an @ sign and each address is on its own separate line. Save the `email.txt` file in your project directory along with the `ListAllUsernames.java` file (the code from Listing 18-3). For more details, refer to Chapter 16.

With Listing 18-3, you've reached an important milestone. You've analyzed a delicate programming problem and found a complete, working solution. The tools you used included thinking about strategies and role-playing the computer. As time goes on, you can use these tools to solve bigger and better problems.

TRY IT OUT

Nothing is more challenging for novice programmers than creating nested loops. That's why I cover looping techniques in four of this book's chapters (Chapters 11, 12, 18, and 19). You might say that I loop through my coverage of programming loops.

Try writing the code that I suggest in the next few paragraphs. Don't be afraid to make lots of mistakes. If you get stuck, slow down, take a step back, and think about what the computer will do when it follows instructions to the letter.

The solutions are on my web page at `http://beginprog.allmycode.com`. But don't jump to the solutions until you've experimented with lots of different ideas. Follow this tried-and-true formula:

```
Write some code;
Run your code;
while (your program doesn't work correctly) {
    Step through your code, one statement after another, keeping track
         of the values of the variables and the computer's output as
         Java follows your instructions exactly as they're written;
    In the step by step execution of statements, notice the place where
         Java does something that you don't want it to do;
    Ask yourself how you'd change the statements so that Java would do
         what you want it to do;
    Change the statements in your code;
    Run your code again;
}
```

END OF THE ROAD

A file named `input.txt` contains only four characters:

```
Java
```

What’s the output when you run the following code?

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) throws FileNotFoundException {
        var diskScanner = new Scanner(new File("input.txt"));

        while (diskScanner.hasNext()) {
            char symbol = diskScanner.findWithinHorizon(".", 0).charAt(0);
            System.out.print(Character.toUpperCase(symbol));
        }

        diskScanner.close();
    }

}
```

WHAT’S IN THE STARS?

In this chapter’s earlier “How it feels to be a computer” section, I examine each line of a program’s code and ask myself what the computer does when it executes that line. Do the same thing with the following program:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
```

```
public class ReadStars {

    public static void main(String[] args) throws FileNotFoundException {

        var diskScanner = new Scanner(new File("input.txt"));
        char symbol;

        while (diskScanner.hasNext()) {
            symbol = diskScanner.findWithinHorizon(".", 0).charAt(0);

            while (symbol == '*') {
                System.out.print(symbol);
                symbol = diskScanner.findWithinHorizon(".", 0).charAt(0);
            }

            System.out.println();
        }

        diskScanner.close();
    }
}
```

What happens when the `input.txt` file contains the following characters?

```
*****x***y*****z
```

LOOP SOUP

Make a chart to keep track of the changes to the values of `i` and `j` as Java executes the following code:

```
int i = 5;
int j;
while (i > 0) {
    System.out.println(i);
    i--;
    j = 3;
    while (j > 0) {
        System.out.print(j);
        j--;
    }
    System.out.println();
}
```

Based on the values in your chart, what will be the output of the code? After making your chart, run the code in IntelliJ IDEA to find out whether your prediction is correct.

MAKE SOME CHANGES AROUND HERE

Modify the code from the previous experiment (the "Loop soup" one) so that the output has no lines containing the `321` digit sequence. In place of the `321` lines, the output has lines containing the `123` digit sequence.

SEEING STARS

This experiment comes in two parts. The first part requires only one loop; the second part requires nested loops:

- Write a program that asks the user how many stars to display. When the user enters a number, the program displays that many stars. Here's a sample run:

  ```
  How many stars? 5
  *****
  ```

- Write a program that repeatedly asks whether the user wants to see a row of stars. As long as the user replies with the letter `y`, the program does what it did in the previous bullet. That is, the program asks the user how many stars to display and then displays that many stars. As soon as the user replies with the letter `n`, the program stops running.

  Here's a sample run of the program:

  ```
  Do you want a row of stars? (y/n) y
  How many stars? 5
  *****
  Do you want a row of stars? (y/n) y
  How many stars? 2
  **
  Do you want a row of stars? (y/n) y
  How many stars? 8
  ********
  Do you want a row of stars? (y/n) n
  ```

  To create this program, take the code that you wrote in the previous bullet (the first part of "Seeing stars") and surround some of that code inside a second loop.

APROPOS OF NOTHING

In Figure 18-3, the run of a Java program throws a `NullPointerException`. These `NullPointerException` messages are never fun, but the more of these messages you encounter, the less frightening they are.

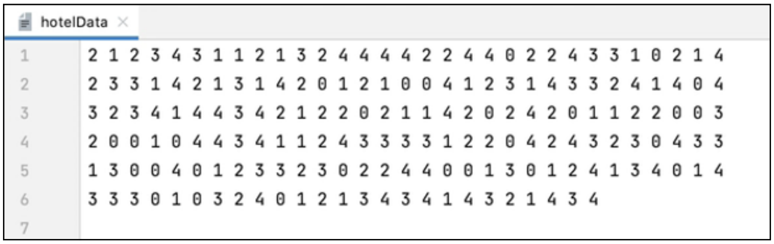
To help desensitize you to `NullPointerException` messages, generate one of them intentionally. Run the following two-line code snippet with IntelliJ's JShell console:

```
String name = null;
System.out.println(name.length());
```

Using Nested for Loops

Because you're reading *Beginning Programming with Java For Dummies,* 6th Edition, I assume that you manage a big-name hotel. Chapter 19 tells you everything you need to know about hotel management. But before you begin reading that chapter, you can get a little preview in this section.

I happen to know that your hotel has nine floors, and that each floor of your hotel has 20 rooms. On this sunny afternoon, someone hands you a flash drive containing a file full of numbers. You copy this `hotelData` file to your hard drive and then display the file in IntelliJ's editor. You see the stuff shown in Figure 18-10.

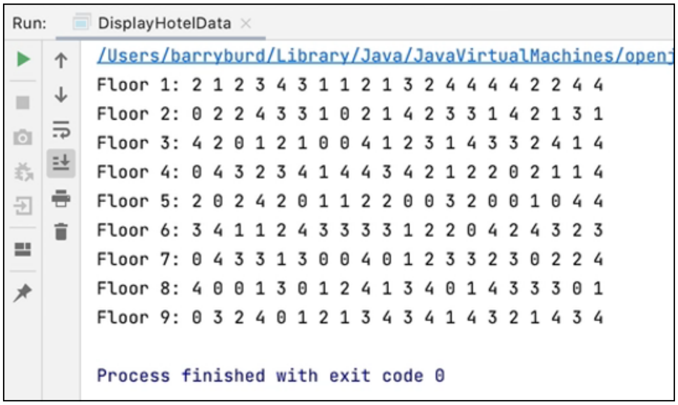


```
2 1 2 3 4 3 1 1 2 1 3 2 4 4 4 4 2 2 4 4 0 2 2 4 3 3 1 0 2 1 4
2 3 3 1 4 2 1 3 1 4 2 0 1 2 1 0 0 4 1 2 3 1 4 3 3 2 4 1 4 0 4
3 2 3 4 1 4 4 3 4 2 1 2 2 0 2 1 1 4 2 0 2 4 2 0 1 1 2 2 0 0 3
2 0 0 1 0 4 4 3 4 1 1 2 4 3 3 3 3 1 2 2 0 4 2 4 3 2 3 0 4 3 3
1 3 0 0 4 0 1 2 3 3 2 3 0 2 2 4 4 0 0 1 3 0 1 2 4 1 3 4 0 1 4
3 3 3 0 1 0 3 2 4 0 1 2 1 3 4 3 4 1 4 3 2 1 4 3 4
```

FIGURE 18-10: A file containing hotel occupancy data.

This file gives the number of guests in each room. For example, at the start of the file, you see `2 1 2`. This means that, on the first floor, Room 1 has two guests, Room 2 has one guest, and Room 3 has two guests. After reading 20 of these numbers, you see `0 2 2`. So, on the second floor, Room 1 has zero guests, Room 2 has two guests, and Room 3 has two guests. The story continues until the last number in the file. According to that number, Room 20, on the ninth floor, has four guests.

You'd like a more orderly display of these numbers — a display of the kind in Figure 18-11. So you whip out your keyboard to write a quick Java program.



```
/Users/barryburd/Library/Java/JavaVirtualMachines/openj
Floor 1: 2 1 2 3 4 3 1 1 2 1 3 2 4 4 4 4 2 2 4 4
Floor 2: 0 2 2 4 3 3 1 0 2 1 4 2 3 3 1 4 2 1 3 1
Floor 3: 4 2 0 1 2 1 0 0 4 1 2 3 1 4 3 3 2 4 1 4
Floor 4: 0 4 3 2 3 4 1 4 4 3 4 2 1 2 2 0 2 1 1 4
Floor 5: 2 0 2 4 2 0 1 1 2 2 0 0 3 2 0 0 1 0 4 4
Floor 6: 3 4 1 1 2 4 3 3 3 3 1 2 2 0 4 2 4 3 2 3
Floor 7: 0 4 3 3 1 3 0 0 4 0 1 2 3 3 2 3 0 2 2 4
Floor 8: 4 0 0 1 3 0 1 2 4 1 3 4 0 1 4 3 3 3 0 1
Floor 9: 0 3 2 4 0 1 2 1 3 4 3 4 1 4 3 2 1 4 3 4

Process finished with exit code 0
```

FIGURE 18-11: A readable display of the data in Figure 18-10.

As in some other examples, you decide which statements go where, by asking yourself how many times each statement should be executed. For starters, the display in Figure 18-11 has nine lines, and each line has 20 numbers:

```
for (each of 9 floors)
    for (each of 20 rooms on a floor)
        get a number from the file and display the number on the screen.
```

So your program has a `for` loop within a `for` loop — a pair of *nested* `for` loops.

Next, you look at in Figure 18-11 and notice how each line begins. Each line contains the word `Floor`, followed by the floor number. Because this `Floor` display occurs only nine times in Figure 18-11, the statements to print this display belong in the for-each-of-9-floors loop (and not in the for-each-of-20-rooms loop). The statements should come before the for-each-of-20-rooms loop because this `Floor` display comes once before each line's 20-number display:

```
for (each of 9 floors)
    display "Floor" and the floor number,
    for (each of 20 rooms on a floor)
        get a number from the file and display the number on the screen.
```

You're almost ready to write the code. But there's one detail that's easy to forget. (Well, it's a detail that I always forget.) After displaying 20 numbers, the program advances to a new line. This new-line action happens only nine times during the run of the program, and it always happens *after* the program displays 20 numbers:

```
for (each of 9 floors)
    display "Floor" and the floor number,
    for (each of 20 rooms on a floor)
        get a number from the file and display the number on the screen,
    Go to the next line.
```

That does it. That's all you need. The code to create the display of Figure 18-11 is in Listing 18-4.

LISTING 18-4: Hey! Is This a For-by-For?

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

import static java.lang.System.out;

class DisplayHotelData {

    public static void main(String[] args) throws FileNotFoundException {

        var diskScanner = new Scanner(new File("hotelData"));

        for (int floor = 1; floor <= 9; floor++) {
            out.print("Floor ");
            out.print(floor);
            out.print(": ");

            for (int roomNum = 1; roomNum <= 20; roomNum++) {
                out.print(diskScanner.nextInt());
                out.print(' ');
            }

            out.println();
        }
        diskScanner.close();
    }
}
```

The code in Listing 18-4 has the variable `floor` going from 1 to 9 and has the variable `roomNum` going from 1 to 20. Because the `roomNum` loop is inside the `floor` loop, the writing of 20 numbers happens nine times. That's good. It's exactly what I want.

TRY IT OUT

When it comes to writing code with loops, there's no such thing as having too much practice. Try these problems. Work slowly and don't get discouraged. Remember that solutions are available at `http://beginprog.allmycode.com`.

MYSTERY CODE

This experiment comes in two parts:

- Without running the following code, try to predict the code's output:

  ```
  for (int row = 0; row < 5; row++) {
      for (int column = 0; column < 5; column++) {
          System.out.print("*");
      }
      System.out.println();
  }
  ```

  After making your prediction, run the code to find out whether your prediction is correct.

- The code in this bullet is a slight variation on the code in the previous bullet. First, try to predict what the code will output. Then run the code to find out whether your prediction is correct:

  ```
  for (int row = 0; row < 5; row++) {
      for (int column = 0; column <= row; column++) {
          System.out.print("*");
      }
      System.out.println();
  }
  ```

DRAW A PATTERN

This experiment comes in four parts:

- Write a program that reads a number from the keyboard. The program uses a `for` loop to display that number of dashes.

  For example, if the user types the number 5, the program displays

  ```
  -----
  ```

- Modify the program you wrote in the previous bullet. The modified program uses two `for` loops to display two lines of characters. The second line is one character shorter than the first line. For example, if the user types the number 7, the program displays

```
-------
------
```

- Modify the program you wrote in the previous bullet. The modified program uses nested `for` loops to display several lines of characters, each shorter than the line that comes before it. For example, if the user types the number 5, the program displays

```
-----
----
---
--
-
```

  Hint: The code in this program is much like one of the snippets in the earlier "Mystery code" experiment.

- For an extra challenge, modify the code you wrote in the previous bullet so that it displays a slash (`/`) at the end of each line. For example, if the user types the number 5, the program displays

```
----/
---/
--/
-/
/
```

TIMES TABLE

This experiment comes in four parts:

- Write a program that reads a number from the keyboard. The program uses a `for` loop to display all numbers up to and including that number. For example, if the user types 9, the program displays

```
1       2       3       4       5       6       7       8       9
```

TIP

For some suggestions about displaying space between the numbers, refer to Chapter 11.

- Write a program that reads a number from the keyboard. The program uses a `for` loop to display two times 1, two times 2, and so on, up to and including two times the user's number. For example, if the user types 9, the program displays

```
2        4        6        8        10        12        14        16        18
```

- Write a program that uses nested `for` loops to display a multiplication table:

```
1        2         3         4         5         6         7         8         9
2        4         6         8         10        12        14        16        18
3        6         9         12        15        18        21        24        27
4        8         12        16        20        24        28        32        36
5        10        15        20        25        30        35        40        45
6        12        18        24        30        36        42        48        54
7        14        21        28        35        42        49        56        63
8        16        24        32        40        48        56        64        72
9        18        27        36        45        54        63        72        81
```

- For an extra challenge, add a header row and header column to your multiplication table. The resulting display looks like this:

```
        1      2       3       4       5       6       7       8       9
       -----------------------------------------------------------------
1      |1      2       3       4       5       6       7       8       9
2      |2      4       6       8       10      12      14      16      18
3      |3      6       9       12      15      18      21      24      27
4      |4      8       12      16      20      24      28      32      36
5      |5      10      15      20      25      30      35      40      45
6      |6      12      18      24      30      36      42      48      54
7      |7      14      21      28      35      42      49      56      63
8      |8      16      24      32      40      48      56      64      72
9      |9      18      27      36      45      54      63      72      81
```

IN THIS CHAPTER

- Using for loops to the max
- Storing many values in a single variable
- Working with groups of values

Chapter 19
Out of Many, One

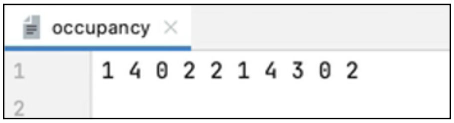
This chapter has nine illustrations. For these illustrations, the people at Wiley Publishing insist on the following numbering: Figure 19-1, Figure 19-2, Figure 19-3, Figure 19-4, Figure 19-5, Figure 19-6, Figure 19-7, Figure 19-8, and Figure 19-9. But I like a different kind of numbering. I'd like to number the illustrations `figure[0]`, `figure[1]`, `figure[2]`, `figure[3]`, `figure[4]`, `figure[5]`, `figure[6]`, `figure[7]`, and `figure[8]`. In this chapter, you find out why.

Some Loops in Action

The Java Motel, with its ten comfortable rooms, sits in a quiet place off the main highway. Aside from a small, separate office, the motel is just one long row of ground-floor rooms. Each room is easily accessible from the spacious front parking lot.

Oddly enough, the motel's rooms are numbered 0 through 9. I could say that the numbering is a fluke — something to do with the builder's original design plan. But the truth is, starting with 0 makes the examples in this chapter easier to write.

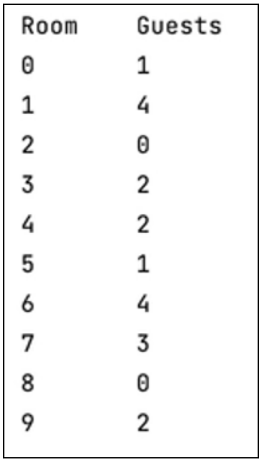
You, as the Java Motel's manager, store occupancy data in a file on your computer's hard drive. The file has one entry for each room in the motel. For example, in Figure 19-1, Room 0 has one guest, Room 1 has four guests, Room 2 is empty, and so on.



```
1 4 0 2 2 1 4 3 0 2
```

FIGURE 19-1: Occupancy data for the Java Motel.

You want a report showing the number of guests in each room. Because you know how many rooms you have, this problem begs for a `for` loop. The code to solve this problem is in Listing 19-1, and a run of the code is shown in Figure 19-2.



```
Room    Guests
0       1
1       4
2       0
3       2
4       2
5       1
6       4
7       3
8       0
9       2
```

FIGURE 19-2: Running the code in Listing 19-1.

LISTING 19-1: A Program to Generate an Occupancy Report

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

import static java.lang.System.out;

public class ShowOccupancy {

    public static void main(String[] args) throws FileNotFoundException {
        var diskScanner = new Scanner(new File("occupancy"));
        out.println("Room    Guests");

        for (int roomNum = 0; roomNum < 10; roomNum++) {
            out.print(roomNum);
            out.print("       ");
            out.println(diskScanner.nextInt());
        }

        diskScanner.close();
    }
}
```

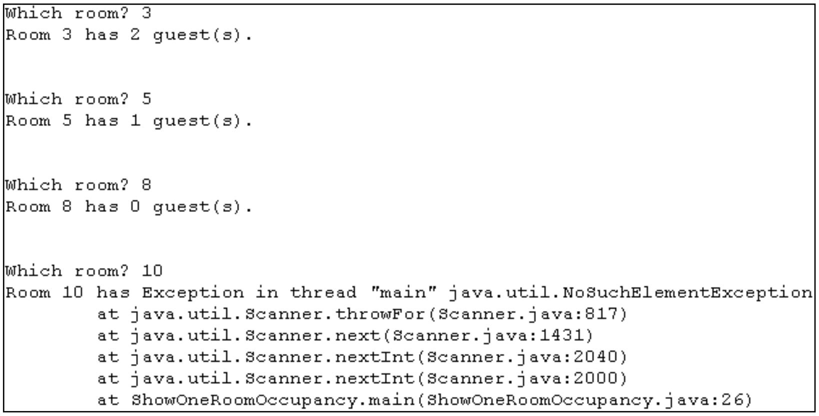
Listing 19-1 uses a `for` loop — a loop of the kind described in Chapter 12. As the `roomNum` variable's value marches from 0 to 9, the program displays one number after another from the `occupancy` file. To read more about getting numbers from a disk file like my `occupancy` file, see Chapter 16.

REMEMBER

This example's input file is named `occupancy` — not `occupancy.txt`. If you use Windows Notepad to make an `occupancy` file, you must use quotation marks in the Save As dialog box's File Name field. That is, you must type **"occupancy"** (with quotation marks) in the File Name field. If you don't surround the name with quotation marks, Notepad adds a default extension to the file's name (turning `occupancy` into `occupancy.txt`). A similar issue applies to the Macintosh's TextEdit program. By default, TextEdit adds the `.rtf` extension to each new file. To override the `.rtf` default for a particular file, choose Format ➪ Make Plain Text. Then, in the Save As dialog box, remove the check mark from the check box labeled If No Extension Is Provided, Use ".txt". (To override the default for all newly created files, choose TextEdit ➪ Preferences. Then, in the Format part of the Preferences dialog box's New Document tab, select Plain Text.)

Deciding on a loop's limit at runtime

On occasion, you may want a more succinct report than the one in Figure 19-2. "Don't give me a long list of rooms," you say. "Just give me the number of guests in Room 3." To get such a report, you need a slightly smarter program. The program is in Listing 19-2, with runs of the program shown in Figure 19-3.



```
Which room? 3
Room 3 has 2 guest(s).


Which room? 5
Room 5 has 1 guest(s).


Which room? 8
Room 8 has 0 guest(s).


Which room? 10
Room 10 has Exception in thread "main" java.util.NoSuchElementException
        at java.util.Scanner.throwFor(Scanner.java:817)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at ShowOneRoomOccupancy.main(ShowOneRoomOccupancy.java:26)
```

FIGURE 19-3: Some individual room reports.

LISTING 19-2: Report on One Room Only, Please

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

import static java.lang.System.out;

public class ShowOneRoomOccupancy {

    public static void main(String[] args) throws FileNotFoundException {
        var keyboard = new Scanner(System.in);
        var diskScanner = new Scanner(new File("occupancy"));
        int whichRoom;

        out.print("Which room? ");
        whichRoom = keyboard.nextInt();
        for (int roomNum = 0; roomNum < whichRoom; roomNum++) {
            diskScanner.nextInt();
        }

        out.print("Room ");
        out.print(whichRoom);
        out.print(" has ");
        out.print(diskScanner.nextInt());
        out.println(" guest(s).");

        keyboard.close();
        diskScanner.close();
    }
}
```

GRABBING INPUT HERE AND THERE

Listing 19-2 illustrates some pithy issues surrounding the input of data. For one thing, the program gets input from both the keyboard and a disk file. (The program gets a room number from the keyboard. Then the program gets the number of guests in that room from the occupancy file.) To make this happen, Listing 19-2 sports two Scanner declarations: one to declare keyboard and a second to declare diskScanner.

Later in the program, the call keyboard.nextInt reads from the keyboard, and diskScanner.nextInt reads from the file. Within the program, you can read from the keyboard or the disk as many times as you want. You can even intermingle the calls — reading once from the keyboard, and then three times from the disk, and then twice from the keyboard, and so on. All you have to do is remember to use keyboard whenever you read from the keyboard and use diskScanner whenever you read from the disk.

Another interesting tidbit in Listing 19-2 concerns the occupancy file. Many of this chapter's examples read from an occupancy file, and I use the same data in each of the examples. (I use the data shown in Figure 19-1.) To run an example, I copy the occupancy file from one IntelliJ project to another. (Before running the code in Listing 19-2, I right-click the occupancy file in IntelliJ's Project tool window and choose Copy from the context menu. Then I right-click the topmost branch in the Project tool window for Listing 19-2. On the resulting context menu, I choose Paste.)

In real life, having several copies of a data file can be dangerous. You can modify one copy and then accidentally read out-of-date data from a different copy. Sure, you should have backup copies, but you should have only one master copy — the copy from which all programs get the same input.

In a real-life program, you don't copy the occupancy file from one project to another. What do you do instead? You put an occupancy file in one place on your hard drive and then have each program refer to the file using the names of the file's directories. For example, if your occupancy file is in the c:\Oct\22 directory, you write

```
var diskScanner = new Scanner(new File("c:\\Oct\\22\\occupancy"));
```

The ""Name that file" sidebar in Chapter 16 has more details about filenames and double backslashes.

If Listing 19-2 has a moral, it's that the number of `for` loop iterations can vary from one run to another. The loop in Listing 19-2 runs on and on as long as the counting variable `roomNum` is less than a room number specified by the user. When the `roomNum` is the same as the number specified by the user (that is, when `roomNum` is the same as `whichRoom`), the computer jumps out of the loop. Then the computer grabs one more `int` value from the `occupancy` file and displays that value on the screen.

As you stare at the runs in Figure 19-3, it's important to remember the unusual numbering of rooms. Room 3 has two guests because Room 3 is the *fourth* room in the `occupancy` file of Figure 19-1. That's because the motel's rooms are numbered 0 through 9.

Using all kinds of conditions in a for loop

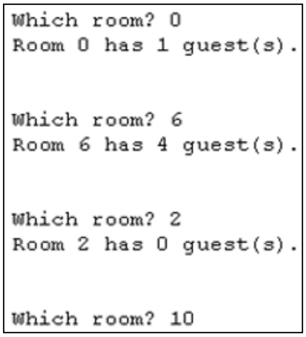
Look at the run in Figure 19-3 and notice the program's awful behavior when the user mistakenly asks about a nonexistent room: The motel has no Room 10. If you ask for the number of guests in Room 10, the program tries to read more numbers than the `occupancy` file contains. This unfortunate attempt causes a `NoSuchElementException`.

Listing 19-3 fixes the end-of-file problem.

LISTING 19-3: A More Refined Version of the One-Room Code

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

import static java.lang.System.out;

public class BetterShowOneRoom {

    public static void main(String[] args) throws FileNotFoundException {
        var keyboard = new Scanner(System.in);
        var diskScanner = new Scanner(new File("occupancy"));
        int whichRoom;

        out.print("Which room? ");
        whichRoom = keyboard.nextInt();

        for (int roomNum = 0;
                    roomNum < whichRoom && diskScanner.hasNext();
                    roomNum++) {
            diskScanner.nextInt();
        }
        if (diskScanner.hasNext()) {
            out.print("Room ");
            out.print(whichRoom);
            out.print(" has ");
            out.print(diskScanner.nextInt());
            out.println(" guest(s).");
        }

        keyboard.close();
        diskScanner.close();
    }
}
```

The code in Listing 19-3 isn't earth-shattering. To get this code, you take the code in Listing 19-2 and add a few tests for the end of the `occupancy` file. You perform the `diskScanner.hasNext` test before each call to `nextInt`. That way, if the call to `nextInt` is doomed to failure, you catch the potential failure before it happens. A few test runs of the code in Listing 19-3 are shown in Figure 19-4.



```
Which room? 0
Room 0 has 1 guest(s).


Which room? 6
Room 6 has 4 guest(s).


Which room? 2
Room 2 has 0 guest(s).


Which room? 10
```

FIGURE 19-4: The bad room number 10 gets no response.

TECHNICAL STUFF

In Listing 19-3, I want to know whether the `occupancy` file contains any more data (any data that I haven't read yet). So I call the `Scanner` class's `hasNext` method. The `hasNext` method looks ahead to see whether I can read any kind of data — an `int` value, a `double` value, a word, a `boolean`, or whatever. That's okay for this section's example, but in some situations, you need to be pickier about your input data. For example, you may want to know whether you can call `nextInt` (as opposed to `nextDouble` or `nextLine`). Fortunately, Java has methods for your pickiest input needs. A method like `if (diskScanner.hasNextInt())` tests to see whether you can read an `int` value from the disk file. Java also has methods like `hasNextLine`, `hasNextDouble`, and so on. For more information on the plain old `hasNext` method, see Chapter 18.

Listing 19-3 has a big, fat condition to keep the for loop going:

```
for (int roomNum = 0;
           roomNum < whichRoom && diskScanner.hasNext();
           roomNum++) {
```

Many for loop conditions are simple "less-than" tests, but there's no rule saying that all for loop conditions have to be so simple. In fact, any expression can be a for loop's condition, as long as the expression has the value true or false. The condition in Listing 19-3 combines a less-than with a call to the Scanner class's hasNext method.

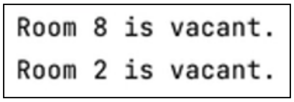
Reader, Meet Arrays; Arrays, Meet the Reader

A weary traveler steps up to the Java Motel's front desk. "I'd like a room," says the traveler. So the desk clerk runs a report like the one in Figure 19-2. Noticing the first vacant room in the list, the clerk suggests Room 2. "I'll take it," says the traveler.

It's hard to get good help these days. How many times have you told the clerk to fill the higher-numbered rooms first? The lower-numbered rooms are older, and they are badly in need of repair. For example, Room 3 has an indoor pool. (The pipes leak, so the carpet is soaking wet.) Room 2 has no heat (not in wintertime, anyway). Room 1 has serious electrical problems (for that room, you always get payment in advance). Besides, Room 8 is vacant, and you charge more for the higher-numbered rooms.

Here's where a subtle change in presentation can make a big difference. You need a program that lists vacant rooms in reverse order. That way, Room 8 catches the clerk's eye before Room 2 does.

Think about strategies for a program that displays data in reverse. With the input from Figure 19-1, the program's output should look like the display shown in Figure 19-5.



```
Room 8 is vacant.
Room 2 is vacant.
```

FIGURE 19-5: A list of vacant rooms, with higher-numbered rooms shown first.

Here's the first (bad) idea for a programming strategy:

```
Get the last value in the occupancy file.
If the value is 0, print the room number.

Get the next-to-last value in the occupancy file.
If the value is 0, print the room number.

And so on... .
```

With some fancy input/output programs, this strategy may be workable. But no matter what input/output program you use, jumping directly to the end or to the middle of a file is a big pain in the boot. It's especially bad if you plan to jump repeatedly. So go back to the drawing board and think of something better.

Here's an idea! Read all values in the `occupancy` file and store each value in a variable of its own. Then step through the variables in reverse order, displaying a room number when it's appropriate to do so.

This idea works, but the code is so ugly that I refuse to dignify it by calling it a listing. No, this is just a "see the following code" kind of thing. So please, see the following ugly code:

```
/*
* Ugh! I can't stand this ugly code!
*/
guestsIn0 = diskScanner.nextInt();
guestsIn1 = diskScanner.nextInt();
guestsIn2 = diskScanner.nextInt();
guestsIn3 = diskScanner.nextInt();
guestsIn4 = diskScanner.nextInt();
guestsIn5 = diskScanner.nextInt();
guestsIn6 = diskScanner.nextInt();
guestsIn7 = diskScanner.nextInt();
guestsIn8 = diskScanner.nextInt();
guestsIn9 = diskScanner.nextInt();

if (guestsIn9 == 0) {
    System.out.println(9);
}
if (guestsIn8 == 0) {
    System.out.println(8);
}
if (guestsIn7 == 0) {
    System.out.println(7);
}
if (guestsIn6 == 0) {

// And so on ...
```

What you're lacking is a uniform way of naming ten variables. That is, it would be nice to write

```
/*
* Nice idea, but this is not real Java code:
*/

//Read forward
for (int roomNum = 0; roomNum < 10; roomNum++) {
    guestsInroomNum = diskScanner.nextInt();
}

//Write backward
for (int roomNum = 9; roomNum >= 0; roomNum--) {
    if (guestsInroomNum == 0) {
        System.out.println(roomNum);
    }
}
```

Well, you can write loops of this kind. All you need are some square brackets. When you add square brackets to the idea shown in the preceding code, you get what's called an array. An *array* is a row of values, like the row of rooms in a one-floor motel. To picture the array, just picture the Java Motel:

- First, picture the rooms lined up next to one another.
- Next, picture the same rooms with their front walls missing. Inside each room, you can see a certain number of guests.
- If you can, forget that the two guests in Room 9 are putting piles of bills into a big briefcase. Ignore the fact that the guest in Room 5 hasn't moved away from the TV set in a day-and-a-half. Instead of all these details, just see numbers. In each room, see a number representing the count of guests in that room. (If freeform visualization isn't your strong point, take a look at Figure 19-6.)

In the lingo of Java programming, the entire row of rooms is called an *array*. Each room in the array is called a *component* of the array (also known as an array *element*). Each component has two numbers associated with it:

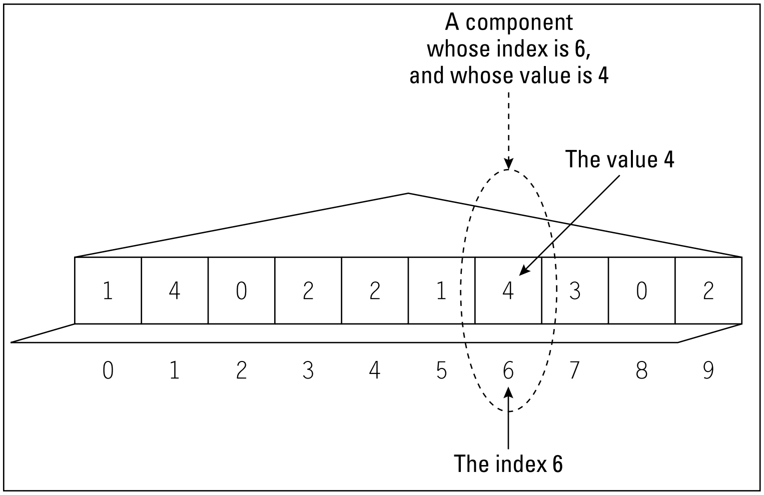


FIGURE 19-6: An abstract snapshot of rooms in the Java Motel.

- **Index:** In the case of the Java Motel array, the index is the room number (a number from 0 to 9).
- **Value:** In the Java Motel array, the value is the number of guests in a given room (a number stored in a component of the array).

Using an array saves you from having to declare ten separate variables: `guestsIn0`, `guestsIn1`, `guestsIn2`, and so on. To declare an array with ten values in it, you can write two fairly short lines of code:

```
int[] guestsIn;
guestsIn = new int[10];
```

You can even squish these two lines into one longer line:

```
int[] guestsIn = new int[10];
```

And, under the right circumstances, you can avoid repeating yourself by using the word `var`:

```
var guestsIn = new int[10];
```

In any of these code snippets, notice the use of the number `10`. This number tells the computer to make the `guestsIn` array have ten components. Each component of the array has a name of its own. The starting component is named `guestsIn[0]`, the next is named `guestsIn[1]`, and so on. The last of the ten components is named `guestsIn[9]`.

REMEMBER

In creating an array, you always specify the number of components. The array's indices always start with 0 and end with the number that's one fewer than the total number of components. For example, if your array has ten components (and you declare the array with `new int[10]`), the array's indices go from 0 to 9.

TIP

When you create an array variable, you can put square brackets after either the type name or the variable name. In other words, you can write

```
int[] guestsIn;
```

as I do in this section, or you can write

```
int guestsIn[];
```

as some programmers do. Either way, you're defining exactly the same array variable. In the same way, you see

```
public static void main(String[] args)
```

and you also see

```
public static void main(String args[])
```

These two method headers have precisely the same meaning.

Storing values in an array

After you've created an array, you can put values into the array's components. For example, the guests in Room 6 are fed up with all those mint candies that you put on people's beds. So they check out and Room 6 becomes vacant. You should put the value `0` into the `6` component. You can do it with this assignment statement:

```
guestsIn[6] = 0;
```

On one weekday, business is awful. No one's staying at the motel. But then you get a lucky break: A big bus pulls up to the motel. The side of the bus sports a Loners' Convention sign. Out of the bus come 25 people, each walking to the motel's small office, none paying attention to the others who were on the bus. Each person wants a private room. Only 10 of them can stay at the Java Motel, but that's okay because you can send the other 15 loners down the road to the old C-Side Resort and Motor Lodge.

Anyway, to register ten of the loners at the Java Motel, you put one guest in each of your ten rooms. Having created an array, you can take advantage of the array's indexing and write a for loop, like this:

```
for (int roomNum = 0; roomNum < 10; roomNum++) {
    guestsIn[roomNum] = 1;
}
```

This loop takes the place of ten assignment statements because the computer executes the statement guestsIn[roomNum] = 1 ten times. The first time around, the value of roomNum is 0, so in effect, the computer executes

```
guestsIn[0] = 1;
```

In the next loop iteration, the value of roomNum is 1, so the computer executes the equivalent of the following statement:

```
guestsIn[1] = 1;
```

During the next iteration, the computer behaves as though it's executing

```
guestsIn[2] = 1;
```

And so on. When roomNum gets to be 9, the computer executes the equivalent of the following statement:

```
guestsIn[9] = 1;
```

Notice that the loop's counter goes from 0 to 9. Compare this with Figure 19-6 and remember that the indices of an array go from 0 to one fewer than the number of components in the array. Looping with room numbers from 0 to 9 covers all rooms in the Java Motel.

REMEMBER

When you work with an array and you step through the array's components using a for loop, you normally start the loop's counter variable at 0. To form the condition that tests for another iteration, you often write an expression like roomNum < arraySize, where arraySize is the number of components in the array.

Creating a report

The code to create the report in Figure 19-5 is shown in Listing 19-4. This new program uses the idea in the world's ugliest code (the code from several pages back, with variables guestsIn0, guestsIn1, and so on). But rather than have ten separate variables, Listing 19-4 uses an array.

LISTING 19-4: Traveling through Data Both Forward and Backward

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class VacanciesInReverse {

    public static void main(String[] args) throws FileNotFoundException {
        var diskScanner = new Scanner(new File("occupancy"));
        var guestsIn = new int[10];

        for (int roomNum = 0; roomNum < 10; roomNum++) {
            guestsIn[roomNum] = diskScanner.nextInt();
        }

        for (int roomNum = 9; roomNum >= 0; roomNum--) {
            if (guestsIn[roomNum] == 0) {
                System.out.print("Room ");
                System.out.print(roomNum);
                System.out.println(" is vacant.");
            }
        }

        diskScanner.close();
    }
}
```

Notice the stuff in parentheses in the `VacanciesInReverse` program's second `for` loop. It's easy to get these things wrong. You're aiming for a loop that checks Room 9, and then Room 8, and so on:

```
if (guestsIn[9] == 0) {
    System.out.print(roomNum);
}
if (guestsIn[8] == 0) {
    System.out.print(roomNum);
}
if (guestsIn[7] == 0) {
    System.out.print(roomNum);
}

// ... And so on, until you get to ...

if (guestsIn[0] == 0) {
    System.out.print(roomNum);
}
```

Some observations about the code:

- The loop's counter must start at 9:

  ```
  for (int roomNum = 9; roomNum >= 0; roomNum--)
  ```

- Each time through the loop, the counter goes *down* by one:

  ```
  for (int roomNum = 9; roomNum >= 0; roomNum--)
  ```

- The loop keeps going as long as the counter is *greater than or equal to* 0:

  ```
  for (int roomNum = 9; roomNum >= 0; roomNum--)
  ```

Think through each of these three items and you'll write a perfect `for` loop.

Stuffing values into an array

In Listing 19-4, you put values into the `guestsIn` array by repeatedly reading numbers from a disk file and storing the numbers, element by element, in the array. Rather than drop array values in one by one, you can populate an array in one fell swoop. This section describes a few ways for you to do it:

- **You can use Java's `fill` method.**

  When your hotel is brand-spanking-new, you start off with no guests in any of the rooms. You can put zeros in each room with the following statement:

  ```
  Arrays.fill(guestsIn, 0);
  ```

  If for some reason you want to put two guests in each room, you'd do this:

  ```
  Arrays.fill(guestsIn, 2);
  ```

  The only prerequisite is an `import` declaration at the start of your program:

  ```
  import java.util.Arrays;
  ```

  TECHNICAL STUFF

  What's going on when you call `Arrays.fill`? Java has a class named `Arrays`, and that class has a static method named `fill`. For more on static methods, refer to Chapter 14.

- **You can use curly braces.**

  The `Arrays.fill` method is useful for putting several copies of a particular value into an array. For more flexibility, you can do this:

  ```
  public class Vacancies {

      public static void main(String[] args) {

          int[] guestsIn;
          guestsIn = new int[]{1, 4, 0, 2, 2, 1, 4, 3, 0, 2};

          for (int roomNum = 0; roomNum < 10; roomNum++) {
              System.out.println(guestsIn[roomNum]);
          }
      }
  }
  ```

  In this code, the bold assignment statement tells Java to create a new array with 1 in `guestsIn[0]`, 4 in `guests[1]`, 0 in `guestsIn[2]`, and so on.

  TIP

  You don't have to count the number of values that you typed between the open and close braces. The word `length` can do the counting for you:

  ```
  for (int roomNum = 0; roomNum < guestsIn.length; roomNum++) {
  ```

- **You can initialize an array.**

  When you combine a declaration and an assignment, you get an *array initialization*. In an array initialization, you can omit words like `new int[]`:

  ```
  int[] guestsIn = {1, 4, 0, 2, 2, 1, 4, 3, 0, 2};
  ```

  But beware! Initializations and assignments are two different kinds of things. In an ordinary assignment statement, you need more than just curly braces:

  ```
  int[] guestsIn = {1, 4, 0, 2, 2, 1, 4, 3, 0, 2};   // This is good.

  int[] bedsIn;
  bedsIn = new int[]{2, 1, 1, 1, 2, 1, 1, 1, 1, 2};  // This is good.

  int[] towelsIn;
  towelsIn = {2, 4, 2, 2, 2, 4, 4, 0, 0, 2};         // This is bad.
  ```

  In some programming languages, a line like the one marked `This is bad` is a valid assignment statement. But Java isn't one of those languages. Java thinks that this line stinks.

Working with Arrays

Earlier in this chapter, a busload of loners showed up at your motel. When they finally left, you were glad to get rid of them, even if it meant having all your rooms empty for a while. But now another bus pulls into the parking lot. This bus sports a Gregarian Club sign. Out of the bus come 50 people, each more gregarious than the next. Now everybody in the parking lot is clamoring to meet everyone else. While they meet-and-greet, they're all frolicking toward the front desk, singing the club's theme song. (Oh, no! It's the Gregarian chant!)

The first five Gregarians all want Room 7. It's a tight squeeze, but you were never big on fire codes, anyway. Next comes a group of three with a yen for Room 0. (They're computer programmers, and they think the room number is cute.) Then there's a pack of four Gregarians who want Room 3. (The in-room pool sounds attractive to them.)

With all this traffic, you had better switch on your computer. You start a program that enables you to enter new occupancy data. The program has five parts:

- **Create an array and put** 0 **in each of the array's components.**

  When the Loners' Club members left, the motel was suddenly empty. (Heck, even before the Loners' Club members left, the motel seemed empty.) To declare an array and fill the array with zeros, you execute code of the following kind:

  ```
  var guestsIn = new int[10];
  Arrays.fill(guestsIn, 0);
  ```

- **Get a room number and then get the number of guests who will be staying in that room.**

  Reading numbers typed by the user is pretty humdrum stuff. Do a little prompting and a little `nextInt` calling and you're all set:

  ```
  out.print("Room number: ");
  whichRoom = keyboard.nextInt();
  out.print("How many guests? ");
  numGuests = keyboard.nextInt();
  ```

- **Use the room number and the number of guests to change a value in the array.**

  Earlier in this chapter, to put one guest in Room 2, you executed

  ```
  guestsIn[2] = 1;
  ```

  So now you have two variables: `numGuests` and `whichRoom`. Maybe `numGuests` is `5` and `whichRoom` is `7`. To put `numGuests` in `whichRoom` (that is, to put five guests in Room 7), you can execute

  ```
  guestsIn[whichRoom] = numGuests;
  ```

  That's the crucial step in the design of your new program.

- **Ask the user whether the program should continue.**

  Are there more guests to put in rooms? To find out, execute this code:

  ```
      out.print("Do another? ");
  } while (keyboard.findWithinHorizon(".",0).charAt(0) == 'Y');
  ```

- **Display the number of guests in each room.**

  No problem! You already did this. You can steal the code (almost verbatim) from Listing 19-1:

  ```
  out.println("Room    Guests");
  for (int roomNum = 0; roomNum < 10; roomNum++) {
      out.print(roomNum);
      out.print("       ");
      out.println(guestsIn[roomNum]);
  }
  ```

  The only difference between this latest code snippet and the stuff in Listing 19-1 is that this new code uses the `guestsIn` array. The first time through this loop, the code does

  ```
  out.println(guestsIn[0]);
  ```

  displaying the number of guests in Room 0. The next time through the loop, the code does

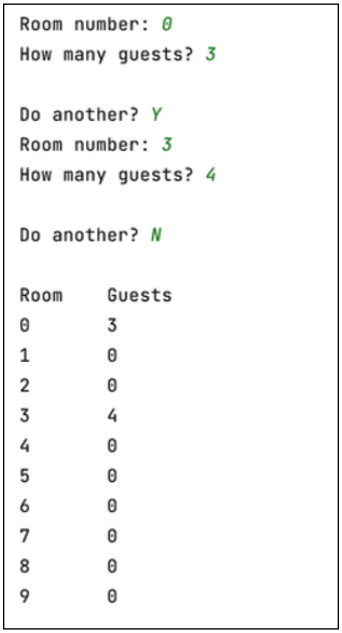
  ```
  out.println(guestsIn[1]);
  ```

  displaying the number of guests in Room 1. The last time through the loop, the code does

  ```
  out.println(guestsIn[9]);
  ```

  That's perfect.

The complete program (with these five pieces put together) is in Listing 19-5. A run of the program is shown in Figure 19-7.



```
Room number: 0
How many guests? 3

Do another? Y
Room number: 3
How many guests? 4

Do another? N

Room    Guests
0       3
1       0
2       0
3       4
4       0
5       0
6       0
7       0
8       0
9       0
```

FIGURE 19-7: Running the code in Listing 19-5.

LISTING 19-5: Storing Occupancy Data in an Array

```
import java.util.Arrays;
import java.util.Scanner;

import static java.lang.System.out;

public class AddGuests {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
        int whichRoom, numGuests;
        var guestsIn = new int[10];

        Arrays.fill(guestsIn, 0);

        do {
            out.print("Room number: ");
            whichRoom = keyboard.nextInt();
            out.print("How many guests? ");
            numGuests = keyboard.nextInt();
            guestsIn[whichRoom] = numGuests;
            out.println();
            out.print("Do another? ");
        } while (keyboard.findWithinHorizon(".", 0).charAt(0) == 'Y');
        out.println();
```

(continued)

LISTING 19-5: ***(continued)***

```
        out.println("Room    Guests");
        for (int roomNum = 0; roomNum < 10; roomNum++) {
            out.print(roomNum);
            out.print("       ");
            out.println(guestsIn[roomNum]);
        }

        keyboard.close();
    }
}
```

Hey! The program in Listing 19-5 is *big!* It may be the biggest program so far in this book. But big doesn't necessarily mean difficult. If each piece of the program makes sense, you can create each piece on its own and then put all the pieces together. Voilà! The code is manageable.

Looping in Style

The last loop in Listing 19-5 program looks something like this:

```
for (int roomNum = 0; roomNum < 10; roomNum++) {
    out.println(guestsIn[roomNum]);
}
```

You can simplify this code by using an *enhanced* `for` *loop*. To create an enhanced `for` loop, you make up a new variable name. What about the name `howMany`? I like that name:

```
for (int howMany : guestsIn) {
    out.println(howMany);
}
```

Whatever name you choose, the new variable ranges over the values in the `guestsIn` array. For example, if the `guestsIn` array stores the `int` values 1, 4, 0, 2, 2, 1, 4, 3, 0, and 2, then the value of `howMany` becomes 1, then 4, then 0, then 2, and so on. The output of the code looks like this:

```
1
4
0
2
2
1
4
3
0
2
```

Enhanced `for` loops are nice and concise. But don't be too eager to use enhanced loops with arrays. This feature has some nasty limitations. For example, my new `howMany` loop doesn't display room numbers — the array indices 0 through 9. Unfortunately, an enhanced loop doesn't provide ready access to an array's indices.

Here's another unpleasant surprise. Start with the following loop from Listing 19-4:

```
for (int roomNum = 0; roomNum < 10; roomNum++) {
    guestsIn[roomNum] = diskScanner.nextInt();
}
```

Turn this traditional `for` loop into an enhanced `for` loop and you get the following misleading code:

```
for (int howMany : guestsIn) {
    howMany = diskScanner.nextInt(); //Don't do this
}
```

The new enhanced `for` loop doesn't do what you want it to do. This loop reads values from an input file and then dumps these values into the garbage can. In the end, the array's values remain unchanged.

It's sad but true. To make full use of an array, you have to fall back on Java's plain old `for` loop.

TRY IT OUT

Would you like to flex some array muscles? If so, here are some things for you to try:

INITIALIZE AN ARRAY

This experiment comes in three parts:

- Run the following program to find out how array initialization works:

```
public class Main {
    public static void main(String[] args) {
        int[] myArray = { 9, 21, 35, 16, 21, 7 };
        System.out.println(myArray[0]);
        System.out.println(myArray[1]);
        System.out.println(myArray[5]);
        // System.out.println(myArray[6]);
    }
}
```

- What happens when you uncomment the last `System.out.println` call in the previous bullet's program and then run the program? Why does this happen?
- What happens when you try to replace one line of code with two lines in the first bullet's program?

```
int[] myArray;
myArray = { 9, 21, 35, 16, 21, 7 };
```

Does this explain why an array initialization isn't called an array assignment?

PICK AN ELEMENT

Create a program containing the following array initialization:

```
int[] amounts = {19, 21, 16, 14, 99, 86, 31, 19, 0, 101};
```

In your program, ask the user to input a position number — a number from 0 to 9. Have your program respond by displaying the value in that position of the `amounts` array. For example, if the user inputs 0, the program displays 19. If the user inputs 1, the program displays 21. And so on.

DISPLAY THE ELEMENTS

Create a program containing the following array initialization:

```
int[] amounts = {19, 21, 16, 14, 99, 86, 31, 19, 0, 101};
```

Add code to display all indices and values in the array. The first three lines of output should look like this:

```
The 0 element's value is 19.
The 1 element's value is 21.
The 2 element's value is 16.
```

DISPLAY SOME OF THE ELEMENTS

Create a program containing the following array initialization:

```
int[] amounts = {19, 21, 16, 14, 99, 86, 31, 19, 0, 101};
```

Add a loop that displays the values in even-numbered positions of the array. The program's output is `19 16 99 31 0`.

GENERATE SQUARES

I've created a program that uses a loop to generate an array of the first 50 perfect squares. Here's my program, with some code missing:

```
public class Main {

    public static void main(String[] args) {
        int[] squares = ___________;

        for (______________________) {
            squares[i] = _____;
        }

        System.out.println(squares[0]);
        System.out.println(squares[1]);
        System.out.println(squares[2]);
        System.out.println(squares[49]);
    }
}
```

Fill in the missing code. When you run the program, the output looks like this:

```
0
1
4
2401
```

FIND ONE VACANCY

Someone shows up at the front desk, asking for a room. The hotel clerk doesn't need a list of all vacant rooms. All the clerk needs is the number of a single vacant room. Any vacant room will do. Modify the code in Listing 19-4 so that it shows only one room number (the number of a room that's vacant).

SELECT A ROOM

Modify the code in Listing 19-5 so that it doesn't ask the user which room number to put guests in. The code automatically selects a room from the rooms that are vacant.

HOW MANY GUESTS?

Modify the code in Listing 19-4 or Listing 19-5 so that the program displays the total number of guests in the motel. (To do this, the code adds up the numbers of guests in each room.)

AN ARRAY OF STRINGS

A fancy hotel in Philadelphia has seven conference rooms — each named after one of the city's distinguishing characteristics. Add code to display the names of the seven rooms:

```
public class MeetingRooms {

    public static void main(String[] args) {
        var roomName = new String[7];

        roomName[0] = "Liberty Bell";
        roomName[1] = "Mummers";
        roomName[2] = "Rocky Balboa";
        roomName[3] = "Cheesesteak";
        roomName[4] = "Hoagie";
        roomName[5] = "Water Ice";
        roomName[6] = "El Train";

        // Your code goes here.
    }
}
```

AN ARRAY OF PURCHASES

With the `guestsIn` array from Listing 19-4, each value is an `int`. Create an array in which each value is a `Purchase`. Use the `Purchase` class from Chapter 13.

FUN WITH WORD ORDER

Write a program that inputs six words from the keyboard. The program outputs six sentences, each with the first word in a different position. For example, the output of one run might look like this:

```
only I have eyes for you.
I only have eyes for you.
I have only eyes for you.
I have eyes only for you.
I have eyes for only you.
I have eyes for you only.
```

PARALLEL ARRAYS

Create a new IntelliJ project and put the following code in the project's `main` method:

```
char[] cipher = { 's', 'f', 'k', 'l', 'd', 'o', 'h', 'z', 'm', 'b',
        't', 'a', 'n', 'g', 'u', 'v', 'I', 'q', 'x', 'w', 'y', 'c',
        'j', 'r', 'p', 'e' };
char[] plain = { 'e', 'q', 's', 'f', 'I', 'n', 'h', 'u', 'r', 'k',
        'g', 'z', 'c', 'y', 'x', 'l', 'm', 'd', 'w', 'a', 'b', 't',
        'p', 'j', 'v', 'o' };
```

This code creates two arrays. In the first array, `cipher[0]` is `'s'`, `cipher[1]` is `'f'`, `cipher[2]` is `'k'`, and so on. In the second array, `plain[0]` is `'e'`, `plain[1]` is `'q'`, `plain[2]` is `'s'`, and so on.

Finish writing the `main` method so that when the user types a lowercase letter, the program looks for that letter in the `cipher` array and responds by displaying the corresponding letter in the `plain` array.

For example, if the user types the letter `s`, the program answers back with the letter `e`. (The program discovers that `s` is in the 0 position of the `cipher` array, so the program displays the letter in the 0 position of the `plain` array. And the letter in the 0 position of the `plain` array is `e`.)

Similarly, if the user types `f`, the program displays `q` because `f` is in the 1 position of the `cipher` array and `q` is in the 1 position of the `plain` array.

DECIPHER CIPHERTEXT

Here's a challenging task for all you ciphertext enthusiasts! Enclose in a loop the code that you just wrote for the "Parallel arrays" experiment. Have the user type a word, followed immediately by a blank space. When the user presses Enter, the program repeatedly does what the code in the parallel-arrays experiment did: The program looks up all the user's letters in the `cipher` array and displays the corresponding `plain` array letters. For example, if the user types `rwpw`, the program responds by displaying the word `java`.

When Good Arrays Go Bad

Arrays are useful, but they have some serious limitations. Imagine that you store customer names in some predetermined order. Your code contains an array, and the array has space for 100 names:

```
var name = new String[100];

for (int i = 0; i < 100; i++) {
    name[i] = diskScanner.next();
}
```

All is well until, one day, your 101st customer shows up. You enter data for the 101st customer, hoping desperately that the array with 100 components can expand to fit your growing needs.

No such luck. Arrays don't expand. You have to turn away your 101st customer. Too bad! That one was planning to write you a stellar review!

"In my next life, I'll create arrays of length 1,000," you say to yourself. And when your next life rolls around, you do just that:

```
var name = new String[1000];
for (int i = 0; i < 1000; i++) {
    name[i] = = diskScanner.next();
}
```

But during your next life, an economic recession occurs. Rather than have 101 customers, you have only 3 customers. Now you're wasting space for 1,000 names when space for 3 names would do.

And what if no economic recession occurs? You have 825 customers. You're sailing along with your array of size 1,000, using a tidy 825 spaces in the array.

The components with indices 0 through 824 are being used, and the components with indices 825 through 999 are waiting quietly to be filled.

One day, a brand-new customer shows up. Because your customers are stored in order (alphabetically by last name or numerically by Social Security number or whatever), you want to squeeze this customer into the correct component of your array. The trouble is that this customer belongs very early on in the array, at the component with index 7. What happens then?

You take the name in component number 824 and move it to component 825. Then you take the name in component 823 and move it to component 824. Take the name in component 822 and move it to component 823. You keep doing this until you've moved the name in component 7. Then you put the new customer's name into component 7. What a pain! Sure, the computer doesn't complain. (If the computer has feelings, it probably likes this kind of busy work.) But, as you move around all these names, you waste processing time, you waste power, and you waste all kinds of resources.

"In my next life, I'll leave three empty components between every two names." And, of course, your business expands. Eventually, you find that three aren't enough.

What to Do When Arrays Go Awry

The issues raised in the previous section aren't new. Computer scientists have been working on these issues for a long time. They haven't discovered any magic one-size-fits-all solution, but they've discovered some clever tricks.

The Java API has a bunch of classes known as *collection* classes. Each collection class has methods for storing bunches of values, and each collection class's methods use some clever tricks. For you, the bottom line is this: Certain collection classes deal as efficiently as possible with the issues raised in the previous section. If you have to deal with such issues when writing code, you can use these collection classes and call the classes' methods. Rather than fret about a customer whose name belongs in position 7, you can just call a class's `add` method. The method inserts the name at a position of your choice and deals reasonably with whatever ripple effects have to take place. In the best circumstances, the insertion is very efficient. In the worst circumstances, you can rest assured that the code does everything the best way it can.

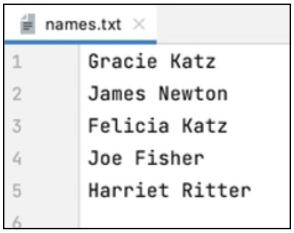
Using an ArrayList

One of the most versatile of Java's collection classes is the `ArrayList`. Listing 19-6 shows you how it works.

LISTING 19-6: Working with a Java Collection

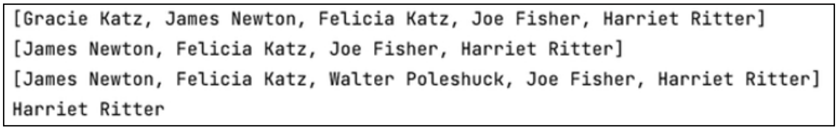
```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

import static java.lang.System.out;

public class ShowNames {

    public static void main(String[] args) throws FileNotFoundException {

        var people = new ArrayList<String>();
        var diskScanner = new Scanner(new File("names.txt"));

        while (diskScanner.hasNext()) {
            people.add(diskScanner.nextLine());
        }

        out.println(people);

        people.remove(0);
        out.println(people);

        people.add(2, "Walter Poleshuck");
        out.println(people);

        out.println(people.get(4));

        diskScanner.close();
    }
}
```

Figure 19-8 shows you a sample names.txt file. The code in Listing 19-6 reads that names.txt file and prints the stuff in Figure 19-9.



```
Gracie Katz
James Newton
Felicia Katz
Joe Fisher
Harriet Ritter
```

FIGURE 19-8: Several names in a file.



```
[Gracie Katz, James Newton, Felicia Katz, Joe Fisher, Harriet Ritter]
[James Newton, Felicia Katz, Joe Fisher, Harriet Ritter]
[James Newton, Felicia Katz, Walter Poleshuck, Joe Fisher, Harriet Ritter]
Harriet Ritter
```

FIGURE 19-9: The code in Listing 19-6 changes some of the names.

When you declare an array, you give Java two important pieces of information — the number of values stored in the array and the type of each value in the array. For example, in Listing 19-5, the line

```
var guestsIn = new int[10];
```

tells Java that `guestsIn` stores ten values and that each of these values is an `int` value. In contrast, when you declare an `ArrayList`, you give Java only one piece of information — namely, the type of each value in the `ArrayList`. For example, in Listing 19-6, the declaration

```
var people = new ArrayList<String>();
```

tells Java that the `people` list will be storing `String` values. In this declaration, the word `<String>` with angle brackets is called a *generic parameter*. Later on in the program, if you try to hide an `int` value inside the `people` list, Java tells you to put that stinking `int` value somewhere else.

CROSS REFERENCE

For more details about generic parameters, see the "Esoteric generics" sidebar, later in this chapter.

The `ArrayList` declaration in Listing 19-6 says nothing about the number of values the list can store. To bring this point home, look at the loop in Listing 19-6:

```
while (diskScanner.hasNext()) {
    people.add(diskScanner.nextLine());
}
```

The loop repeatedly takes whatever name it finds on a line of the input file and appends that name to the end of the list. What happens if the input file contains 10,000 names? Does the `ArrayList` run out of space? Might the `ArrayList` waste lots of space?

None of the above. Java's `ArrayList` knows how to shrink and grow. Starting with the file shown in Figure 19-8, the `ArrayList` ends up storing 5 values. But if the `names.txt` file were 100,000 lines long, the `ArrayList` in Listing 19-6 would end up having 100,000 values.

When the program in Listing 19-6 finishes reading the names.txt file, some interesting things happen:

- **The program displays the values it read from the names.txt file.**

  This display is the top line in Figure 19-9. When you call println with an ArrayList parameter, Java puts commas between the list's values and surrounds the entire list with square brackets. If you don't like commas and square brackets, you can get more control using an enhanced for loop:

  ```
  for (String aName : people) {
      out.println("Hello, " + aName);
  }
  ```

- **The program removes an element from the list.**

  The people variable refers to an ArrayList object. When you call that object's remove method,

  ```
  people.remove(0);
  ```

  you eliminate a value from the list. In this case, you eliminate whatever value is in the list's initial position (the position numbered 0). So, in Listing 19-6, the call to remove takes the name Gracie Katz out of the list. (Refer to the second line in Figure 19-9.)

- **The program adds an element in the middle of the list.**

  An ArrayList object has two different add methods. The add method inside the loop has only one parameter. This one-parameter add method appends its value to what's currently the end of the ArrayList object.

  In contrast, the method that adds Walter Poleshuck to the list has two parameters: a position number and a value to be added. The statement

  ```
  people.add(2, "Walter Poleshuck");
  ```

  inserts a name into position number 2. (After Gracie has been removed, position number 2 is the position occupied by Joe Fisher, so Joe moves to position 3, and Walter Poleshuck becomes the number 2 person.) The result appears in the third line of Figure 19-9.

- **The program displays a particular element in the list.**

  The call people.get(4) stands for the string "Harriet Ritter", so Harriet's name appears on the fourth line in Figure 19-9. Calling people.get(4) in Listing 19-6 is like asking for guestsIn[4] in any of this chapter's hotel array examples. With either an array or an ArrayList, you can pinpoint individual elements.

ESOTERIC GENERICS

In the ArrayList of Listing 19-6, each element is a String. The declaration's generic parameter tells you so:

```
var people = new ArrayList<String>();
```

You can shove almost any type into a generic parameter. For example, in Chapter 13, you declare your own Purchase class. You can create an ArrayList to store Purchase instances:

```
var recentPurchases = new ArrayList<Purchase>();
```

In Chapter 15, you declare your own Account class. You can make an ArrayList to hold your Account instances:

```
var activeAccounts = new ArrayList<Account>();
```

Imagine a program that reads from ten different files. You can create an ArrayList of File objects and an ArrayList of Scanner objects.

The only gotcha for a novice programmer is that you can't store primitive type values in an ArrayList. For example, you can't write

```
var tallies = new ArrayList<int>();  //Bad!
```

Chapter 7 lists six of Java's primitive types, but that chapter doesn't describe the corresponding *wrapper types*. You get a glimpse of the wrapper types in Chapter 14, where you encounter Java's Integer class. The word Integer represents a class, and a class can have methods. So, in Chapter 14, you use the Integer class's parseInt method:

```
numberOfChips = Integer.parseInt(reply);
```

Best of all, Integer can be a generic parameter.

```
var tallies = new ArrayList<Integer>();        // Good!
tallies.add(150);
tallies.add(225);
out.println(tallies.get(0) + tallies.get(1));  // Displays 375
```

Each of Java's primitive types has a corresponding wrapper class. The Table 19-1 tells you all about it.

TABLE 19-1: Wrapper Classes for Primitive Types

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

REMEMBER

Follow my lead from Listing 19-6. When you use an `ArrayList` in your code, begin by importing `java.util.ArrayList`.

TECHNICAL STUFF

Editor's note: Near the start of this section, that untrustworthy author Barry Burd writes, ". . . when you declare an `ArrayList`, you give Java only one piece of information — namely, the type of each value in the `ArrayList`." He wrote this because, in his words, "the paragraph flows better when I don't write about all the details." What a schmo he is! In truth, you can supply more than one piece of information when you declare a new `ArrayList`. You can even omit the generic parameter and supply almost no information. For more about all this, consult Oracle's official Java API documentation at

```
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/
            java/util/ArrayList.html
```

Java's many collection classes

The `ArrayList` class is only the tip of the Java collections iceberg. The Java library contains many collections classes, each with its own advantages. Table 19-2 contains an abbreviated list.

Each collection class has its own set of methods. To learn about these methods and to find out which collection classes best meet your needs, visit the Java API documentation pages at

```
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/
            java/util/package-summary.html#CollectionsFramework
```

TABLE 19-2: Some Collection Classes

Class Name	Characteristic
`ArrayList`	A resizable array.
`LinkedList`	A list of values, each having a field that points to the next one on the list.
`Stack`	A structure that grows from bottom to top. The structure is optimized for access to the top-most value. You can easily add a value to the top or remove the value from the top.
`Queue`	A structure that grows at one end. The structure is optimized for adding values to one end (the rear) and removing values from the other end (the front).
`Priority-Queue`	A structure, like a queue, that lets certain (higher-priority) values move toward the front.
`HashSet`	A collection containing no duplicate values.
`HashMap`	A collection of key/value pairs.

TRY IT OUT

Once again, I'd like to put you to work:

THE NAME GAME

Using the `names.txt` file in Figure 19-8, modify Listing 19-6 so that the output looks like this:

```
Harriet Ritter
Gracie Katz
Felicia Katz
James Newton
Joe Fisher
```

Use the `ArrayList` class's `add` and `remove` methods.

FIND THE LARGEST VALUE

Create an `ArrayList` containing `Integer` values. Then step through the values in the list to find the largest value among all values in the list. For example, if the list contains the numbers 85, 19, 0, 103, and 13, display the number 103.

GET YOUR DUCKS IN A ROW

Create an `ArrayList` containing `String` values in alphabetical order. When the user types an additional word on the keyboard, the program inserts the new word into the `ArrayList` in the proper (alphabetically ordered) place.

For example, imagine that the list starts off containing the words "cat", "dog", "horse", and "zebra" (in that order). After the user types the word fish on the keyboard (and presses Enter), the list contains the words "cat", "dog", "fish", "horse", and "zebra" (in that order).

To write this program, you may find the String class's compareToIgnoreCase method and the ArrayList class's size method useful. You can find out about these methods by visiting one of these:

```
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/
          java/lang/String.html
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/
          java/util/ArrayList.html
```

IN THIS CHAPTER

- Swinging into action
- Displaying an image
- Using buttons and text boxes

Chapter 20
Oooey-GUI Was a Worm

Have you ever heard that wonderful old joke about a circus acrobat jumping over mice? Unfortunately, I'd get sued for copyright infringement if I included the joke in this book.

Anyway, the joke is about starting small and working your way up to bigger things. That's what you do when you read *Beginning Programming with Java For Dummies,* 6th Edition.

Most of the examples in this book are command line programs. A *command line* program has no windows, no dialog boxes — nothing of that kind. With a command line program, the user types characters in Console view, and the program displays output in the same Console view.

These days, few publicly available programs are command line. Almost all programs use a *GUI* — a *g*raphical *u*ser *i*nterface. So if you've read every word of this book so far, you're probably saying to yourself, "When am I going to find out how to create a GUI?"

Well, now's the time! This chapter introduces you to the world of GUI programming in Java.

ON THE WEB

You can see GUI versions of many examples from this book by visiting the book's website (`http://beginprog.allmycode.com`).

Put Some Swing in Your Step

Java's *Swing* classes create graphical objects on a computer screen. The objects can include buttons, icons, text fields, check boxes, and other good things that make windows so useful.

JAVA GUIs

Java comes with three sets of classes for creating GUI applications:

- **The Abstract Window Toolkit (AWT):** The original set of classes, dating back to JDK 1.0.

  Classes in this set belong to packages whose names begin with `java.awt`. Components in this set have names like `Button`, `TextField`, `Frame`, and so on.

  The AWT implements only the kinds of components that were available on all common operating systems in the mid-1990s. So, using AWT, you can add a button to your application, but you can't easily add a table or a tree.

- **Java Swing:** A set of classes created to fix some of the difficulties posed by the use of the AWT. Swing was introduced in 1998 as part of Java 1.2 (also known as J2SE 1.2).

  Classes in this set belong to packages whose names begin with `javax.swing`. Components in this set have names like `JButton`, `JTextField`, and `JFrame`.

  In a Swing program, you can create table components, tree components, and many other kinds of components. Java's Swing classes replace some (but not all) of the classes in the older AWT. To use some of the Swing classes, you have to call on some of the old AWT classes.

- **JavaFX:** An alternative to Swing, announced in May 2007. JavaFX comes with new(er) versions of Java 7 and with all later versions of Java.

  Classes in this set belong to packages whose names begin with `javafx`.

  JavaFX supports over 100 kinds of components. (Sure, you want a `Button` component. But do you also want an `Accordion` component? JavaFX has one.) In addition, JavaFX supports multitouch operations and takes advantage of each processor's specialized graphics capabilities. Unlike AWT and Swing, the JavaFX platform isn't included as part of the standard Java SDK. So, if you want to create a JavaFX application, you have to do some additional project setup steps.

  For more information about JavaFX, visit `https://openjfx.io`.

The name Swing isn't an acronym. When the stewards of the Java programming language were first creating the code for these classes, one of the developers named it Swing because swing music was enjoying a nostalgic revival. And yes, in addition to `String` and `Swing`, the standard Java API has a `Spring` class. But that's another story.

Actually, Java's API has several sets of windowing components. For details, see the nearby "Java GUIs" sidebar.

The merry window

The program in Listing 20-1 displays a window on your computer screen. To see the window, look at Figure 20-1.

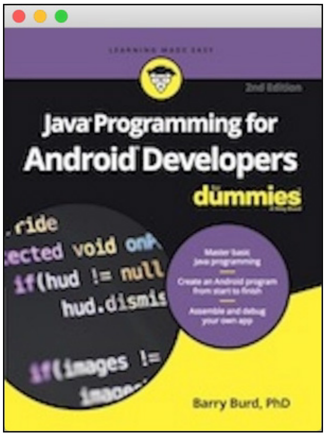

LEARNING MADE EASY
2nd Edition
Java Programming for
Android Developers
dummies
Master basic Java programming
Create an Android program from start to finish
Assemble and debug your own app
Barry Burd, PhD

FIGURE 20-1: What a nice window!

The code in Listing 20-1 has little logic of its own. Instead, this code pulls together a bunch of classes from the Java API.

LISTING 20-1: Creating a Window with an Image in It

```
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class ShowPicture {

    public static void main(String[] args) {
        var frame = new JFrame();
        var icon = new ImageIcon("androidBook.jpg");
```

(continued)

LISTING 20-1: ***(continued)***

```
        var label = new JLabel(icon);
        frame.add(label);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Over in Listing 13-5 (in Chapter 13), I created an instance of the Purchase class with the line

```
var purchase1 = new Purchase();
```

In Listing 20-1, I do the same kind of thing — I create instances of the JFrame, ImageIcon, and JLabel classes with the following lines:

```
var frame = new JFrame();
var icon = new ImageIcon("androidBook.jpg");
var label = new JLabel(icon);
```

Here's some gossip about each of these lines:

- **A JFrame is like a window (except that it's called a JFrame, not a window).** In Listing 20-1, the line

  ```
  var frame = new JFrame();
  ```

  creates a JFrame object, but this line doesn't display the JFrame object anywhere. (The displaying comes later in the code.)

- **An ImageIcon object is a picture.** At the root of the program's project directory, I have a file named androidBook.jpg. That file contains the picture shown earlier, in Figure 20-1. So, in Listing 20-1, the line

  ```
  var icon = new ImageIcon("androidBook.jpg");
  ```

  creates an ImageIcon object — an icon containing the androidBook.jpg picture.

  REMEMBER

  For some reason that I'll never understand, you may not want to use my androidBook.jpg image file when you run Listing 20-1. You can use almost any .gif, .jpg, or .png file in place of my (lovely) Android book cover image. To do so, copy your own image file to IntelliJ's Project tool window. (Drag it to the root of this example's project folder.) Then, in IntelliJ's editor, change the name androidBook.jpg to your own image file's name. That's it!

- **I need a place to put the icon.** I can put it on something called a JLabel. So, in Listing 20-1, the line

  ```
  var label = new JLabel(icon);
  ```

  creates a JLabel object and puts the androidBook.jpg icon on the new label's face.

If you read the previous bullets, you may get a false impression. The wording may suggest that the use of each component (JFrame, ImageIcon, JLabel, and so on) is a logical extension of what you already know. "Where do you put an ImageIcon? Well, of course, you put it on a JLabel." When you've worked long and hard with Java's Swing components, all these things become natural to you. But until then, you look up everything in Java's API documentation.

REMEMBER

You never need to memorize the names or features of Java's API classes. Instead, you keep Java's API documentation handy. When you need to know about a class, you look it up in the documentation. If you need a certain class often enough, you'll remember its features. For classes that you don't use often, you always have the docs.

For tips on using Java's API documentation, see my article "Making Sense of Java's API Documentation," at

```
www.dummies.com/programming/java/making-sense-of-javas-
        api-documentation
```

A class act

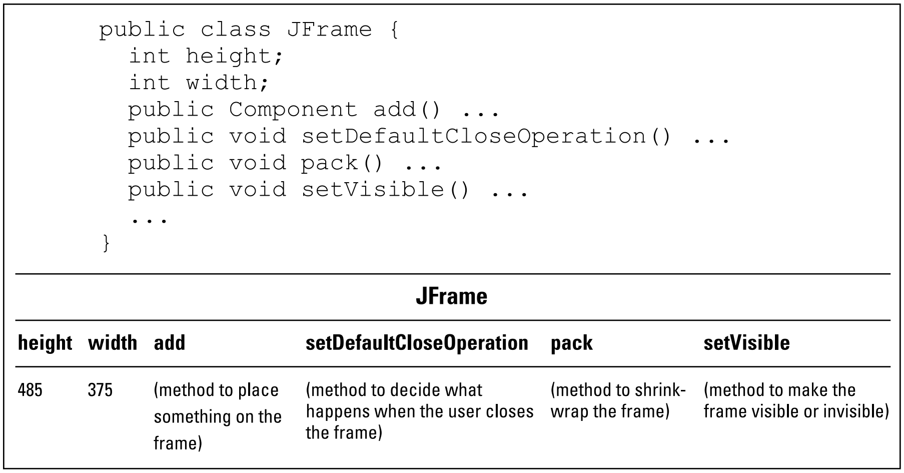
What is a JFrame? Like any other class, a JFrame has several parts. For a simplified view of some of these parts, see Figure 20-2.

Like the String in Figure 14-6 (in Chapter 14), the JFrame class has both fields and methods. The fields include the frame's height and width. The methods include add, setDefaultCloseOperation, pack, and setVisible. All told, the JFrame class has about 320 methods.

TECHNICAL STUFF

For technical reasons too burdensome for this book, you can't refer to the height and width fields of a JFrame with statements like frame.height = 485 or myWidth = frame.width. Those statements don't work. But you can use methods that access those fields. For example, you can write myWidth = frame.getWidth() or frame.setSize(375, 485). For more information, visit

```
https://docs.oracle.com/en/java/javase/17/docs/api/java.desktop/
        javax/swing/JFrame.html
```



```
public class JFrame {
  int height;
  int width;
  public Component add() ...
  public void setDefaultCloseOperation() ...
  public void pack() ...
  public void setVisible() ...
  ...
}
```

JFrame

height	width	add	setDefaultCloseOperation	pack	setVisible
485	375	(method to place something on the frame)	(method to decide what happens when the user closes the frame)	(method to shrink-wrap the frame)	(method to make the frame visible or invisible)

FIGURE 20-2: A simplified depiction of the `JFrame` class.

Here's the scoop on the `JFrame` methods in Listing 20-1:

- The call `frame.add(label)` plops the label onto the frame. The label displays my `androidBook.jpg` picture, so this call makes the picture appear on the frame.
- A call to `frame.setDefaultCloseOperation` tells Java what to do when you try to close the frame. (In Windows, you click the X in the upper right corner, by the title bar. On a Mac, the little red Close button is in the frame's upper left corner.) For a frame that's part of a larger application, you may want the frame to disappear when you click the X or the Close button, but you probably don't want the application to stop running.

  In Listing 20-1, the frame is the entire application — the whole enchilada. When you click the X or the red Close button, you want the Java Virtual Machine to shut itself down. To make this happen, you call the `setDefaultCloseOperation` method with parameter `JFrame.EXIT_ON_CLOSE`. The other alternatives are described in this list:

  - `JFrame.HIDE_ON_CLOSE`: The frame disappears, but it still exists in the computer's memory.
  - `JFrame.DISPOSE_ON_CLOSE`: The frame disappears and no longer exists in the computer's memory.
  - `JFrame.DO_NOTHING_ON_CLOSE`: The frame still appears, still exists, and still does everything it did before you clicked the X. Nothing happens when you click X. So, with this `DO_NOTHING_ON_CLOSE` option, you can become quite confused.

WARNING

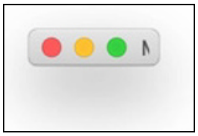
If you don't call setDefaultCloseOperation, Java automatically chooses the HIDE_ON_CLOSE option. When you click the X, the frame disappears but the Java program keeps running. Of course, with no visible frame, the running of Listing 20-1 doesn't do much. The only noticeable effect of the run is your development environment's behavior. With IntelliJ IDEA, a little square near the top of the main window retains its bright red color. When you hover over the square, you see the Stop tooltip. To end the Java program's run (and to return the square to its light-gray hue), simply click this little square.

- A frame's pack method shrink-wraps the frame around whatever has been added to the frame. Without calling pack, the frame can be much bigger or much smaller than is necessary.

  Unfortunately, the default is to make a frame much smaller than necessary. If, in Listing 20-1, you forget to call frame.pack(), you get the tiny frame shown in Figure 20-3. Sure, you can enlarge the frame by dragging the frame's edges with the mouse. But why should you have to do that? Just call frame.pack() instead.

- Calling setVisible(true) makes the frame appear on your screen. If you forget to call setVisible(true) (and I often do), when you run the code in Listing 20-1, you see nothing on your screen. It's always disconcerting until you figure out what you did wrong.

FIGURE 20-3: On a Mac, a frame that hasn't been packed or otherwise resized.



Constructor calls

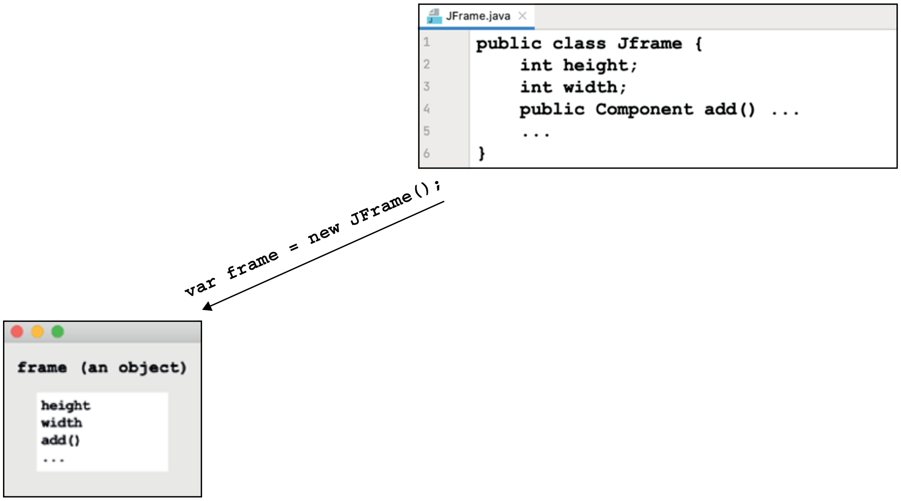
In Listing 13-5 (in Chapter 13), I created an instance of the Purchase class with the line

```
var purchase1 = new Purchase();
```

The code in Listing 20-1 does the same kind of thing. In Listing 20-1, I create an instance of the JFrame class with the following line:

```
var frame = new JFrame();
```

Compare Figure 13-4 (in Chapter 13) with this chapter's Figure 20-4.



JFrame.java

```
public class Jframe {
    int height;
    int width;
    public Component add() ...
    ...
}
```

FIGURE 20-4: An object created from the JFrame class.

In both figures, a new *SomethingOrOther*() call creates an object from an existing class:

- **In Chapter 13, I create an instance of my Purchase class.**

  This object represents an actual purchase (with a purchase amount and a tax, for example).

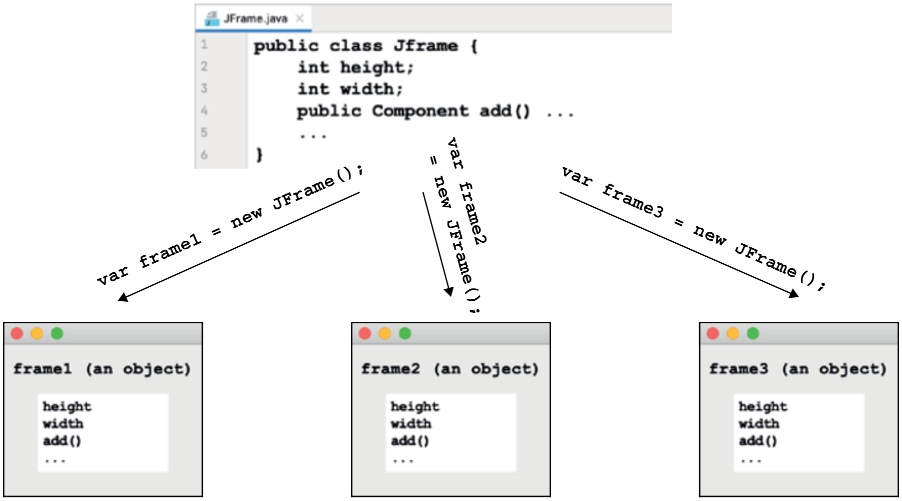
- **In this chapter, I create an instance of the JFrame class.**

  This object represents a frame on the computer screen (a frame with borders, a Close button, and so on). In a more complicated application — an app that displays several frames — the code might create several objects from a class such as JFrame. (See Figure 20-5.)

In Listing 20-1, the lines

```
var frame = new JFrame();
var icon = new ImageIcon("androidBook.jpg");
var label = new JLabel(icon);
```

look as though they contain method calls. After all, a method call consists of a name followed by parentheses. You might put some parameters between the open and close parentheses. The expression keyboard.nextLine() is a call to a method named nextLine. So, in Listing 20-1, is JFrame() a call to a method named JFrame? No, it's not.



```
public class Jframe {
    int height;
    int width;
    public Component add() ...
    ...
}
```

FIGURE 20-5: Creating three objects from the JFrame class.

In the expression `new JFrame()`, Java's `new` keyword signals a call to a constructor. A constructor is like a method, except that a constructor's name is the same as the name of a Java class. Java's standard API contains classes named `JFrame`, `ImageIcon`, and `JLabel`, and the code in Listing 20-1 calls the `JFrame`, `ImageIcon`, and `JLabel` constructors.

As the terminology suggests, a *constructor* is a piece of code that constructs an object. So, in Listing 20-1, when you call

```
var frame = new JFrame();
```

you make a `frame` variable refer to a newly constructed object (an object constructed from the `JFrame` class). In the same way, the declaration

```
var purchase1 = new Purchase();
```

in Chapter 13 makes `purchase1` refer to a new `Purchase` object using the `new Purchase()` constructor call.

Constructors and methods have a lot in common with one another. You can't call a method without having a corresponding method declaration somewhere in the code. (In the case of Java's `nextLine` method, the method declaration lives somewhere inside Java's enormous bunch of API classes.) The same is true of constructors. You can't call `new JFrame()` without having a constructor for the `JFrame` class somewhere in your code. And, sure enough, inside the Java API class, you can find a declaration for the `JFrame()` constructor. The code looks something like this:

```
public class JFrame {
    int height;
    int width;
    public Component add() ...
    public void setDefaultCloseOperation() ...
    public void pack() ...
    public void setVisible() ...
    ...

    /**
     * Constructs a new frame that is initially invisible.
     */
    public JFrame() {
        ...
    }
    ...
}
```

The constructor declaration looks almost like a method declaration. But notice that the constructor declaration doesn't start with `public void JFrame()` or with `public double JFrame()` or with `public` *anything* `JFrame()`. Aside from the optional word `public`, a constructor declaration contains only the name of the class whose object is being constructed. More on this in the next section.

A division of labor

In your Java-related travels, you see several variations on the code in Listing 20-1. This section explores one such variation.

This section's example does exactly what the previous section's example does — the only difference is the way the two examples deal with the `JFrame` class. This section's code is in Listings 20-2 and 20-3.

LISTING 20-2: Extending Java's JFrame Class

```
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame extends JFrame {
    public MyFrame() {
        var icon = new ImageIcon("androidBook.jpg");
        var label = new JLabel(icon);

        add(label);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
}
```

LISTING 20-3: Making a MyFrame Object

```
public class ShowPictureAgain {

    public static void main(String[] args) {
        new MyFrame();
    }
}
```

REMEMBER

To run the code in Listings 20-2 and 20-3, one IntelliJ project must contain two different files — a file named `MyFrame.java` and another file named `ShowPictureAgain.java`. For details, refer to Chapter 13.

Frame changer

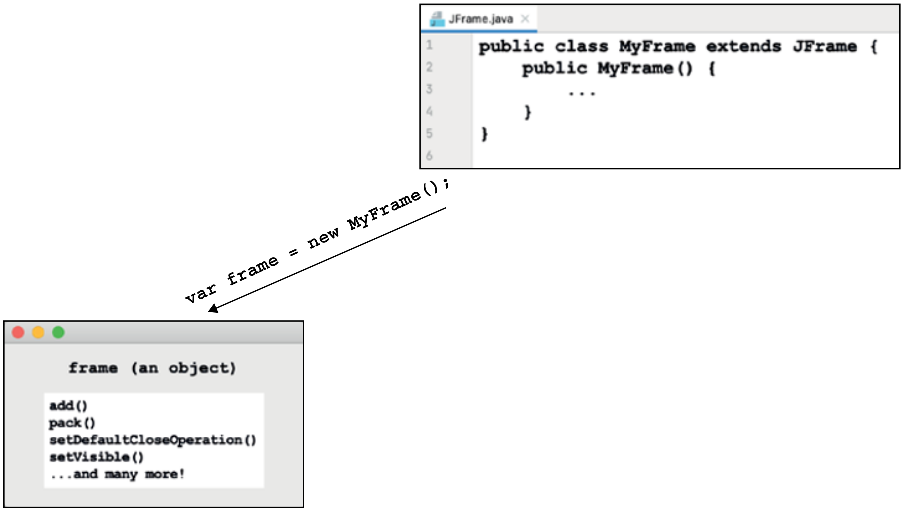
In Listing 20-2, the words `extends JFrame` are particularly important. When you see Java's `extends` keyword, imagine replacing that keyword with the phrase *is a kind of:*

```
public class MyFrame is a kind of JFrame {
```

When you type `MyFrame extends JFrame`, you declare that your new `MyFrame` class has the fields and methods that are built into Java's own `JFrame` class, and possibly more. For example, a `JFrame` instance has `setDefaultCloseOperation`, `pack`, and `setVisible` methods, so every new `MyFrame` instance has `setDefaultCloseOperation`, `pack`, and `setVisible` methods (see Figure 20-6).

When you put the words `extends JFrame` in your code, you get the `JFrame` methods for free. The `MyFrame` class's code doesn't need declarations for methods, such as `setDefaultCloseOperation`, `pack`, and `setVisible`. Those declarations are already in the `JFrame` class in Java's API. The only declarations in the `MyFrame` class's code are for brand-new things — things that are specific to your newly declared MyFrame class. It's as though Listing 20-2 contained the following information:

```
public class MyFrame is a kind of JFrame {

    // And in addition to what's in JFrame, MyFrame also has
    //         a brand new constructor:
    public MyFrame() {
        // Etc.
    }
}
```



```
public class MyFrame extends JFrame {
    public MyFrame() {
        ...
    }
}
```

FIGURE 20-6: A MyFrame instance has many methods.

In Listing 20-3, the words new MyFrame() get the MyFrame constructor to do its work. And the constructor in Listing 20-2 does quite a bit of work! The constructor does the stuff that the main method does in Listing 20-1:

- The constructor creates an ImageIcon containing the androidBook.jpg picture.
- The constructor creates a JLabel object and puts the androidBook.jpg icon on the new label's face.
- The constructor adds the JLabel object.

  Time out! What's being added to what? In Listing 20-1, the statement

  ```
  frame.add(label);
  ```

adds the `JLabel` object to the `frame`. But in Listing 20-2, there's no `frame` variable. In Listing 20-2, all you have is

```
add(label);
```

Well, here's the good news: Inside a constructor declaration, the object that you're constructing is "a given." You don't name that new object in order to refer to that new object. It's as though the constructor's code looked like this:

```
public MyFrame() {
    var icon = new ImageIcon("androidBook.jpg");
    var label = new JLabel(icon);
    new_frame_that_is_being_constructed.add(label);
    new_frame_that_is_being_constructed
            .setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    new_frame_that_is_being_constructed.pack();
    new_frame_that_is_being_constructed.setVisible(true);
}
```

Here's how the constructor in Listing 20-2 finishes its work:

- The constructor adds the `JLabel` object to the `MyFrame` object that's being constructed.
- The constructor tells the Java Virtual Machine to shut itself down when you close the frame.
- The constructor shrink-wraps the frame around the image that appears on the frame.
- The constructor makes the frame appear on your screen.

The `extends` keyword adds a fundamental idea to Java programming — the notion of *inheritance.* In Listing 20-2, the newly created `MyFrame` class *inherits* fields and methods that are declared in the existing `JFrame` class. Inheritance is a pivotal feature of an object-oriented programming language.

TRY IT OUT

Make some changes to this chapter's programs.

MIX IT UP

Change the order of the statements in the `main` method of Listing 20-1. Do the same with the statements in the constructor of Listing 20-2. Does the ordering of the statements make a difference? Must any statements come before other statements in the code?

MAKE IT YOUR OWN

Run the code in Listing 20-1 with your own image file in place of my `androidBook.jpg` file. Do the same with the code in Listings 20-2 and 20-3.

Drag-and-Drop for GUI Greatness

GUI programs have these two interesting characteristics:

- **GUI programs typically contain lots of code.**

  Much of this code differs little from one GUI program to another.

- **GUI programs involve visual elements.**

  The best way to describe visual elements is to "draw" them. Describing them with code can be slow and unintuitive.

To make your GUI life easier, you can use IntelliJ's GUI Designer to describe your program visually, and then it automatically turns your visual description into a Java application.

Hello, GUI Designer

Consider this wise old saying: "A picture is worth a thousand words." In this section, you use pictures instead of words to create a new GUI window. Follow these steps:

1. **Use IntelliJ IDEA to create a new Java project.**

   As you march through IntelliJ's dialog boxes, don't put a check mark in the Create Project from Template check box.

   When you finish creating the new project, IntelliJ shows you a main window.

2. **In IntelliJ's Project tool window, right-click your project's `src` folder.**

3. **From the resulting context menu, choose New ⇨ Swing UI Designer ⇨ GUI Form.**

   TIP

   When you choose New, is the Swing UI Designer menu item greyed-out? If so, you probably right-clicked the wrong folder. In Step 2, remember to click the `src` folder.

   If all goes well, a small dialog box appears. To absolutely no one's surprise, the box's title is New GUI Form.

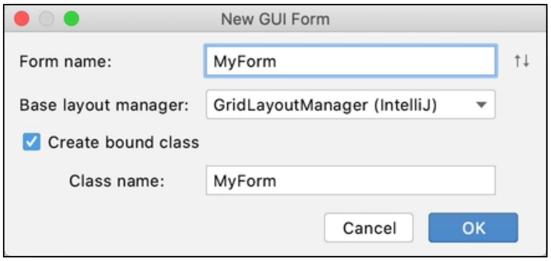
4. **In the dialog box's Form Name field, type** MyForm**. (See Figure 20-7.)**

WARNING

Don't put any blank spaces in the Form Name field. IntelliJ automatically copies your Form Name text into the dialog box's Class Name field, and you can't have a blank space in a Java class name.

Later, when you run the Java code, the word MyForm appears on the app's title bar. At that point in the game, you can add blank spaces.

5. **Put a check mark in the dialog box's Create Bound Class check box. (Refer to Figure 20-7.)**



Form name: MyForm

Base layout manager: GridLayoutManager (IntelliJ)

☑ Create bound class

Class name: MyForm

FIGURE 20-7: Creating a new form.

6. **Click OK to dismiss the New GUI Form dialog box.**

To paraphrase the movie *Babe*, "That'll do, dialog box."

When the dust settles, you see IntelliJ's GUI Designer. In the Designer's upper left corner, you see a component tree. The tree contains a branch labeled Form (MyForm). See Figure 20-8.

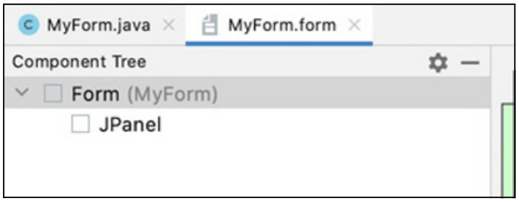
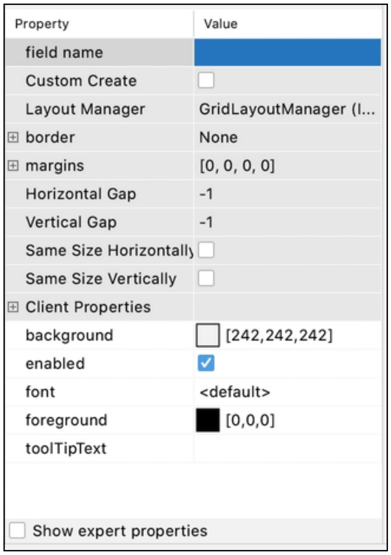


FIGURE 20-8: Components grow on trees.

7. **Expand the Form (MyForm) branch to reveal another branch; namely, the `JPanel` branch.**

When you select the `JPanel` branch, a list of properties and their values appears below the component tree. This list is called a *property sheet*. (See Figure 20-9.)

Java's API has a class named `JPanel`, and your new form contains an instance of that class. That instance's properties include its border, margins, background color, and several others. According to Figure 20-9, your `JPanel` object's background color is [242, 242, 242] — a slightly off shade of white. The trouble is, your `JPanel` object doesn't yet have a field name.



Property	Value
field name	
Custom Create	□
Layout Manager	GridLayoutManager (I...
⊞ border	None
⊞ margins	[0, 0, 0, 0]
Horizontal Gap	-1
Vertical Gap	-1
Same Size Horizontally	□
Same Size Vertically	□
⊞ Client Properties	
background	□ [242,242,242]
enabled	☑
font	<default>
foreground	■ [0,0,0]
toolTipText	

□ Show expert properties

FIGURE 20-9: The `JPanel` component's property sheet.

8. **In the `JPanel` object's property sheet, select the Value column of the Field Name row.**

9. **In that Field Name/Value cell, type the name `jpanel1` . . .**

   . . . or any other name, as long as it's a valid Java variable name.

   After naming the panel, you're finished with this section's GUI Designer instructions. All that's left for you to do is to create a bit of Java code. Fortunately, IntelliJ can write the code for you. Here's how:

10. **In IntelliJ's Project tool window, look for a `MyForm` branch inside of another `MyForm` branch, which is, in turn, inside the `src` branch. (See Figure 20-10.) Double-click that branch.**

    Your `MyForm` Java code appears in the IntelliJ editor. That code consists of an import declaration followed by a `public class MyForm` declaration.

11. **Right-click anywhere inside the `public class MyForm` declaration.**

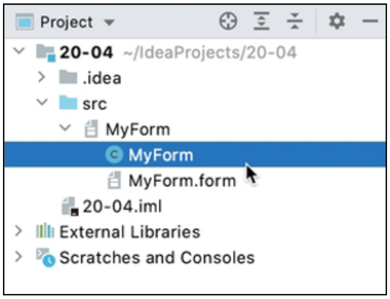


FIGURE 20-10: Finding the `MyForm` Java file.

12. **From the resulting context menu, choose Generate ➪ Form main().**

    If all goes well, IntelliJ adds a `main` method to your code. Nice!

    So far, your frame has no components in it, so you don't want to pack the frame. That would be like sucking the air out of an empty bag.

13. **In IntelliJ's editor, replace the line `frame.pack()` with `frame.setSize (200, 200)`. See Listing 20-4.**

    Ah, that's better! It's time to run your code.

14. **At the top of IntelliJ's editor, right-click the tab labeled `MyForm.java`. From the resulting context menu, choose Run 'MyForm.main()'.**

15. **Try not to be too excited when your computer displays the window shown in Figure 20-11.**

    The fact that this window contains no images, no buttons, no text fields — no *nothing* — comes from the way the GUI Designer creates your project. The designer populates the project with a minimum amount of code. That way, the new project is a blank slate — an empty shell to which you add buttons, text fields, or other useful components.

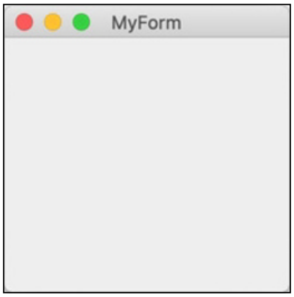


FIGURE 20-11: Blinded by the white.

LISTING 20-4: Enlarging a Frame

```
import javax.swing.*;

public class MyForm {
    private JPanel jpanel1;

    public static void main(String[] args) {
        JFrame frame = new JFrame("MyForm");
        frame.setContentPane(new MyForm().jpanel1);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 200);
        frame.setVisible(true);
    }
}
```

TECHNICAL STUFF

The code in Listing 20-4 uses features of Java that I don't cover in this book. If you want to read about these features, rush out and buy the most recent edition of *Java For Dummies*, by Barry Burd (and published by Wiley). That book starts from scratch, but it covers twice as much material as this beginning programming book.

Window dressing

I like empty spaces. When I lived on my own right out of college, my apartment had no pictures on the walls. I didn't want to stare at the same works of art day after day. I preferred to fill in the plain white spaces with images from my own imagination. So, for me, the empty window in Figure 20-11 is soothing.

But if Figure 20-11 isn't acquired by New York's Museum of Modern Art, the window is quite useless. (By the way, I'm still waiting to hear back from the museum's curator.) When you create a high-powered GUI program, you start by creating a window with buttons and other widgets. Then you add methods to respond to keystrokes, button clicks, and other such things.

The next section contains some code to respond to a user's button clicks. But in this section, you use IntelliJ's GUI Designer to describe a text field and a button.

Start with the project you created in this chapter's earlier section "Hello, GUI Designer":

1. **In the Project tool window, double-click the `MyForm.form` branch.**

   The GUI Designer appears once again.

2. **In the component tree, select `jpanel1`.**

   On the `jpanel1` property sheet, the Layout Manager row's Value cell contains a drop-down list.

3. **In the drop-down list, select FlowLayout.**

   When you start putting components on the panel, Java's `FlowLayout` positions them automatically.

   The middle of the GUI Designer looks like an empty square. This square is called the *Form Workspace*.

   The rightmost part of the GUI Designer is a palette. The *palette* offers you a choice of components such as `JButton`, `JCheckBox`, `JLabel`, `JTextField`, and many more. To create an application's window, you drag items from the palette to the Form Workspace. (See Figure 20-12.)

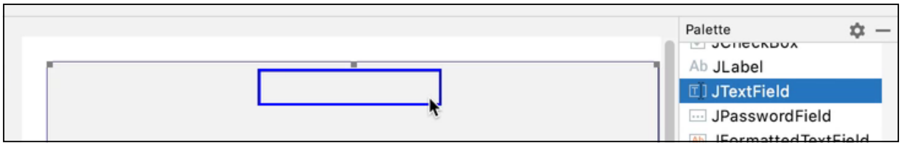


FIGURE 20-12: Dragging a `JTextField` component.

4. **Drag a `JTextField` component from the Palette to the Form Workspace.**

   Don't worry about the text field's size or position. Just plop the text field on the Form Workspace and march on to the next step.

   You don't even have to assign a name to the new text field. The first row of the text field's property sheet contains the name `textField1`. If you want, you can change this name, but I'm lazy. I don't want to change it.

5. **On the `textField1` property sheet, change the Columns value from 0 to 15.**

   As a result, the Form Workspace displays a wider text field.

6. **Drag a `JButton` component from the Palette to the Form Workspace.**

   IntelliJ automatically names this thing `button1`. Don't concern yourself with the position of `button1`. It is where it is.

7. **On the `button1` property sheet, change the `Text` value from `Button` to `Click Me`.**

   Happily, that change shows up in the Form Workspace. IntelliJ also changes the button's name from `button1` to `clickMeButton`. If you don't like the name `clickMeButton`, you can type a different name on the button's property sheet.

   Figure 20-13 shows you how the Form Workspace looks after all these steps.

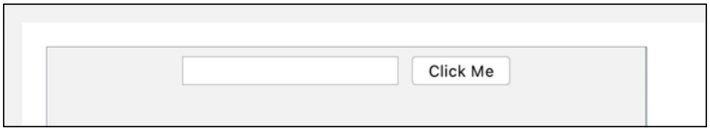


FIGURE 20-13: A preview of the new window.

8. **At the top of IntelliJ's editor, right-click the tab labeled `MyForm.java`. From the resulting context menu, choose Run 'MyForm.main()'.**

   When you do, the window shown in Figure 20-14 appears on your computer screen. Take a minute to stare at it and enjoy the fruits of your labor.

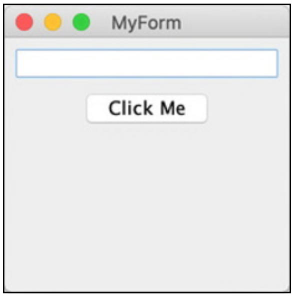


FIGURE 20-14: This makes me want to click the button!

When the project runs, the application doesn't do anything. When you click the button, nothing happens. When you type in the text field, nothing happens. What a waste!

In the next section, you get the button and the text field to do something.

INTELLIJ'S SECRETS

The Java code in Listing 20-4 produces the window in Figure 20-14. Do you notice anything unusual? The button in Figure 20-14 has the words *Click Me* on its face, but there's no mention of the string `"Click Me"` in the Java code. In Step 5 of the earlier section "Window dressing," you set the text field's Columns property to 15 but your Java code says nothing about Columns being 15. What gives?

While you were dragging-and-dropping GUI Designer components, IntelliJ was adding code to a file named `MyForm.form`. This `MyForm.form` file contains a textual description of the scene that you're creating in the designer's Form Workspace. You never have to look at that text, but you can if you want to. If you open `MyForm.form` with Windows Notepad or Mac TextEdit, you see the following crazy stuff:

```
<?xml version="1.0" encoding="UTF-8"?>
<form xmlns="http://www.intellij.com/uidesigner/form/" version="1"
    bind-to-class="MyForm">
  <grid id="27dc6" binding="jpanel1" layout-manager="FlowLayout"
    hgap="5" vgap="5" flow-align="1">
    <constraints>
      <xy x="20" y="20" width="500" height="400"/>
    </constraints>
    <properties/>
    <border type="none"/>
    <children>
      <component id="6e26c" class="javax.swing.JTextField"
          binding="textField1" default-binding="true">
        <constraints/>
        <properties>
          <columns value="15"/>
        </properties>
      </component>
      <component id="10c39" class="javax.swing.JButton"
          binding="clickMeButton" default-binding="true">
        <constraints/>
        <properties>
          <text value="Click Me"/>
        </properties>
      </component>
    </children>
  </grid>
</form>
```

This isn't Java code. It's eXtensible Markup Language (XML) code. In this code, you find the `columns` value `15`, the `text` value `"Click Me"`, the `layout-manager` `FlowLayout`, and lots of other good stuff. To add this XML information to your application, IntelliJ puts the following line in Listing 20-4:

```
frame.setContentPane(new MyForm().jpanel1);
```

Under the hood, IntelliJ groups some of its own Java files along with your `MyForm.java` and `MyForm.form` files to create one mighty `MyForm.class` file. When you run your app, IntelliJ runs the `MyForm.class` file.

I introduce `.class` files in Chapter 1. You don't need IntelliJ to run the `MyForm.class` file. Any computer with Java installed can run your GUI application.

Taking action

The window in Figure 20-14 looks nice, but it's quite useless. When you click the button, nothing happens. What good is that? In this section, you make the button respond to a mouse click. You do this by typing only one line of code! You start with the project you created in this chapter's earlier section "Window dressing."

1. **In the Project tool window, double-click the `MyForm.form` branch.**

   The GUI Designer appears once again.

2. **In the component tree, right-click the `clickMeButton`.**

   A context menu appears.

3. **From the context menu, choose Create Listener.**

   A pop-up with the title Create Listener appears. The pop-up contains a long list of items. (See Figure 20-15.)

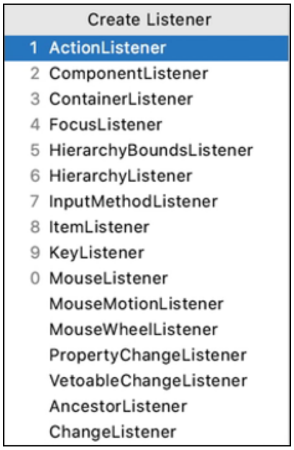


FIGURE 20-15: Listeners galore!

4. **From the Create Listener pop-up, choose `ActionListener`.**

   A dialog box labeled Select Methods to Implement appears. (See Figure 20-16.)

5. **In the dialog box, double-click `actionPerformed(e:ActionEvent):void`.**

   IntelliJ returns you to the editor where `MyForm.java` has magically acquired some new lines of code. These new lines "listen" for a user to click the button. To be a bit more precise, when a user clicks the button, Java responds by calling a newly created `actionPerformed` method. It's your job to put instructions inside the `actionPerformed` method.

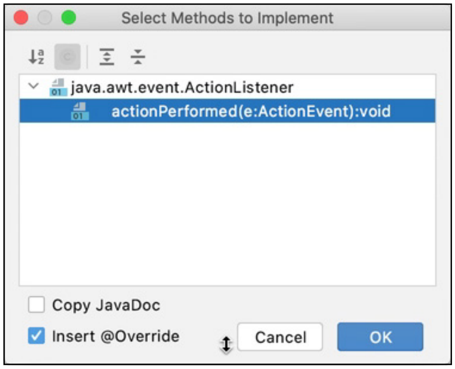

Select Methods to Implement
java.awt.event.ActionListener
actionPerformed(e:ActionEvent):void
□ Copy JavaDoc
☑ Insert @Override
Cancel
OK

FIGURE 20-16: Not much choice here.

6. **To the code in the editor, add one new line — the bold line in Listing 20-5.**

   Your new line of code tells Java to do the following:

   - `textField1.getText`: Get whatever characters the user has typed in the text field.
   - `toUpperCase`: Create an uppercase copy of those characters.
   - `textField1.setText`: Put the uppercase copy back into the text field.

   So much for all your sly maneuvering. It's time to run the code.

7. **At the top of IntelliJ's editor, right-click the tab labeled `MyForm.java`. From the resulting context menu, choose Run 'MyForm.main()'.**

   When you run the program, you see something like the screen shots in Figures 20-17, 20-18, and 20-19. *Et voilà!* When you click the button, Java capitalizes your text!

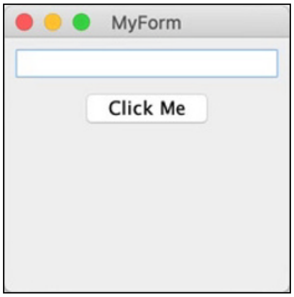
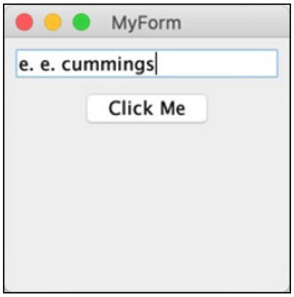
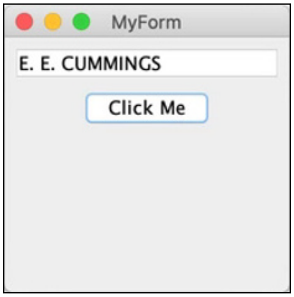


FIGURE 20-17: A brand-new frame.



e. e. cummings

FIGURE 20-18: The user types in the text box.



E. E. CUMMINGS

FIGURE 20-19: Clicking the button capitalizes the text in the text box.

LISTING 20-5: **The Bold and the Beautiful**

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyForm {
    private JPanel jpanel1;
    private JTextField textField1;
    private JButton clickMeButton;

    public MyForm() {
        clickMeButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                textField1.setText(textField1.getText().toUpperCase());
            }
        });
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("MyForm");
        frame.setContentPane(new MyForm().jpanel1);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 200);
        frame.setVisible(true);
    }
}
```

TRY IT OUT

Tweak the program in this section for some exciting results. (Okay, the results may be uninteresting to some people, but they're exciting to me!)

COPY TEXT TO A TEXT FIELD

Use IntelliJ's GUI Designer to create a frame with a button and two text fields. When the user clicks the button, Java copies text from the first text field to the second text field.

COPY TEXT TO A LABEL

In Java Swing, a label is similar to a text field. Like a text field, a label may display text. In fact, Swing's `JLabel` component has a `setText` method, like the `setText` method used in Listing 20-5.

But a label looks different. A label's appearance doesn't invite the user to change the label's text. A label looks like text that's been planted permanently on the frame. Of course, the text isn't permanent, because your code can change a label's text.

Use the GUI Designer to create a frame with a `JButton`, a `JTextField`, and a `JLabel`. When the user clicks the button, Java copies text from the text field to the label.

COPY BETWEEN TEXT FIELDS

Use the GUI Designer to create a frame with two buttons and two text fields. When the user clicks the first button, Java copies text from the first text field to the second text field. When the user clicks the second button, Java copies text from the second text field to the first text field.

Hint: Follow Steps 2 through 6 two times in this chapter's earlier section "Taking action" — once for `button1` and a second time for `button2`.

CREATE A SMALL CALCULATOR

Use the GUI Designer to create a frame with two text fields, a button, and a label. The user types a number in one of the text fields and another number in the other text field. When the user clicks the button, Java displays the sum of the two numbers in the label.

Hint: Whatever Java gets from a text field's `getText` method has `String` type. When you put a plus sign (+) between two `String` values, Java simply pastes the values together — for example, `"42" + "98"` is `"4298"`. Before you can add these values together, you have to convert them to numbers. You do this with Java's `Integer.parseInt` method. (Refer to Chapter 14.)

When you call a label's `setText` method, the call's parameter must have the `String` type. To get a `String` value from an `int` value, use the `Integer.toString` method — for example, `Integer.toString(86)` is `"86"`.

6

The Part of Tens

IN THIS PART . . .

Seeing the tip of the iceberg — a few of Java's most useful classes

The Dear Barry advice column

IN THIS CHAPTER

» Finding out more about classes

» Discovering other helpful classes

Chapter 21
Ten Useful Classes in the Java API

I'm proud of myself. I've written around 400 pages in this book about Java using fewer than 30 classes from the Java API. The standard API has thousands of classes, so I think I'm doing very well.

Anyway, to help acquaint you with some of my favorite Java API classes, this chapter contains a brief list. Some of the classes in this list appear in examples throughout this book. Others are so darn useful that I can't finish the book without including them.

For more information on the classes in this chapter, check Java's online API documentation at `https://docs.oracle.com/en/java/javase/17/docs/api`.

ArrayList

How often do you deal with many things at once? You have many customers, many contacts, many bills to pay, many social media posts — many, many, many! Each Java collection class has its own, special characteristics. A `Queue` stores things as they wait to take their turn. A `Stack` ignores older things in favor of more recent things. A `Set` has no ordering. A `Map` connects names with their values.

My go-to collection class is the ArrayList. An ArrayList is like an array, except that ArrayList objects grow and shrink as needed. You can also insert new values without pain using the ArrayList class's add method. ArrayList objects are useful because they do all kinds of nice things that arrays can't do.

To get started with the ArrayList class, refer to Chapter 19.

File

Talk about your useful Java classes! The File class does a bunch of things that aren't included in this book's examples. Method canRead tells you whether you can read from a file. Method canWrite tells you whether you can write to a file. Calling method setReadOnly ensures that you can't accidentally write to a file. Method deleteOnExit erases a file, but not until your program stops running. Method exists checks to see whether you have a particular file. Methods isHidden, lastModified, and length give you even more information about a file. You can even create a new directory by calling the mkdir method. Face it: This File class is powerful stuff!

Want to read about the File class? Refer to Chapters 12 and 16.

Integer

Chapter 14 describes the Integer class and its parseInt method. The Integer class has lots of other features that come in handy when you work with int values. For example, Integer.MAX_VALUE stands for the number 2147483647. That's the largest value that an int variable can store. (Refer to Chapter 7.) The expression Integer.MIN_VALUE stands for the number -2147483648 (the smallest value that an int variable can store). A call to Integer.toBinaryString takes an int and returns its base 2 (binary) representation. And what Integer.toBinaryString does for base 2, Integer.toHexString does for base 16 (hexadecimal).

JFrame

Chapter 20 has a JFrame example. A JFrame can be the starting point for an app's appearance on the screen. A JFrame is like a window, so you can put buttons, text fields, and other useful widgets on a JFrame. In Chapter 20, I put an image on a JFrame.

A JFrame has one of several different layouts. For example, a *border layout* divides the JFrame into five regions: the NORTH, SOUTH, EAST, WEST, and CENTER regions. An item in the CENTER region is usually the largest. It's the centerpiece of the JFrame. Items in the other border layout regions live on the four edges of the JFrame.

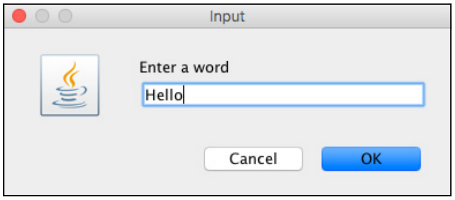
With a *grid layout,* you put items into table cells, and with a *flow layout,* you place items one after another in a row.

JOptionPane

For a quick-and-easy graphical user interface, use a JOptionPane. Here's some code:

```
String word = JOptionPane.showInputDialog("Enter a word");
JOptionPane.showMessageDialog(null, word);
String string = JOptionPane.showInputDialog("Enter an int value");
int number = Integer.parseInt(string);
number++;
JOptionPane.showMessageDialog(null, "One more is " + number);
```

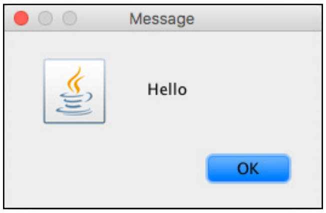
When you run this code, you see the dialog boxes in Figures 21-1 through 21-4.



Enter a word

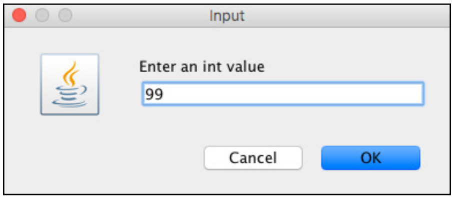
Hello

FIGURE 21-1: Input dialog box.



Hello

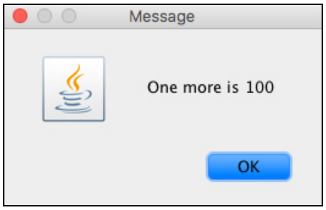
FIGURE 21-2: Message dialog box.



Enter an int value

99

FIGURE 21-3: Another input dialog box.



One more is 100

FIGURE 21-4: Another message dialog box.

Math

Do you have any numbers to crunch? Do you use your computer to do exotic calculations? If so, try Java's `Math` class. (It's a piece of code, not a place to sit down and listen to lectures about algebra.) The `Math` class deals with uc, *e*, logarithms, trig functions, square roots, and all those other mathematical doodads that give most people the creeps.

NumberFormat

Chapter 14 has a section about the `NumberFormat.getCurrencyInstance` method. With this method, you can turn `20.338500000000003` into `$20.34`. If the United States isn't your home, or if your company sells products worldwide, you can enhance your currency instance with a Java `Locale`. For example, with `euro = NumberFormat.getCurrencyInstance(Locale.FRANCE)`, a call to `euro.format(3)` returns `3,00 €` instead of `$3.00`.

The `NumberFormat` class also has methods for displaying things that aren't currency amounts. For example, you can display a number with or without commas, with or without leading zeros, and with as many digits beyond the decimal point as you care to include.

Scanner

Java's `Scanner` class can do more than what it does in this book's examples. Like the `NumberFormat` class, the `Scanner` can handle numbers from various locales. For example, to input 3,5 and have it mean "three-and-a-half," you can type `myScanner.useLocale(Locale.FRANCE)`. You can also tell a `Scanner` to skip certain input strings or use numeric bases other than 10. All in all, the `Scanner` class is quite versatile.

String

Chapter 14 examines Java's `String` class. The chapter describes (in gory detail) a method named `equals`. The `String` class has many other useful methods. For example, with the `length` method, you find the number of characters in a string. With `replaceAll`, you can easily change the phrase `"my fault"` to `"your fault"` wherever `"my fault"` appears inside a string. And, with `compareTo`, you can sort strings alphabetically.

System

You're probably familiar with `System.in` and `System.out`. What about `System.getProperty`? The `getProperty` method reveals all kinds of information about your computer. Some of the information you can find includes your operating system's name, your processor's architecture, your Java virtual machine version, your classpath, your username, and whether your system uses a backslash or forward slash to separate folder names from one another. Sure, you may already know all this stuff. But does your Java code need to discover it on the fly?

IN THIS CHAPTER

- What to learn
- How to learn more
- What to do when you're looking for a job
- Other free advice

Chapter 22
Ten Bits of Advice for New Software Developers

I enjoy hearing from the people who read my books. "Nice job!" one reader says. Another reader asks, "Can I run Java programs without installing IntelliJ IDEA?" Yet another posts this comment: "You're Barry Burd. Does that mean you're related to Larry Bird?"

In all the questions I receive from readers, one of the popular themes is "What to do next?" More specifically, people ask me what else to learn, what else to read, how to get practice writing software, how to find work, and other questions of that kind. I'm flattered to be asked, but I'm reluctant to think of myself as an authority on such matters. No two people give you the same answers to these questions, and if you ask enough people, you're sure to find disagreement.

This article contains ten pieces of advice based on questions I've received from readers. But remember that, in addition to these ten hints for living and learning, I have one additional, overriding piece of advice:

> Think critically about the advice you receive. When in doubt, trust your intuition.

Collect opinions. Talk to people about the issues. Try things and, if they work (or even if they don't work but they show some promise), keep doing them. If they show no promise, try other things. And, sharing is important. Don't forget to share.

How Long Does It Take to Learn Java?

The answer depends on you — on your goals, on your existing knowledge, on your capacity to think logically, on the amount of spare time you have, and on your interest in the subject.

The more excited you are about computer programming, the quicker you learn. The more ambitious your goals, the longer it takes to achieve them.

REMEMBER

There's no such thing as "knowing all about Java." No matter how much you know, you always have more to learn. I've written several Java books and, as far as I'm concerned, I've barely scratched the surface.

Which of Barry's Books Should I Read?

Funny you should ask! I've written several books, including these three:

- *Beginning Programming with Java For Dummies*
- *Java For Dummies*
- *Java Programming for Android Developers For Dummies*

Each book starts from scratch, so you don't need to know anything about app development to read any of these books. But each book covers (roughly) twice as much material as the previous book in the list. For example, *Java For Dummies* goes twice as fast and covers twice as much material as *Beginning Programming with Java For Dummies*. Which book you read depends on your level of comfort with technical subjects. If you're in doubt about where to start, find some sample pages from any of these books to help you determine which book is best for you.

Are Books Other than Barry's Good for Learning Java and Android Development?

Yes. I'd love to recommend some, but I'm not conscientious enough to carefully read and review other peoples' books.

Which Computer Programming Language(s) Should I Learn?

The answer depends on your goals and (if you plan to work as a developer) on-the-job opportunities where you live. The TIOBE Programming Community Index (`www.tiobe.com/index.php/content/paperinfo/tpci`) provides monthly ratings for popular programming languages. But the TIOBE Index might not apply specifically to your situation. In June 2021, the Haskell language ranks only 47th among the languages used around the world. But maybe there's a hotbed of Haskell programming in the town where you live.

Do you want to write applications for large enterprises? Then Java is a must-have language. Do you want to write code for the iPhone? You probably want to learn Swift. Do you want to create web pages? Learn HTML, CSS, and JavaScript.

What Skills Other than Computer Coding Should I Learn?

Sorry to disappoint you, but you're asking someone who has an ax to grind. I'm a college professor. I believe that no learning, no matter how impractical it might seem to be, is ever wasted.

If you insist on a more definitive answer, go learn a little about databases. Database work isn't necessarily coding, but it's important stuff. Also, read as much as you can about *software engineering* — the study of techniques for the effective design and maintenance of computer code. Don't be afraid of math, either (because learning math stretches your logical-thinking muscles). And, whenever you can, hone your communication skills. The better you communicate, the more valuable your work is to other people.

How Should I Continue My Learning as a Software Developer?

Practice, practice, practice. Take the examples you find in my book (or anywhere else) and think of ways to change the code. Add an option here or a button there.

Find out what happens when you try to improve the code. If it works, think of another way to make a change. If it doesn't work, search the documentation for a solution to your problem. If the documentation doesn't help (and often, documentation doesn't help) search the web for answers to your problem. Post questions at an online forum. If you don't find an answer, put the problem aside for a while and let it incubate in your mind.

REMEMBER

You don't learn programming by only reading about it — you have to scrape some knuckles while writing code and seeking solutions. Only after trying, failing, and trying again can you appreciate the work involved in developing computer software.

How Else Should I Continue My Learning as a Developer?

How did you know that I have a second suggestion? I recommend finding like-minded people where you live and getting together with them regularly. These days, you can find tech user groups in almost every corner of the globe. Find a Java user group that meets in your area and attend the group's meetings frequently. If you're a novice, you might not understand much of the discussion, but you'll be exposed to the issues that concern today's Java developers.

Look for more tech groups and attend their meetings. Find meetings about other programming languages, other technologies, and other topics that aren't solely about technology. Meet people face-to-face and find out which topics will be in next year's books.

To complement those face-to-face meetings, search the web for screen-to-screen meetings. You can find free online technical sessions almost any day of the year.

How Can I Get a Job Developing Software?

Do all the things you'd normally do when you look for a job, but don't forget about the advice in the previous two paragraphs. User groups are fantastic places for networking.

Go to meetings and be a good listener. Don't think about selling yourself. Be patient and enjoy the ride. I landed a great consulting opportunity only after several years of attending one group's meetings. In the meantime, I learned a lot about software (and quite a bit about dealing with other people).

I Still Don't Know What to Do with My Life

That's not a question. But it's okay anyway.

Everyone has to make ends meet. If you manage to put food on your table, the next step is to find out what you love to do. I've spent a lifetime teaching college students, writing books, and developing computer code — and I love doing all of it. (Well, I love most of it. I detest grading papers, and I dislike proofreading my own work.)

Fortunately, I can make money teaching, writing, and developing. I could make more money working 9-to-5 for a big company, managing a software team or creating the next big start-up, but I don't like doing those things. My life has been enriched because I do what I like doing, whether I'm working or not.

My advice is, find the best match of the things you like to do and the things that help you earn a living. Compromise, if you must, but be honest with yourself about the things that make you happy. (Of course, these things shouldn't make other people unhappy.)

Finally, be specific about your likes and dislikes. For example, saying, “I’d like to be rich” isn’t specific at all. Saying, “I’d like to create a great game” is more specific, but you can do better. Saying, “I like to design game software, but I need a partner who can do the marketing for me” is quite specific and makes quite a tidy set of goals.

If I Have Other Questions, How Can I Contact Barry Burd?

Send email to `BeginProg@allmycode.com`. Follow me on Facebook (`/allmycode`) or Twitter (`@allmycode`). Visit my ugly-but-informative website: `www.allmycode.com`. Attach two tin cans to a very long string. Put a note in an old pneumatic tube. Train a carrier pigeon to fly to my office. Hire a chimpanzee to. . ..

Index

About the Author

Barry Burd received an MS degree in Computer Science at Rutgers University and a PhD in Mathematics at the University of Illinois. As a teaching assistant in Champaign–Urbana, Illinois, he was elected five times to the university-wide List of Teachers Ranked as Excellent by Their Students.

Since 1980, Barry has been a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. He has spoken at conferences in the United States, Europe, Australia, and Asia. In 2020, he was named a Java Champion as part of a project sponsored by Oracle. He is the author of several articles and books, including *Java For Dummies*, *Flutter For Dummies*, and *Java Programming For Android Developers For Dummies,* all from Wiley Publishing.

Barry lives in Madison, New Jersey, with his wife of *n* years, where *n* > 40. As an avid indoors enthusiast, Barry enjoys sleeping, talking, and eating. You can reach him at `BeginProg@allmycode.com`.

Dedication

For

Abram and Katie, Benjamin and Jennie, Sam and Ruth, Harriet, Sam, and Jennie,

Author's Acknowledgments

I heartily and sincerely thank Paul Levesque, for his work on so many of my books in this series. Thanks also to Kelsey Baird, for her patience and support. Thanks to Becky Whitney, for keeping my grammar and punctuation in check. Thanks to the staff at Wiley Publishing, for helping to bring this book to bookshelves.

Thanks to Frank Greco and the leaders of the New York JavaSIG — Jeanne Boyarsky, Rodrigo Graciano, Chandra Guntur, Justin Lee, Sai Sharan Donthi, Lily Luo, and Vinay Karle. Thanks to Michael Redlich and the directors of the Garden State Java User Group — Paul Syers and Chandra G. (again). Thanks to my colleagues, the faculty members in the Mathematics and Computer Science Department at Drew University — Sarah Abramowitz, Chris Apelian, Seth Harris, Emily Hill, Steve Kass, Diane Liporace, Yi Lu, Ziyuan Meng, Ellie Small, and Steve Surace. Finally, a special thanks to Richard Bonacci, Peter Lubbers, Cameron McKenzie, Scott Stoll, and Gaisi Takeuti, for their long-term help and support.

Publisher's Acknowledgments

Acquisitions Editor: Kelsey Baird

Senior Project Editor: Paul Levesque

Copy Editor: Becky Whitney

Tech Editor: Chad Darby

Project Coordinator: Ram Prabakaran

Cover Image: © Anees Alangadan/Shutterstock; kowalskichal/Shutterstock

Take dummies with you everywhere you go!

Whether you are excited about e-books, want more from the web, must have your mobile apps, or are swept up in social media, dummies makes everything easier.

Find us online!

dummies.com

dummies®
A Wiley Brand

Leverage the power

Dummies is the global leader in the reference category and one of the most trusted and highly regarded brands in the world. No longer just focused on books, customers now have access to the dummies content they need in the format they want. Together we'll craft a solution that engages your customers, stands out from the competition, and helps you meet your goals.

Advertising & Sponsorships

Connect with an engaged audience on a powerful multimedia site, and position your message alongside expert how-to content. Dummies.com is a one-stop shop for free, online information and know-how curated by a team of experts.

- Targeted ads
- Video
- Email Marketing
- Microsites
- Sweepstakes sponsorship

20 MILLION PAGE VIEWS EVERY SINGLE MONTH

15 MILLION UNIQUE VISITORS PER MONTH

43% OF ALL VISITORS ACCESS THE SITE VIA THEIR MOBILE DEVICES

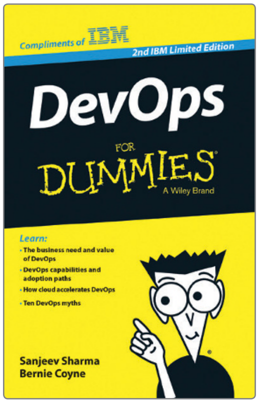
700,000 NEWSLETTER SUBSCRIPTIONS TO THE INBOXES OF 300,000 UNIQUE INDIVIDUALS EVERY WEEK

# of dummies

Custom Publishing

Reach a global audience in any language by creating a solution that will differentiate you from competitors, amplify your message, and encourage customers to make a buying decision.

- Apps
- Books
- eBooks
- Video
- Audio
- Webinars

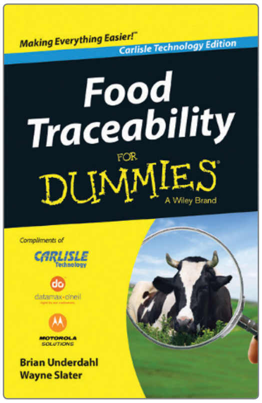


Compliments of IBM
2nd IBM Limited Edition

DevOps FOR DUMMIES® A Wiley Brand

Learn:
- The business need and value of DevOps
- DevOps capabilities and adoption paths
- How cloud accelerates DevOps
- Ten DevOps myths

Sanjeev Sharma
Bernie Coyne

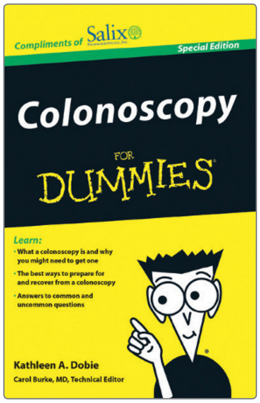


Making Everything Easier!™
Carlisle Technology Edition

Food Traceability FOR DUMMIES® A Wiley Brand

Compliments of
CARLISLE Technology
datamax-o'neil
MOTOROLA SOLUTIONS

Brian Underdahl
Wayne Slater

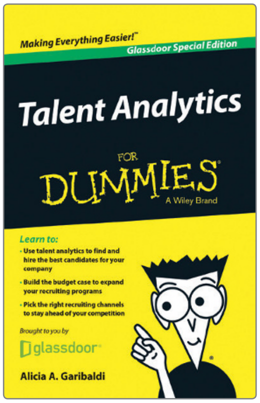


Compliments of Salix
Special Edition

Colonoscopy FOR DUMMIES®

Learn:
- What a colonoscopy is and why you might need to get one
- The best ways to prepare for and recover from a colonoscopy
- Answers to common and uncommon questions

Kathleen A. Dobie
Carol Burke, MD, Technical Editor



Making Everything Easier!™
Glassdoor Special Edition

Talent Analytics FOR DUMMIES® A Wiley Brand

Learn to:
- Use talent analytics to find and hire the best candidates for your company
- Build the budget case to expand your recruiting programs
- Pick the right recruiting channels to stay ahead of your competition

Brought to you by
glassdoor®

Alicia A. Garibaldi

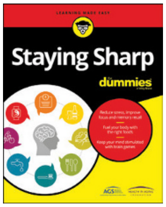
Brand Licensing & Content

Leverage the strength of the world's most popular reference brand to reach new audiences and channels of distribution.

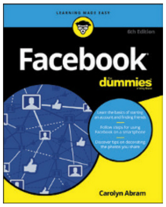
For more information, visit dummies.com/biz

dummies®
A Wiley Brand

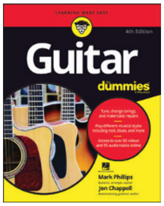
PERSONAL ENRICHMENT



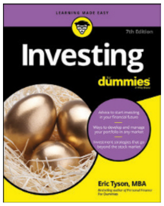
9781119187790
USA $26.00
CAN $31.99
UK £19.99



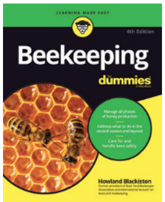
9781119179030
USA $21.99
CAN $25.99
UK £16.99



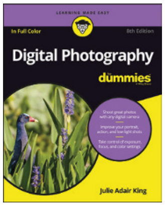
9781119293354
USA $24.99
CAN $29.99
UK £17.99



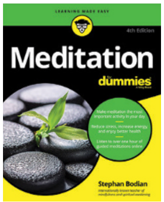
9781119293347
USA $22.99
CAN $27.99
UK £16.99



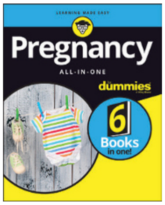
9781119310068
USA $22.99
CAN $27.99
UK £16.99



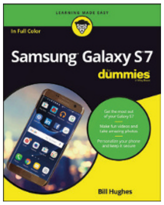
9781119235606
USA $24.99
CAN $29.99
UK £17.99



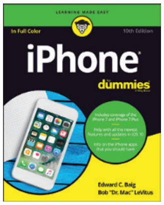
9781119251163
USA $24.99
CAN $29.99
UK £17.99



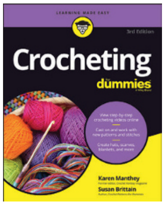
9781119235491
USA $26.99
CAN $31.99
UK £19.99



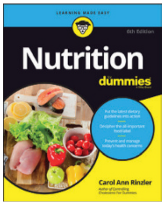
9781119279952
USA $24.99
CAN $29.99
UK £17.99



9781119283133
USA $24.99
CAN $29.99
UK £17.99

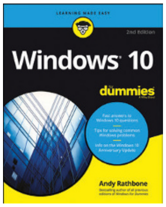


9781119287117
USA $24.99
CAN $29.99
UK £16.99

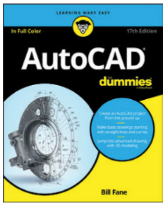


9781119130246
USA $22.99
CAN $27.99
UK £16.99

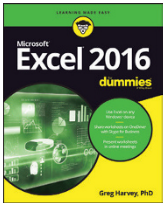
PROFESSIONAL DEVELOPMENT



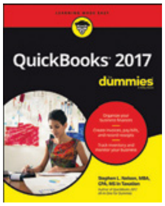
9781119311041
USA $24.99
CAN $29.99
UK £17.99



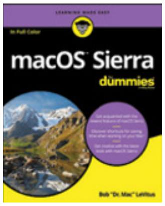
9781119255796
USA $39.99
CAN $47.99
UK £27.99



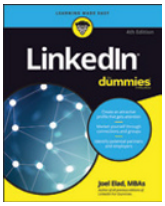
9781119293439
USA $26.99
CAN $31.99
UK £19.99



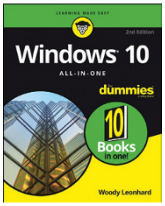
9781119281467
USA $26.99
CAN $31.99
UK £19.99



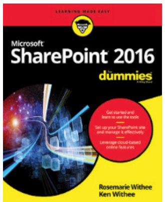
9781119280651
USA $29.99
CAN $35.99
UK £21.99



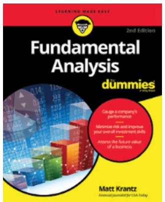
9781119251132
USA $24.99
CAN $29.99
UK £17.99



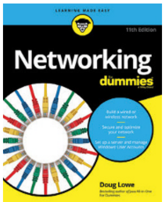
9781119310563
USA $34.00
CAN $41.99
UK £24.99



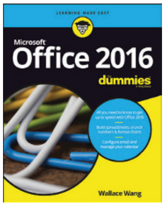
9781119181705
USA $29.99
CAN $35.99
UK £21.99



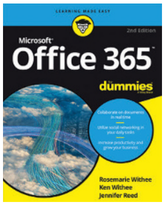
9781119263593
USA $26.99
CAN $31.99
UK £19.99



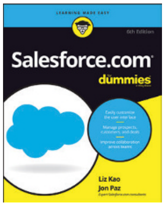
9781119257769
USA $29.99
CAN $35.99
UK £21.99



9781119293477
USA $26.99
CAN $31.99
UK £19.99



9781119265313
USA $24.99
CAN $29.99
UK £17.99



9781119239314
USA $29.99
CAN $35.99
UK £21.99

9781119293323
USA $29.99
CAN $35.99
UK £21.99

dummies.com

dummies®
A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.