Java®

7th Edition

by Barry Burd, PhD

for dummies®
A Wiley Brand

LEARNING MADE EASY

7th Edition

Java®

for dummies®
A Wiley Brand

Use the new features and tools in Java 9

Create basic Java objects and reuse code

Handle exceptions and events

Barry Burd, PhD

Author of Java Programming for Android Developers For Dummies

Contents at a Glance

Java® For Dummies®, 7th Edition

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

Published simultaneously in Canada

Library of Congress Control Number: 2017932837

ISBN: 978-1-119-23555-2; 978-1-119-23558-3 (ebk); 978-1-119-23557-6 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Table of Contents

Introduction

Java is good stuff. I've been using it for years. I like Java because it's orderly. Almost everything follows simple rules. The rules can seem intimidating at times, but this book is here to help you figure them out. So, if you want to use Java and you want an alternative to the traditional techie, soft-cover book, sit down, relax, and start reading *Java For Dummies*, 7th Edition.

How to Use This Book

I wish I could say, "Open to a random page of this book and start writing Java code. Just fill in the blanks and don't look back." In a sense, this is true. You can't break anything by writing Java code, so you're always free to experiment.

But let me be honest. If you don't understand the bigger picture, writing a program is difficult. That's true with any computer programming language — not just Java. If you're typing code without knowing what it's about and the code doesn't do exactly what you want it to do, you're just plain stuck.

In this book, I divide Java programming into manageable chunks. Each chunk is (more or less) a chapter. You can jump in anywhere you want — Chapter 5, Chapter 10, or wherever. You can even start by poking around in the middle of a chapter. I've tried to make the examples interesting without making one chapter depend on another. When I use an important idea from another chapter, I include a note to help you find your way around.

In general, my advice is as follows:

- If you already know something, don't bother reading about it.
- If you're curious, don't be afraid to skip ahead. You can always sneak a peek at an earlier chapter, if you really need to do so.

Conventions Used in This Book

Almost every technical book starts with a little typeface legend, and *Java For Dummies*, 7th Edition, is no exception. What follows is a brief explanation of the typefaces used in this book:

- New terms are set in *italics*.
- If you need to type something that's mixed in with the regular text, the characters you type appear in bold. For example: "Type **MyNewProject** in the text field."
- You also see this `computerese` font. I use computerese for Java code, filenames, web page addresses (URLs), onscreen messages, and other such things. Also, if something you need to type is really long, it appears in computerese font on its own line (or lines).
- You need to change certain things when you type them on your own computer keyboard. For instance, I may ask you to type

  ```
  public class Anyname
  ```

  which means that you type **public class** and then some name that you make up on your own. Words that you need to replace with your own words are set in *`italicized computerese`*.

What You Don't Have to Read

Pick the first chapter or section that has material you don't already know and start reading there. Of course, you may hate making decisions as much as I do. If so, here are some guidelines that you can follow:

- If you already know what kind of an animal Java is and know that you want to use Java, skip Chapter 1 and go straight to Chapter 2. Believe me, I won't mind.
- If you already know how to get a Java program running, and you don't care what happens behind the scenes when a Java program runs, skip Chapter 2 and start with Chapter 3.
- If you write programs for a living but use any language other than C or C++, start with Chapter 2 or 3. When you reach Chapters 5 and 6, you'll probably find them to be easy reading. When you get to Chapter 7, it'll be time to dive in.

- If you write C (not C++) programs for a living, start with Chapters 2, 3, and 4 and just skim Chapters 5 and 6.
- If you write C++ programs for a living, glance at Chapters 2 and 3, skim Chapters 4 through 6, and start reading seriously in Chapter 7. (Java is a bit different from C++ in the way it handles classes and objects.)
- If you write Java programs for a living, come to my house and help me write *Java For Dummies,* 8th Edition.

If you want to skip the sidebars and the Technical Stuff icons, please do. In fact, if you want to skip anything at all, feel free.

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 the good news: You can run most of the code in this book on almost any computer. The only computers that you can't use to run this code are ancient things that are more than ten years old (give or take a few years).
- **I assume that you can navigate through 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 the 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. When you reach one of these platform-specific tasks, try following the steps in this book. If the steps don't quite fit, consult a book with instructions tailored to your system.
- **I assume that you can think logically.** That's all there is to programming in Java — thinking logically. If you can think logically, you've got it made. If you don't believe that you can think logically, read on. You may be pleasantly surprised.

» **I make few assumptions about your computer programming experience (or your lack of such experience).** In writing this book, I've tried to do the impossible: I've tried to make the book interesting for experienced programmers yet accessible to people with little or no programming experience. This means that I don't assume any particular programming background on your part. If you've never created a loop or indexed an array, that's okay.

On the other hand, if you've done these things (maybe in Visual Basic, Python, or C++), you'll discover some interesting plot twists in Java. The developers of Java took the best ideas in object-oriented programming, streamlined them, reworked them, and reorganized them into a sleek, powerful way of thinking about problems. You'll find many new, thought-provoking features in Java. As you find out about these features, many of them will seem quite natural to you. One way or another, you'll feel good about using Java.

How This Book Is Organized

This book is divided into subsections, which are grouped into sections, which come together to make chapters, which are lumped finally into five parts. (When you write a book, you get to know your book's structure pretty well. After months of writing, you find yourself dreaming in sections and chapters when you go to bed at night.) The parts of the book are listed here.

Part 1: Getting Started with Java

This part is your complete, executive briefing on Java. It includes some "What is Java?" material and a jump-start chapter — Chapter 3. In Chapter 3, you visit the major technical ideas and dissect a simple program.

Part 2: Writing Your Own Java Program

Chapters 4 through 6 cover the fundamentals. These chapters describe the things that you need to know so that you can get your computer humming along.

If you've written programs in Visual Basic, C++, or any another language, some of the material in Part 2 may be familiar to you. If so, you can skip some sections or read this stuff quickly. But don't read too quickly. Java is a little different from some other programming languages, especially in the things that I describe in Chapter 4.

Part 3: Working with the Big Picture: Object-Oriented Programming

Part 3 has some of my favorite chapters. This part covers the all-important topic of object-oriented programming. In these chapters, you find out how to map solutions to big problems. (Sure, the examples in these chapters aren't big, but the examples involve big ideas.) In bite-worthy increments, you discover how to design classes, reuse existing classes, and construct objects.

Have you read any of those books that explain object-oriented programming in vague, general terms? I'm proud to say that *Java For Dummies*, 7th Edition, isn't like that. In this book, I illustrate each concept with a simple-yet-concrete program example.

Part 4: Smart Java Techniques

If you've tasted some Java and you want more, you can find what you need in this part of the book. This part's chapters are devoted to details — the things that you don't see when you first glance at the material. After you read the earlier parts and write some programs on your own, you can dive in a little deeper by reading Part 4.

Part 5: The Part of Tens

The Part of Tens is a little Java candy store. In the Part of Tens, you can find lists — lists of tips for avoiding mistakes, for finding resources, and for all kinds of interesting goodies.

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 in my head. Most of the sentences, I mutter several times. 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.

Where to Go from Here

If you've gotten this far, you're ready to start reading about Java application development. 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, to help you understand.

If you like what you read, send me a note. My email address, which I created just for comments and questions about this book, is JavaForDummies@allmycode.com. If email and chat aren't your favorites, you can reach me instead on Twitter (`@allmycode`) and on Facebook (`www.facebook.com/allmycode`). And don't forget — for the latest updates, visit this book's website. The site's address is `www.allmycode.com/JavaForDummies`.

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

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

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

Question: What's stronger than a Tip, but not as strong as a Warning?

Answer: A Remember icon.

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

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'll 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 (`www.allmycode.com/JavaForDummies`) for hints and solutions.

This icon calls attention to useful material that you can find online. Check it out!

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.

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 Android program. To get this Cheat Sheet, simply go to `www.dummies.com` and type **Java For Dummies Cheat Sheet** in the Search box.

1 Getting Started with Java

IN THIS CHAPTER

- What Java is
- Where Java came from
- Why Java is so cool
- How to orient yourself to object-oriented programming

Chapter 1
All about Java

Say what you want about computers. As far as I'm concerned, computers are good for just two simple reasons:

- **When computers do work, they feel no resistance, no stress, no boredom, and no fatigue.** Computers are our electronic slaves. I have my computer working 24/7 doing calculations for `Cosmology@Home` — a distributed computing project to investigate models describing the universe. Do I feel sorry for my computer because it's working so hard? Does the computer complain? Will the computer report me to the National Labor Relations Board? No.

  I can make demands, give the computer its orders, and crack the whip. Do I (or should I) feel the least bit guilty? Not at all.

- **Computers move ideas, not paper.** Not long ago, when you wanted to send a message to someone, you hired a messenger. The messenger got on his or her horse and delivered your message personally. The message was on paper, parchment, a clay tablet, or whatever physical medium was available at the time.

  This whole process seems wasteful now, but that's only because you and I are sitting comfortably in the electronic age. Messages are ideas, and physical things like ink, paper, and horses have little or nothing to do with real ideas; they're just temporary carriers for ideas (even though people used them to carry ideas for several centuries). Nevertheless, the ideas themselves are paperless, horseless, and messengerless.

IN THIS PART . . .

Find out about the tools you need for developing Java programs.

Find out how Java fits into today's technology scene.

See your first complete Java program.

The neat thing about computers is that they carry ideas efficiently. They carry nothing but the ideas, a couple of photons, and a little electrical power. They do this with no muss, no fuss, and no extra physical baggage.

When you start dealing efficiently with ideas, something very nice happens. Suddenly, all the overhead is gone. Instead of pushing paper and trees, you're pushing numbers and concepts. Without the overhead, you can do things much faster and do things that are far more complex than ever before.

What You Can Do with Java

It would be so nice if all this complexity were free, but unfortunately, it isn't. Someone has to think hard and decide exactly what to ask the computer to do. After that thinking takes place, someone has to write a set of instructions for the computer to follow.

Given the current state of affairs, you can't write these instructions in English or any other language that people speak. Science fiction is filled with stories about people who say simple things to robots and get back disastrous, unexpected results. English and other such languages are unsuitable for communication with computers, for several reasons:

- **An English sentence can be misinterpreted.** "Chew one tablet three times a day until finished."
- **It's difficult to weave a very complicated command in English.** "Join flange A to protuberance B, making sure to connect only the outermost lip of flange A to the larger end of the protuberance B, while joining the middle and inner lips of flange A to grommet C."
- **An English sentence has lots of extra baggage.** "Sentence has unneeded words."
- **English is difficult to interpret.** "As part of this Publishing Agreement between John Wiley & Sons, Inc. ('Wiley') and the Author ('Barry Burd'), Wiley shall pay the sum of one-thousand-two-hundred-fifty-seven dollars and sixty-three cents ($1,257.63) to the Author for partial submittal of *Java For Dummies,* 7th Edition ('the Work')."

To tell a computer what to do, you have to use a special language to write terse, unambiguous instructions. A special language of this kind is called a *computer programming language.* A set of instructions written in such a language is called a *program.* When looked at as a big blob, these instructions are called *software* or *code.* Here's what code looks like when it's written in Java:


```
public class PayBarry {

    public static void main(String args[]) {
        double checkAmount = 1257.63;
        System.out.print("Pay to the order of ");
        System.out.print("Dr. Barry Burd ");
        System.out.print("$");
        System.out.println(checkAmount);
    }
}
```

Why You Should Use Java

It's time to celebrate! You've just picked up a copy of *Java For Dummies,* 7th Edition, and you're reading Chapter 1. At this rate, you'll be an expert Java programmer* in no time at all, so rejoice in your eventual success by throwing a big party.

To prepare for the party, I'll bake a cake. I'm lazy, so I'll use a ready-to-bake cake mix. Let me see . . . add water to the mix and then add butter and eggs — hey, wait! I just looked at the list of ingredients. What's MSG? And what about propylene glycol? That's used in antifreeze, isn't it?

I'll change plans and make the cake from scratch. Sure, it's a little harder, but that way I get exactly what I want.

Computer programs work the same way. You can use somebody else's program or write your own. If you use somebody else's program, you use whatever you get. When you write your own program, you can tailor the program especially for your needs.

Writing computer code is a big, worldwide industry. Companies do it, freelance professionals do it, hobbyists do it — all kinds of people do it. A typical big company has teams, departments, and divisions that write programs for the company. But you can write programs for yourself or someone else, for a living or for fun. In a recent estimate, the number of lines of code written each day by programmers in the United States alone exceeds the number of methane molecules on the planet Jupiter.** Take almost anything that can be done with a computer. With the right amount of time, you can write your own program to do it. (Of course, the "right amount of time" may be very long, but that's not the point. Many interesting and useful programs can be written in hours or even minutes.)

*In professional circles, a developer's responsibilities are usually broader than those of a programmer. But, in this book, I use the terms programmer and developer almost interchangeably.

**I made up this fact all by myself.

Getting Perspective: Where Java Fits In

Here's a brief history of modern computer programming:

- **1954–1957: FORTRAN is developed.**

  FORTRAN was the first modern computer programming language. For scientific programming, FORTRAN is a real racehorse. Year after year, FORTRAN is a leading language among computer programmers throughout the world.

- **1959: Grace Hopper at Remington Rand develops the COBOL programming language.**

  The letter *B* in COBOL stands for *Business,* and business is just what COBOL is all about. The language's primary feature is the processing of one record after another, one customer after another, or one employee after another.

  Within a few years after its initial development, COBOL became the most widely used language for business data processing.

- **1972: Dennis Ritchie at AT&T Bell Labs develops the C programming language.**

  The "look and feel" that you see in this book's examples comes from the C programming language. Code written in C uses curly braces, `if` statements, `for` statements, and so on.

  In terms of power, you can use C to solve the same problems that you can solve by using FORTRAN, Java, or any other modern programming language. (You can write a scientific calculator program in COBOL, but doing that sort of thing would feel really strange.) The difference between one programming language and another isn't power. The difference is ease and appropriateness of use. That's where the Java language excels.

- **1986: Bjarne Stroustrup (again at AT&T Bell Labs) develops C++.**

  Unlike its C language ancestor, the language C++ supports object-oriented programming. This support represents a huge step forward. (See the next section in this chapter.)

- **May 23, 1995: Sun Microsystems releases its first official version of the Java programming language.**

  Java improves upon the concepts in C++. Java's "Write Once, Run Anywhere" philosophy makes the language ideal for distributing code across the Internet.

  Additionally, Java is a great general-purpose programming language. With Java, you can write windowed applications, build and explore databases,

control handheld devices, and more. Within five short years, the Java programming language had 2.5 million developers worldwide. (I know. I have a commemorative T-shirt to prove it.)

- **November 2000: The College Board announces that, starting in the year 2003, the Computer Science Advanced Placement exams will be based on Java.**

  Wanna know what that snot-nosed kid living down the street is learning in high school? You guessed it — Java.

- **2002: Microsoft introduces a new language, named C#.**

  Many of the C# language features come directly from features in Java.

- **June 2004: Sys-Con Media reports that the demand for Java programmers tops the demand for C++ programmers by 50 percent (`http://java.sys-con.com/node/48507`).**

  And there's more! The demand for Java programmers beats the combined demand for C++ and C# programmers by 8 percent. Java programmers are more employable than Visual Basic (VB) programmers by a whopping 190 percent.

- **2007: Google adopts Java as the primary language for creating apps on Android mobile devices.**

- **January 2010: Oracle Corporation purchases Sun Microsystems, bringing Java technology into the Oracle family of products.**

- **June 2010: *eWeek* ranks Java first among its "Top 10 Programming Languages to Keep You Employed" (`www.eweek.com/c/a/Application-Development/Top-10-Programming-Languages-to-Keep-You-Employed-719257`).**

- **2016: Java runs on 15 billion devices (`http://java.com/en/about`), with Android Java running on 87.6 percent of all mobile phones worldwide (`www.idc.com/prodserv/smartphone-os-market-share.jsp`).**

  Additionally, Java technology provides interactive capabilities to all Blu-ray devices and is the most popular programming language in the TIOBE Programming Community Index (`www.tiobe.com/index.php/content/paperinfo/tpci`), on PYPL: the PopularitY of Programming Language Index (`http://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language`), and on other indexes.

  Well, I'm impressed.

Object-Oriented Programming (OOP)

It's three in the morning. I'm dreaming about the history course that I failed in high school. The teacher is yelling at me, "You have two days to study for the final exam, but you won't remember to study. You'll forget and feel guilty, guilty, guilty."

Suddenly, the phone rings. I'm awakened abruptly from my deep sleep. (Sure, I disliked dreaming about the history course, but I like being awakened even less.) At first, I drop the telephone on the floor. After fumbling to pick it up, I issue a grumpy, "Hello, who's this?" A voice answers, "I'm a reporter from the *New York Times*. I'm writing an article about Java, and I need to know all about the programming language in five words or less. Can you explain it?"

My mind is too hazy. I can't think. So I say the first thing that comes to my mind and then go back to sleep.

Come morning, I hardly remember the conversation with the reporter. In fact, I don't remember how I answered the question. Did I tell the reporter where he could put his article about Java?

I put on my robe and rush out to my driveway. As I pick up the morning paper, I glance at the front page and see this 2-inch headline:

Burd Calls Java "A Great Object-Oriented Language"

Object-oriented languages

Java is object-oriented. What does that mean? Unlike languages, such as FORTRAN, that focus on giving the computer imperative "Do this/Do that" commands, object-oriented languages focus on data. Of course, object-oriented programs still tell the computer what to do. They start, however, by organizing the data, and the commands come later.

Object-oriented languages are better than "Do this/Do that" languages because they organize data in a way that helps people do all kinds of things with it. To modify the data, you can build on what you already have rather than scrap everything you've done and start over each time you need to do something new. Although computer programmers are generally smart people, they took a while to figure this out. For the full history lesson, see the sidebar "The winding road from FORTRAN to Java" (but I won't make you feel guilty if you don't read it).

THE WINDING ROAD FROM FORTRAN TO JAVA

In the mid-1950s, a team of people created a programming language named FORTRAN. It was a good language, but it was based on the idea that you should issue direct, imperative commands to the computer. "Do this, computer. Then do that, computer." (Of course, the commands in a real FORTRAN program were much more precise than "Do this" or "Do that.")

In the years that followed, teams developed many new computer languages, and many of the languages copied the FORTRAN "Do this/Do that" model. One of the more popular "Do this/Do that" languages went by the 1-letter name *C*. Of course, the "Do this/Do that" camp had some renegades. In languages named SIMULA and Smalltalk, programmers moved the imperative "Do this" commands into the background and concentrated on descriptions of data. In these languages, you didn't come right out and say, "Print a list of delinquent accounts." Instead, you began by saying, "This is what it means to be an account. An account has a name and a balance." Then you said, "This is how you ask an account whether it's delinquent." Suddenly, the data became king. An account was a thing that had a name, a balance, and a way of telling you whether it was delinquent.

Languages that focus first on the data are called *object-oriented* programming languages. These object-oriented languages make excellent programming tools. Here's why:

- Thinking first about the data makes you a good computer programmer.
- You can extend and reuse the descriptions of data over and over again. When you try to teach old FORTRAN programs new tricks, however, the old programs show how brittle they are. They break.

In the 1970s, object-oriented languages, such as SIMULA and Smalltalk, were buried in the computer hobbyist magazine articles. In the meantime, languages based on the old FORTRAN model were multiplying like rabbits.

So in 1986, a fellow named Bjarne Stroustrup created a language named C++. The C++ language became very popular because it mixed the old C language terminology with the improved object-oriented structure. Many companies turned their backs on the old FORTRAN/C programming style and adopted C++ as their standard.

(continued)

(continued)

But C++ had a flaw. Using C++, you could bypass all the object-oriented features and write a program by using the old FORTRAN/C programming style. When you started writing a C++ accounting program, you could take either fork in the road:

- Start by issuing direct "Do this" commands to the computer, saying the mathematical equivalent of "Print a list of delinquent accounts, and make it snappy."
- Choose the object-oriented approach and begin by describing what it means to be an account.

Some people said that C++ offered the best of both worlds, but others argued that the first world (the world of FORTRAN and C) shouldn't be part of modern programming. If you gave a programmer an opportunity to write code either way, the programmer would too often choose to write code the wrong way.

So in 1995, James Gosling of Sun Microsystems created the language named *Java.* In creating Java, Gosling borrowed the look and feel of C++. But Gosling took most of the old "Do this/Do that" features of C++ and threw them in the trash. Then he added features that made the development of objects smoother and easier. All in all, Gosling created a language whose object-oriented philosophy is pure and clean. When you program in Java, you have no choice but to work with objects. That's the way it should be.

Objects and their classes

In an object-oriented language, you use objects *and* classes to organize your data.

Imagine that you're writing a computer program to keep track of the houses in a new condominium development (still under construction). The houses differ only slightly from one another. Each house has a distinctive siding color, an indoor paint color, a kitchen cabinet style, and so on. In your object-oriented computer program, each house is an object.

But objects aren't the whole story. Although the houses differ slightly from one another, all the houses share the same list of characteristics. For instance, each house has a characteristic known as *siding color.* Each house has another characteristic known as *kitchen cabinet style.* In your object-oriented program, you need a master list containing all the characteristics that a house object can possess. This master list of characteristics is called a *class.*

So there you have it. Object-oriented programming is misnamed. It should be called "programming with classes and objects."

Now notice that I put the word *classes* first. How dare I do this! Well, maybe I'm not so crazy. Think again about a housing development that's under construction. Somewhere on the lot, in a rickety trailer parked on bare dirt, is a master list of characteristics known as a blueprint. An architect's blueprint is like an object-oriented programmer's class. A blueprint is a list of characteristics that each house will have. The blueprint says, "siding." The actual house object has gray siding. The blueprint says, "kitchen cabinet." The actual house object has Louis XIV kitchen cabinets.

The analogy doesn't end with lists of characteristics. Another important parallel exists between blueprints and classes. A year after you create the blueprint, you use it to build ten houses. It's the same with classes and objects. First, the programmer writes code to describe a class. Then when the program runs, the computer creates objects from the (blueprint) class.

So that's the real relationship between classes and objects. The programmer defines a class, and from the class definition, the computer makes individual objects.

What's so good about an object-oriented language?

Based on the preceding section's story about home building, imagine that you've already written a computer program to keep track of the building instructions for houses in a new development. Then, the big boss decides on a modified plan — a plan in which half the houses have three bedrooms and the other half have four.

If you use the old FORTRAN/C style of computer programming, your instructions look like this:

```
Dig a ditch for the basement.
Lay concrete around the sides of the ditch.
Put two-by-fours along the sides for the basement's frame.
...
```

This would be like an architect creating a long list of instructions instead of a blueprint. To modify the plan, you have to sort through the list to find the instructions for building bedrooms. To make things worse, the instructions could be scattered among pages 234, 394–410, 739, 10, and 2. If the builder had to decipher other peoples' complicated instructions, the task would be ten times harder.

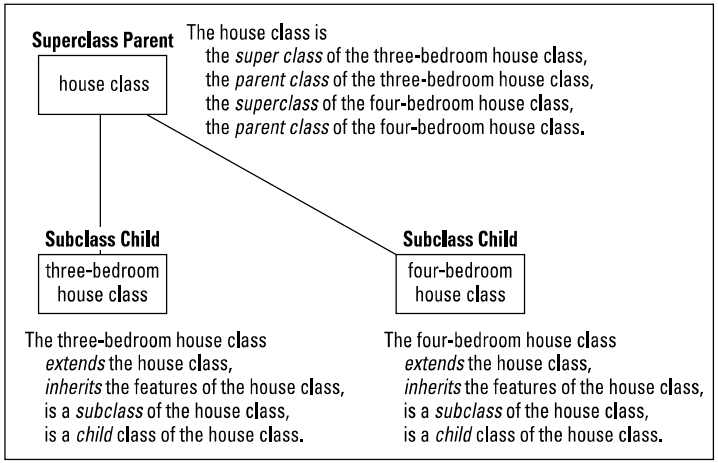

FIGURE 1-1: Terminology in object-oriented programming.

In a confident tone, you reply, "We don't have to mess with the original house blueprint. If someone wants a Jacuzzi in his living room, we can make a new, small blueprint describing only the new living room and call this the *Jacuzzi-in-living-room house* blueprint. Then, this new blueprint can refer to the original house blueprint for info on the rest of the house (the part that's not in the living room)." In the language of object-oriented programming, the Jacuzzi-in-living-room house blueprint still *extends* the original house blueprint. The Jacuzzi blueprint is still a subclass of the original house blueprint. In fact, all the terminology about superclass, parent class, and child class still applies. The only thing that's new is that the Jacuzzi blueprint *overrides* the living room features in the original house blueprint.

In the days before object-oriented languages, the programming world experienced a crisis in software development. Programmers wrote code, and then discovered new needs, and then had to trash their code and start from scratch. This problem happened over and over again because the code that the programmers were writing couldn't be reused. Object-oriented programming changed all this for the better (and, as Burd said, Java is "A Great Object-Oriented Language").

Refining your understanding of classes and objects

When you program in Java, you work constantly with classes and objects. These two ideas are really important. That's why, in this chapter, I hit you over the head with one analogy after another about classes and objects.

Starting with a class, however, is like starting with a blueprint. If you decide to have both three- and four-bedroom houses, you can start with a blueprint called the house blueprint that has a ground floor and a second floor, but has no indoor walls drawn on the second floor. Then you make two more second-floor blueprints — one for the three-bedroom house and another for the four-bedroom house. (You name these new blueprints the *three-bedroom house* blueprint and the *four-bedroom house* blueprint.)

Your builder colleagues are amazed with your sense of logic and organization, but they have concerns. They pose a question. "You called one of the blueprints the 'three-bedroom house' blueprint. How can you do this if it's a blueprint for a second floor and not for a whole house?"

You smile knowingly and answer, "The three-bedroom house blueprint can say, 'For info about the lower floors, see the original house blueprint.' That way, the three-bedroom house blueprint describes a whole house. The four-bedroom house blueprint can say the same thing. With this setup, we can take advantage of all the work we already did to create the original house blueprint and save lots of money."

In the language of object-oriented programming, the three- and four-bedroom house classes are *inheriting* the features of the original house class. You can also say that the three- and four-bedroom house classes are *extending* the original house class. (See Figure 1-1.)

The original house class is called the *superclass* of the three- and four-bedroom house classes. In that vein, the three- and four-bedroom house classes are *subclasses* of the original house class. Put another way, the original house class is called the *parent class* of three- and four-bedroom house classes. The three- and four-bedroom house classes are *child classes* of the original house class. (Refer to Figure 1-1.)

Needless to say, your homebuilder colleagues are jealous. A crowd of homebuilders is mobbing around you to hear about your great ideas. So, at that moment, you drop one more bombshell: "By creating a class with subclasses, we can reuse the blueprint in the future. If someone comes along and wants a five-bedroom house, we can extend our original house blueprint by making a five-bedroom house blueprint. We'll never have to spend money for an original house blueprint again."

"But," says a colleague in the back row, "what happens if someone wants a different first-floor design? Do we trash the original house blueprint or start scribbling all over the original blueprint? That'll cost big bucks, won't it?"

Close your eyes for a minute and think about what it means for something to be a chair . . . .

A chair has a seat, a back, and legs. Each seat has a shape, a color, a degree of softness, and so on. These are the properties that a chair possesses. What I describe is *chairness* — the notion of something being a chair. In object-oriented terminology, I'm describing the `Chair` class.

Now peek over the edge of this book's margin and take a minute to look around your room. (If you're not sitting in a room right now, fake it.)

Several chairs are in the room, and each chair is an object. Each of these objects is an example of that ethereal thing called the `Chair` class. So that's how it works — the class is the idea of *chairness*, and each individual chair is an object.

REMEMBER

A class isn't quite a collection of things. Instead, a class is the idea behind a certain kind of thing. When I talk about the class of chairs in your room, I'm talking about the fact that each chair has legs, a seat, a color, and so on. The colors may be different for different chairs in the room, but that doesn't matter. When you talk about a class of things, you're focusing on the properties that each of the things possesses.

It makes sense to think of an object as being a concrete instance of a class. In fact, the official terminology is consistent with this thinking. If you write a Java program in which you define a `Chair` class, each actual chair (the chair that you're sitting on, the empty chair right next to you, and so on) is called an *instance* of the `Chair` class.

Here's another way to think about a class. Imagine a table displaying all three of your bank accounts. (See Table 1-1.)

TABLE 1-1 A Table of Accounts

Account Number	Type	Balance
16-13154-22864-7	Checking	174.87
1011 1234 2122 0000	Credit	–471.03
16-17238-13344-7	Savings	247.38

Think of the table's column headings as a class, and think of each row of the table as an object. The table's column headings describe the `Account` class.

According to the table's column headings, each account has an account number, a type, and a balance. Rephrased in the terminology of object-oriented programming, each object in the `Account` class (that is, each instance of the `Account` class) has an account number, a type, and a balance. So, the bottom row of the table is an object with account number *16-17238-13344-7*. This same object has type *Savings* and a balance of *247.38*. If you opened a new account, you would have another object, and the table would grow an additional row. The new object would be an instance of the same `Account` class.

What's Next?

This chapter is filled with general descriptions of things. A general description is good when you're just getting started, but you don't really understand things until you get to know some specifics. That's why the next several chapters deal with specifics.

So please, turn the page. The next chapter can't wait for you to read it.

IN THIS CHAPTER

- Understanding the roles of the software development tools
- Selecting the version of Java that's right for you
- Preparing to write and run Java programs

Chapter 2
All about Software

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 chapter gets you ready to do Java by describing the *general* software setup — the software that you must have on your computer whether you run Windows, Mac, Linux, or Joe's Private Operating System. This chapter *doesn't* describe the specific setup instructions for Windows, for a Mac, or for any other system.

For setup instructions that are specific to your system, visit this book's website (`www.allmycode.com/JavaForDummies`).

Quick-Start Instructions

If you're a seasoned veteran of computers and computing (whatever that means), and if you're too jumpy to get detailed instructions from this book's website, you can try installing the required software by following this section's general instructions. The instructions work for many computers, but not all. And this section provides no detailed steps, no if-this-then-do-that alternatives, and no this-works-but-you're-better-off-doing-something-else tips.

To prepare your computer for writing Java programs, follow these steps:

1. **Install the Java Development Kit.**

   To do so, visit `www.oracle.com/technetwork/java/javase/downloads`.

   Follow the instructions at that website to download and install the newest Java SE JDK.


REMEMBER

   Look for the Standard Edition (SE). Don't bother with the Enterprise Edition (EE) or any other such edition. Also, go for the JDK, not the JRE. If you see a code number, such as 9u3, this stands for "the 3rd update of Java 9." Generally, anything marked Java 9 or later is good for running the examples in this book.

2. **Install an integrated development environment.**

   An *integrated development environment* (IDE) is a program to help you compose and test new software. For this book's examples, you can use almost any IDE that supports Java.

   Here's a list of the most popular Java IDEs:

   - *Eclipse*

     According to `www.baeldung.com/java-ides-2016`, 48.2 percent of the world's Java programmers used the Eclipse IDE in mid-2016.

     To download and use Eclipse, follow the instructions at `http://eclipse.org/downloads`. Eclipse's download page may offer you several different packages, including Eclipse for Java EE, Eclipse for JavaScript, Eclipse for Java and DSL, and others. To run this book's examples, you need a relatively small Eclipse package — the Eclipse IDE for Java Developers.

     Eclipse is free for commercial and noncommercial use.

   - *IntelliJ IDEA*

     In Baeldung's survey of Java IDEs (`http://www.baeldung.com/java-ides-2016`), IntelliJ IDEA comes in a close second, with 43.6 percent of all programmers onboard.

     When you visit `www.jetbrains.com/idea`, you can download the Community Edition (which is free) or the Ultimate Edition (which isn't free). To run this book's examples, you can use the Community Edition. You can even use the Community Edition to create commercial software!

   - *NetBeans*

     Baeldung's survey of Java IDEs (`http://www.baeldung.com/java-ides-2016`) gives NetBeans a mere 5.9 percent. But NetBeans is Oracle's official Java IDE. If the site offers you a choice of download bundles, choose the Java SE bundle.

     To get your own copy of NetBeans, visit `https://netbeans.org/downloads`.

     NetBeans is free for commercial and noncommercial use.

3. **Test your installed software.**

   What you do in this step depends on which IDE you choose in Step 2. Anyway, here are some general instructions:

   a. Launch your IDE (Eclipse, IntelliJ IDEA, NetBeans, or whatever).

   b. In the IDE, create a new Java project.

   c. Within the Java project, create a new Java class named `Displayer`. (Selecting File ➪ New ➪ Class works in most IDEs.)

   d. Edit the new `Displayer.java` file by typing the code from Listing 3-1 (the first code listing in Chapter 3).

      For most IDEs, you add the code into a big (mostly blank) editor pane. Try to type the code exactly as you see it in Listing 3-1. If you see an uppercase letter, type an uppercase letter. Do the same with all lowercase letters.

      What? You say you don't want to type a bunch of code from the book? Well, all right then! Visit this book's website (`www.allmycode.com/JavaForDummies`) to find out how to download all the code examples and load them into the IDE of your choice.

   e. Run `Displayer.java` and check to make sure that the run's output reads `You'll love Java!`.

That's it! But remember: Not everyone (computer geek or not) can follow these skeletal instructions flawlessly. So you have several alternatives:

» **Visit this book's website.**

  Do not pass Go. Do not try this section's quick-start instructions. Follow the more detailed instructions that you find at `www.allmycode.com/JavaForDummies`.

- **Try this section's quick-start instructions.**

  You can't hurt anything by trying. If you accidentally install the wrong software, you can probably leave the wrong software on your computer. (You don't have to uninstall it.) If you're not sure whether you've installed the software correctly, you can always fall back on my website's detailed instructions.

- **E-mail your questions to me at** `JavaForDummies@allmycode.com`**.**
- **Tweet me at** `@allmycode`**.**
- **Visit my** `/allmycode` **Facebook page.**

  I like hearing from readers.

What You Install on Your Computer

I once met a tool-and-die maker. He used tools to make tools (and dies). I was happy to meet him because I knew that, one day, I'd make an analogy between computer programmers and tool-and-die makers.

A computer programmer uses existing programs as tools to create new programs. The existing programs and new programs might perform very different kinds of tasks. For example, a Java program (a program that you create) might keep track of a business's customers. To create that customer-tracking program, you might use an existing program that looks for errors in your Java code. This general-purpose error-finding program can find errors in any kind of Java code — customer-tracking code, weather-predicting code, gaming code, or the code for an app on your mobile phone.

So how many tools do you need for creating Java programs? As a novice, you need three tools:

- **You need a compiler.**

  A *compiler* takes the Java code that you write and turns that code into a bunch of instructions called *bytecode*.

  Humans can't readily compose or decipher bytecode instructions. But certain software that you run on your computer can interpret and carry out bytecode instructions.

- **You need a Java Virtual Machine (JVM).**

  A *Java Virtual Machine* is a piece of software. A Java Virtual Machine interprets and carries out bytecode instructions.

- **You need an integrated development environment (IDE).**

  An *integrated development environment* helps you manage your Java code and provides convenient ways for you to write, compile, and run your code.

  To be honest, you don't actually *need* an integrated development environment. In fact, some programmers take pride in using plain, old text editors such as Windows Notepad, Macintosh TextEdit, or the vim editor in Linux. But, as a novice programmer, a full-featured IDE makes your life much, much easier.

The World Wide Web has free, downloadable versions of each of these tools:

- When you download the Java SE JDK from Oracle's website (`www.oracle.com/technetwork/java/javase/downloads/index.html`), you get the compiler and the JVM.
- When you visit the Eclipse (`www.eclipse.org/downloads`), IntelliJ IDEA (`www.jetbrains.com/idea`, or NetBeans (`https://netbeans.org/downloads`) site, you get an IDE.

You may find variations on the picture that I paint in the preceding two bullets. Many IDEs come with their own JVMs, and Oracle's website may offer a combined JDK+NetBeans bundle. Nevertheless, the picture that I paint with these bullets is useful and reliable. When you follow my instructions, you might end up with two copies of the JVM, or two IDEs, but that's okay. You never know when you'll need a spare.

This chapter provides background information about software you need on your computer. But the chapter contains absolutely no detailed instructions to help you install the software. For detailed instructions, visit this book's website (`www.allmycode.com/JavaForDummies`).

The rest of this chapter describes compilers, JVMs, and IDEs.

What is a compiler?

A compiler takes the Java code that you write and turns that code into a bunch of instructions called bytecode.

—*BARRY BURD*, JAVA FOR DUMMIES, *7TH EDITION*

You're a human being. (Sure, every rule has exceptions. But if you're reading this book, you're probably human.) Anyway, humans can write and comprehend the code in Listing 2-1.

LISTING 2-1: Looking for a Vacant Room

```
// This is part of a Java program.
// It's not a complete Java program.
roomNum = 1;
while (roomNum < 100) {
    if (guests[roomNum] == 0) {
        out.println("Room " + roomNum + " is available.");
        exit(0);
    } else {
        roomNum++;
    }
}
out.println("No vacancy");
```

The Java code in Listing 2-1 checks for vacancies in a small hotel (a hotel with room numbers 1 to 99). You can't run the code in Listing 2-1 without adding several additional lines. But here in Chapter 2, those additional lines aren't important. What's important is that, by staring at the code, squinting a bit, and looking past all the code's strange punctuation, you can see what the code is trying to do:

```
Set the room number to 1.
As long as the room number is less than 100,
    Check the number of guests in the room.
    If the number of guests in the room is 0, then
        report that the room is available,
        and stop.
    Otherwise,
        prepare to check the next room by
        adding 1 to the room number.
If you get to the nonexistent room number 100, then
    report that there are no vacancies.
```

If you don't see the similarities between Listing 2-1 and its English equivalent, don't worry. You're reading *Java For Dummies*, 7th Edition, and like most human beings, you can learn to read and write the code in Listing 2-1. The code in Listing 2-1 is called *Java source code.*

So here's the catch: Computers aren't human beings. Computers don't normally follow instructions like the instructions in Listing 2-1. That is, computers don't follow Java source code instructions. Instead, computers follow cryptic instructions like the ones in Listing 2-2.

LISTING 2-2: Listing 2-1 Translated into Java Bytecode

```
aload_0
iconst_1
putfield Hotel/roomNum I
goto 32
aload_0
getfield Hotel/guests [I
aload_0
getfield Hotel/roomNum I
iaload
ifne 26
getstatic java/lang/System/out Ljava/io/PrintStream;
new java/lang/StringBuilder
dup
ldc "Room "
invokespecial java/lang/StringBuilder/<init>(Ljava/lang/String;)V
aload_0
getfield Hotel/roomNum I
invokevirtual java/lang/StringBuilder/append(I)Ljava/lang/StringBuilder;
ldc " is available."
invokevirtual
  java/lang/StringBuilder/append(Ljava/lang/String;)Ljava/lang/StringBuilder;
invokevirtual java/lang/StringBuilder/toString()Ljava/lang/String;
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
iconst_0
invokestatic java/lang/System/exit(I)V
goto 32
aload_0
dup
getfield Hotel/roomNum I
iconst_1
iadd
putfield Hotel/roomNum I
aload_0
getfield Hotel/roomNum I
```

(continued)

wording is " . . . computers follow cryptic instructions *like* the ones in Listing 2-2." The instructions in Listing 2-2 are a lot like instructions that a computer can execute, but generally, computers don't execute Java bytecode instructions. Instead, each kind of computer processor has its own set of executable instructions, and each computer operating system uses the processor's instructions in a slightly different way.

Here's a hypothetical situation: The year is 1992 (a few years before Java was made public) and you run the Linux operating system on a computer that has an old Pentium processor. Your friend runs Linux on a computer with a different kind of processor — a PowerPC processor. (In the 1990s, Intel Corporation made Pentium processors, and IBM made PowerPC processors.)

Listing 2-3 contains a set of instructions to display `Hello world!` on the computer screen.* The instructions work on a Pentium processor running the Linux operating system.

LISTING 2-3: A Simple Program for a Pentium Processor

```
.data
msg:
        .ascii  "Hello, world!\n"
        len = . - msg
.text
    .global _start
_start:
        movl    $len,%edx
        movl    $msg,%ecx
        movl    $1,%ebx
        movl    $4,%eax
        int     $0x80

        movl    $0,%ebx
        movl    $1,%eax
        int     $0x80
```

*I paraphrase these Intel instructions from Konstantin Boldyshev's Linux Assembly HOWTO (`http://tldp.org/HOWTO/Assembly-HOWTO/hello.html`).

LISTING 2-2: *(continued)*

```
bipush 100
if_icmplt 5
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "No vacancy"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
return
```

The instructions in Listing 2-2 aren't Java source code instructions. They're *Java bytecode* instructions. When you write a Java program, you write source code instructions (like the instructions in Listing 2-1). After writing the source code, you run a program (that is, you apply a tool) to your source code. The program is a *compiler*. The compiler translates your source code instructions into Java bytecode instructions. In other words, the compiler takes code that you can write and understand (like the code in Listing 2-1) and translates it into code that a computer has a fighting chance of carrying out (like the code in Listing 2-2).


TECHNICAL STUFF

You might put your source code in a file named `Hotel.java`. If so, the compiler probably puts the Java bytecode in another file named `Hotel.class`. Normally, you don't bother looking at the bytecode in the `Hotel.class` file. In fact, the compiler doesn't encode the `Hotel.class` file as ordinary text, so you can't examine the bytecode with an ordinary editor. If you try to open `Hotel.class` with Notepad, TextEdit, KWrite, or even Microsoft Word, you'll see nothing but dots, squiggles, and other gobbledygook. To create Listing 2-2, I had to apply yet another tool to my `Hotel.class` file. That tool displays a text-like version of a Java bytecode file. I used Ando Saabas's Java Bytecode Editor (`www.cs.ioc.ee/~ando/jbe`).


REMEMBER

No one (except for a few crazy programmers in some isolated labs in faraway places) writes Java bytecode. You run software (a compiler) to create Java bytecode. The only reason to look at Listing 2-2 is to understand what a hard worker your computer is.

What is a Java Virtual Machine?

A Java Virtual Machine is a piece of software. A Java Virtual Machine interprets and carries out bytecode instructions.

—*BARRY BURD,* JAVA FOR DUMMIES, *7TH EDITION*

In the preceding "What is a compiler?" section, I make a big fuss about computers following instructions like the ones in Listing 2-2. As fusses go, it's a very nice fuss. But if you don't read every fussy word, you may be misguided. The exact

Listing 2-4 contains another set of instructions to display `Hello world!` on the screen.** The instructions in Listing 2-4 work on a PowerPC processor running Linux.

LISTING 2-4: A Simple Program for a PowerPC Processor

```
.data
msg:
        .string "Hello, world!\n"
        len = . - msg
.text
        .global _start
_start:
        li      0,4
        li      3,1
        lis     4,msg@ha
        addi    4,4,msg@l
        li      5,len
        sc

        li      0,1
        li      3,1
        sc
```

The instructions in Listing 2-3 run smoothly on a Pentium processor. But these instructions mean nothing to a PowerPC processor. Likewise, the instructions in Listing 2-4 run nicely on a PowerPC, but these same instructions are complete gibberish to a computer with a Pentium processor. So your friend's PowerPC software might not be available on your computer. And your Intel computer's software might not run at all on your friend's computer.

Now go to your cousin's house. Your cousin's computer has a Pentium processor (just like yours), but your cousin's computer runs Windows instead of Linux. What does your cousin's computer do when you feed it the Pentium code in Listing 2-3? It screams, "Not a valid Win32 application" or "Windows can't open this file." What a mess!

**I paraphrase the PowerPC code from Hollis Blanchard's PowerPC Assembly page (`www.ibm.com/developerworks/library/l-ppc`). Hollis also reviewed and critiqued this "What is a Java Virtual Machine?" section for me. Thank you, Hollis.

Java bytecode creates order from all this chaos. Unlike the code in Listings 2-3 and 2-4, Java bytecode isn't specific to one kind of processor or to one operating system. Instead, any kind of computer can have a Java Virtual Machine, and Java bytecode instructions run on any computer's Java Virtual Machine. The JVM that runs on a Pentium with Linux translates Java bytecode instructions into the kind of code you see in Listing 2-3. And the JVM that runs on a PowerPC with Linux translates Java bytecode instructions into the kind of code you see in Listing 2-4.

If you write a Java program and compile that Java program into bytecode, then the JVM on your computer can run the bytecode, the JVM on your friend's computer can run the bytecode, the JVM on your grandmother's supercomputer can run the bytecode, and with any luck, the JVM on your cellphone or tablet can run the bytecode.


CROSS REFERENCE

For a look at some Java bytecode, see Listing 2-2. But remember: You never have to write or decipher Java bytecode. Writing bytecode is the compiler's job. Deciphering bytecode is the Java Virtual Machine's job.

With Java, you can take a bytecode file that you created with a Windows computer, copy the bytecode to who-knows-what kind of computer, and then run the bytecode with no trouble at all. That's one of the many reasons why Java has become popular so quickly. This outstanding feature, which gives you the ability to run code on many different kinds of computers, is called *portability.*

What makes Java bytecode so versatile? This fantastic universality enjoyed by Java bytecode programs comes from the Java Virtual Machine. The Java Virtual Machine is one of those three tools that you must have on your computer.

Imagine that you're the Windows representative to the United Nations Security Council. (See Figure 2-1.) The Macintosh representative is seated to your right, and the Linux representative is on your left. (Naturally, you don't get along with either of these people. You're always cordial to one another, but you're never sincere. What do you expect? It's politics!) The distinguished representative from Java is at the podium. The Java representative is speaking in bytecode, and neither you nor your fellow ambassadors (Mac and Linux) understand a word of Java bytecode.

But each of you has an interpreter. Your interpreter translates from bytecode to Windows while the Java representative speaks. Another interpreter translates from bytecode to Macintosh-ese. And a third interpreter translates bytecode into Linux-speak.

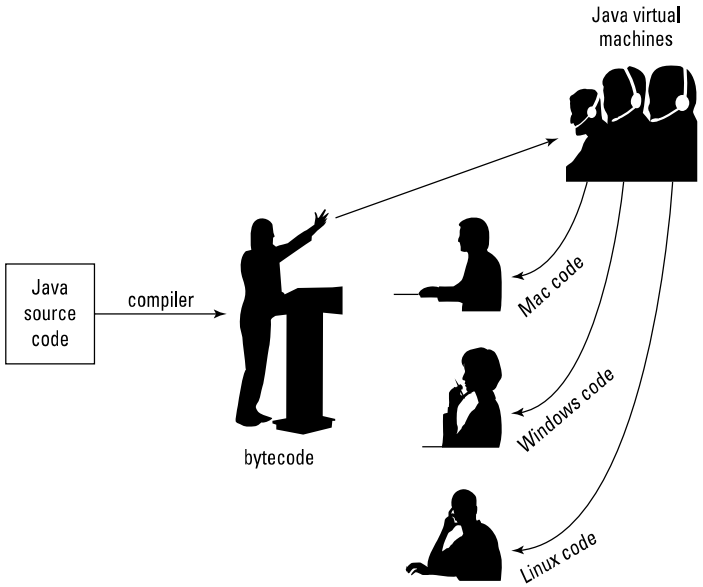

FIGURE 2-1: An imaginary meeting of the UN Security Council.

Think of your interpreter as a virtual ambassador. The interpreter doesn't really represent your country, but the interpreter performs one of the important tasks that a real ambassador performs. The interpreter listens to bytecode on your behalf. The interpreter does what you would do if your native language were Java bytecode. The interpreter pretends to be the Windows ambassador and sits through the boring bytecode speech, taking in every word and processing each word in some way or another.

You have an interpreter — a virtual ambassador. In the same way, a Windows computer runs its own bytecode-interpreting software. That software is the Java Virtual Machine.

A Java Virtual Machine is a proxy, an errand boy, a go-between. The JVM serves as an interpreter between Java's run-anywhere bytecode and your computer's own system. While it runs, the JVM walks your computer through the execution of bytecode instructions. The JVM examines your bytecode, bit by bit, and carries out the instructions described in the bytecode. The JVM interprets bytecode for your Windows system, your Mac, or your Linux box, or for whatever kind of computer you're using. That's a good thing. It's what makes Java programs more portable than programs in any other language.

WHAT ON EARTH IS JAVA 2 STANDARD EDITION 1.2?

If you poke around the web looking for Java tools, you find things with all kinds of strange names. You find the Java Development Kit, the Software Development Kit, the Java Runtime Environment, and other confusing names.

- The names *Java Development Kit* (JDK) and *Software Development Kit* (SDK) stand for different versions of the same toolset — a toolset whose key component is a Java compiler.
- The name *Java Runtime Environment* (JRE) stands for a toolset whose key component is a Java Virtual Machine.

  If you install the JDK on your computer, the JRE comes along with it. You can also get the JRE on its own. In fact, you can have many combinations of the JDK and JRE on your computer. For example, my Windows computer currently has JDK 1.6, JDK 1.8, and JRE 8 in its `c:\program files\Java directory` and has JDK 9 in its `c:\program files (x86)\Java directory`. Only occasionally do I run into any version conflicts. If you suspect that you're experiencing a version conflict, it's best to uninstall all JDK and JRE versions except the latest (for example, JDK 9 and JRE 9).

The numbering of Java versions can be confusing. Instead of "Java 1," "Java 2," and "Java 3," the numbering of Java versions winds through an obstacle course. This sidebar's figure describes the development of new Java versions over time. Each Java version has several names. The *product version* is an official name that's used for the world in general, and the *developer version* is a number that identifies versions so that programmers can keep track of them. (In casual conversation, programmers use all kinds of names for the various Java versions.) The *code name* is a more playful name that identifies a version while it's being created.

The asterisks in the figure mark changes in the formulation of Java product-version names. Back in 1996, the product versions were *Java Development Kit 1.0* and *Java Development Kit 1.1*. In 1998, someone decided to christen the product *Java 2 Standard Edition 1.2*, which confuses everyone to this day. At the time, anyone using the term *Java Development Kit* was asked to use *Software Development Kit* (SDK) instead.

(continued)

Developing software

All this has happened before, and it will all happen again.
—PETER PAN *(J.M. BARRIE) AND* BATTLESTAR GALACTICA
(2003–2009, NBC UNIVERSAL)

When you create a Java program, you repeat the same steps over and over again. Figure 2-2 illustrates the cycle.

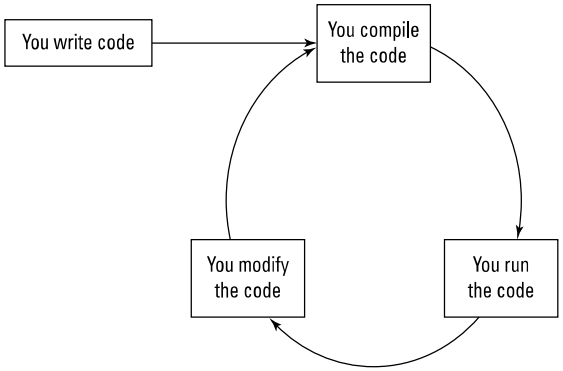

FIGURE 2-2: Developing a Java program.

First, you write a program. After writing the first draft, you repeatedly compile, run, and modify the program. With a little experience, the compile and run steps become very easy. In many cases, one mouse click starts the compilation or the run.

However, writing the first draft and modifying the code are not 1-click tasks. Developing code requires time and concentration.

REMEMBER

Never be discouraged when the first draft of your code doesn't work. For that matter, never be discouraged when the 25th draft of your code doesn't work. Rewriting code is one of the most important things you can do (aside from ensuring world peace).

For detailed instructions on compiling and running Java programs, visit this book's website (`www.allmycode.com/JavaForDummies`).

When people talk about writing programs, they use the wording in Figure 2-2. They say, "You compile the code" and "You run the code." But the "you" isn't always accurate, and the "code" differs slightly from one part of the cycle to the next. Figure 2-3 describes the cycle from Figure 2-2 in a bit more detail.

(continued)

In 2004 the *1.* business went away from the platform version name, and in 2006 Java platform names lost the *2* and the *.0*.

By far the most significant changes for Java programmers came about in 2004. With the release of J2SE 5.0, the overseers of Java made changes to the language by adding new features — features such as generic types, annotations, and the enhanced `for` statement. (To see Java annotations in action, go to Chapters 8, 9, and 16. For examples of the use of the enhanced `for` statement and generic types, see Chapters 11 and 12.)

Most of the programs in this book run only with Java 5.0 or later. They don't run with any version earlier than Java 5.0. Particularly, they don't run with Java 1.4 or Java 1.4.2. Some of this book's examples don't run with Java 9 or lower. But don't worry too much about Java version numbers. Java 6 or 7 is better than no Java at all. You can learn a lot about Java without having the latest Java version.

	Platform	Codename	Features
1995	(Beta)		
1996	**JDK* 1.0**		
1997	**JDK 1.1**		Inner classes, Java Beans, Reflection
1998	**J2SE* 1.2**	Playground	Swing classes for creation of GUI interfaces
1999			
2000	**J2SE 1.3**	Kestrel	Java Naming and Directory interface (JNDI)
2001			
2002	**J2SE 1.4**	Merlin	New I/O, regular expressions, XML parsing
2003			
2004	**J2SE 5.0**	Tiger	Generic types, annotations, enum types, varargs, enhanced for
2005			statement, static imports, new concurrency classes
2006	**Java SE* 6**	Mustang	Scripting language support, performance enhancements
2007			
2008			
2009			
2010			
2011	**Java SE 7**	Dolphin	Strings in switch statement, catching multiple exceptions
2012			try statement with resources, integration with JavaFX
2013			
2014	**Java SE 8**		Lambda expressions
2015			
2016			
2017	**Java SE 9**		Modularity with Project Jigsaw, interactive coding with JShell

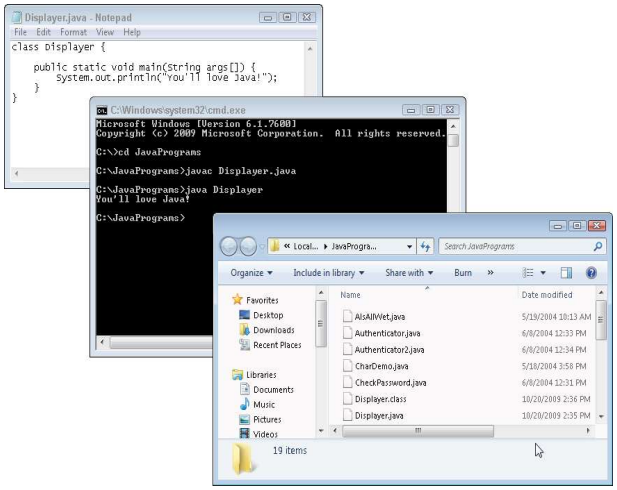

FIGURE 2-4: Developing code without an integrated development environment.

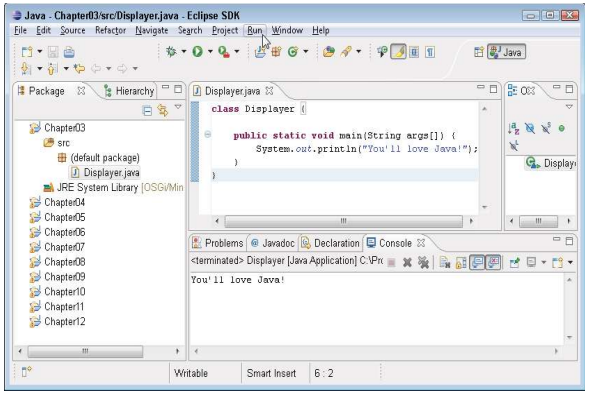

FIGURE 2-5: Developing code with the Eclipse integrated development environment.

Java has its share of integrated development environments such as Eclipse, IntelliJ IDEA, and NetBeans. Many environments have drag-and-drop components so that you can design your graphical interface visually. (See Figure 2-6.)

To run a program, you might click a toolbar button or choose Run from a menu. To compile a program, you might not have to do anything at all. (You might not even have to issue a command. Some IDEs compile your code automatically while you type it.)

For help with installing and using an integrated development environment, see this book's website (`www.allmycode.com/JavaForDummies`).

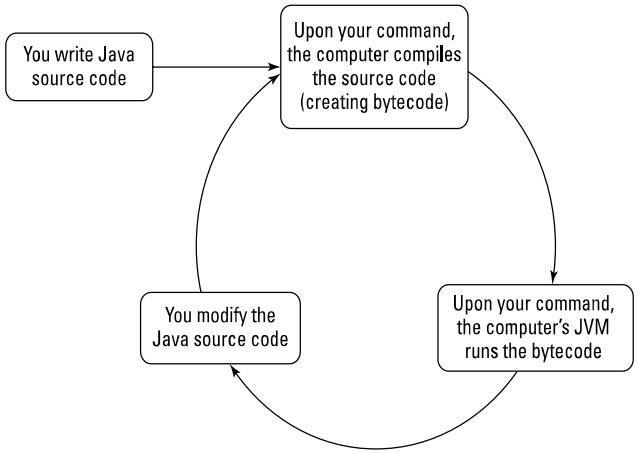

FIGURE 2-3: Who does what with which code?

TIP

For most people's needs, Figure 2-3 contains too much information. If I click a Run icon, I don't have to remember that the computer runs code on my behalf. And for all I care, the computer can run my original Java code or some bytecode knockoff of my original Java code. In fact, many times in this book, I casually write "when you run your Java code," or "when the computer runs your Java program." You can live a very happy life without looking at Figure 2-3. The only use for Figure 2-3 is to help you if the loose wording in Figure 2-2 confuses you. If Figure 2-2 doesn't confuse you, ignore Figure 2-3.

What is an integrated development environment?

"An integrated development environment helps you manage your Java code and provides convenient ways for you to write, compile, and run your code."
—*BARRY BURD,* JAVA FOR DUMMIES, *7TH EDITION*

In the olden days, writing and running a Java program involved opening several windows — a window for typing the program, another window for running the program, and maybe a third window to keep track of all the code you've written. (See Figure 2-4.)

An integrated development environment seamlessly combines all this functionality into one well-organized application. (See Figure 2-5.)

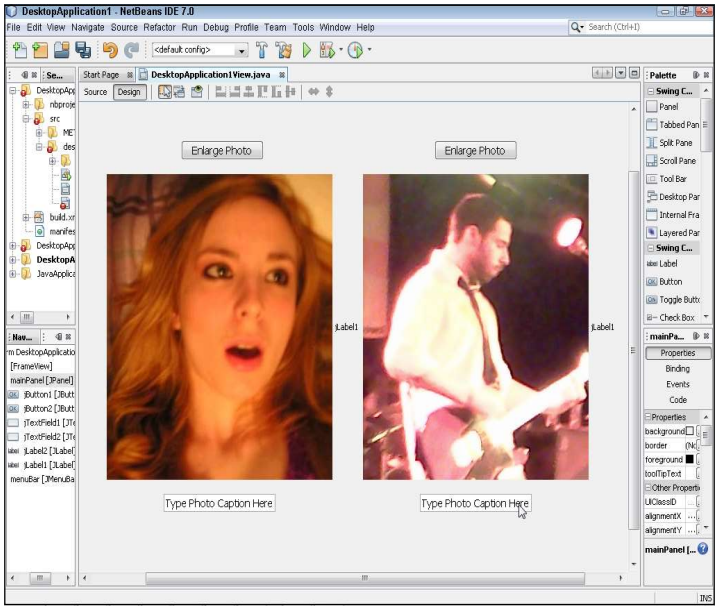

FIGURE 2-6: Using the drag-and-drop Swing GUI Builder in the NetBeans IDE.

IN THIS CHAPTER

- Speaking the Java language: The API and the language specification
- Taking a first glance at Java code
- Understanding the parts of a simple program
- Documenting your code

Chapter 3

Using the Basic Building Blocks

"Все мысли, которые имеют огромные последствия всегда просты. (All great ideas are simple.)"

—LEO TOLSTOY

The quotation applies to all kinds of things — things like life, love, and computer programming. That's why this chapter takes a multilayered approach. In this chapter, you get your first details about Java programming. And in discovering details, you'll see the simplicities.

Speaking the Java Language

If you try to picture in your mind the entire English language, what do you see? Maybe you see words, words, words. (That's what Hamlet saw.) Looking at the language under a microscope, you see one word after another. The bunch-of-words image is fine, but if you step back a bit, you may see two other things:

- The language's grammar
- Thousands of expressions, sayings, idioms, and historical names

The first category (the grammar) includes rules like, "The verb agrees with the noun in number and person." The second category (expressions, sayings, and stuff) includes knowledge like, "Julius Caesar was a famous Roman emperor, so don't name your son Julius Caesar, unless you want him to get beaten up every day after school."

The Java programming language has all the aspects of a spoken language like English. Java has words, grammar, commonly used names, stylistic idioms, and other such elements.

The grammar and the common names

The people at Sun Microsystems who created Java thought of Java as having two parts. Just as English has its grammar and commonly used names, the Java programming language has its specification (its grammar) and its application programming interface (its commonly used names). Whenever I write Java programs, I keep two important pieces of documentation — one for each part of the language — on my desk:

- **The Java Language Specification:** This documentation includes rules like this: "Always put an open parenthesis after the word *for*" and "Use an asterisk to multiply two numbers."
- **The application programming interface:** Java's application programming interface (API) contains thousands of names that were added to Java after the language's grammar was defined. These names range from the commonplace to the exotic. For example, one name — the name *JFrame* — represents a window on your computer's screen. A more razzle-dazzle name — *pow* — helps you raise 5 to the tenth power, or raise whatever to the whatever else power. Other names help you listen for the user's button clicks, query databases, and do all kinds of useful things.

You can download the language specification, the API documents, and all the other Java documentation (or view the documents online) by poking around at `http://docs.oracle.com/javase/specs`.

The first part of Java, the language specification, is relatively small. That doesn't mean you won't take plenty of time finding out how to use the rules in the language specification. Other programming languages, however, have double, triple, or ten times the number of rules.

The second part of Java — the API — can be intimidating because it's so large. The API contains thousands and thousands of names and keeps growing with each new Java language release. Pretty scary, eh? Well, the good news is that you don't

have to memorize anything in the API. Nothing. None of it. You can look up the stuff you need to use in the documentation and ignore the stuff you don't need. What you use often, you'll remember. What you don't use often, you'll forget (like any other programmer).

No one knows all there is to know about the Java API. If you're a Java programmer who frequently writes programs that open new windows, you know how to use the API `JFrame` class. If you seldom write programs that open windows, the first few times you need to create a window, you can look up the `JFrame` class in the API documentation. My guess is that if you prevented a typical Java programmer from looking up anything in the API documentation, the programmer would be able to use less than 2 percent of all the names in the Java API.

You may love the *For Dummies* style, but unfortunately, Java's official API documentation isn't written that way. The API documentation is both concise and precise. For some help deciphering the API documentation's language and style, see this book's website (`www.allmycode.com/JavaForDummies`).

In a way, nothing about the Java API is special. Whenever you write a Java program — even the smallest, simplest Java program — you create a class that's on par with any of the classes defined in the official Java API. The API is just a set of classes and other names that were created by ordinary programmers who happen to participate in the official Java Community Process (JCP) and in the OpenJDK Project. Unlike the names you create, the names in the API are distributed with every version of Java. (I'm assuming that you, the reader, are not a participant in the Java Community Process or the OpenJDK Project. But, with a fine book like *Java For Dummies*, 7th Edition, one never knows.)

If you're interested in the JCP's activities, visit `www.jcp.org`. If you're interested in the OpenJDK Project, visit `http://openjdk.java.net`.

The folks at the JCP don't keep the Java programs in the official Java API a secret. If you want, you can look at all these programs. When you install Java on your computer, the installation puts a file named `src.zip` on your hard drive. You can open `src.zip` with your favorite unzipping program. There, before your eyes, is all the Java API code.

The words in a Java program

A hard-core Javateer will say that the Java programming language has two kinds of words: keywords and identifiers. This is true. But the bare truth, without any other explanation, is sometimes misleading. So I recommend dressing up the truth a bit and thinking in terms of three kinds of words: keywords, identifiers that ordinary programmers like you and I create, and identifiers from the API.

The differences among these three kinds of words are similar to the differences among words in the English language. In the sentence "Sam is a person," the word *person* is like a Java keyword. No matter who uses the word *person*, the word always means roughly the same thing. (Sure, you can think of bizarre exceptions in English usage, but please don't.)

The word *Sam* is like a Java identifier because Sam is a name for a particular person. Words like *Sam*, *Dinswald*, and *McGillimaroo* aren't prepacked with meaning in the English language. These words apply to different people depending on the context and become names when parents pick one for their newborn kid.

Now consider the sentence "Julius Caesar is a person." If you utter this sentence, you're probably talking about the fellow who ruled Rome until the Ides of March. Although the name *Julius Caesar* isn't hard-wired into the English language, almost everyone uses the name to refer to the same person. If English were a programming language, the name Julius Caesar would be an API identifier.

So here's how I, in my mind, divide the words in a Java program into categories:

- **Keywords:** A *keyword* is a word that has its own special meaning in the Java programming language, and that meaning doesn't change from one program to another. Examples of keywords in Java are `if`, `else`, and `do`.

  The JCP committee members, who have the final say on what constitutes a Java program, have chosen all the Java keywords. If you think about the two parts of Java, which I discuss earlier, in the section "The grammar and the common names," the Java keywords belong solidly to the language specification.

- **Identifiers:** An *identifier* is a name for something. The identifier's meaning can change from one program to another, but some identifiers' meanings tend to change more:

  - *Identifiers created by you and me:* As a Java programmer (yes, even as a novice Java programmer), you create new names for classes and other items you describe in your programs. Of course, you may name something `Prime`, and the guy writing code two cubicles down the hall can name something else `Prime`. That's okay because Java doesn't have a predetermined meaning for `Prime`. In your program, you can make `Prime` stand for the Federal Reserve's prime rate. And the guy down the hall can make `Prime` stand for the "bread, roll, preserves, and prime rib." A conflict doesn't arise, because you and your coworker are writing two different Java programs.

- *Identifiers from the API:* The JCP members have created names for many things and thrown tens of thousands of these names into the Java API. The API

comes with each version of Java, so these names are available to anyone who writes a Java program. Examples of such names are `String`, `Integer`, `JWindow`, `JButton`, `JTextField`, and `File`.

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 `JButton` or `JWindow`, this isn't a good idea. If you did, you would confuse the dickens out of other programmers, who are used to the standard API meanings for these familiar identifier names. But even worse, when your code assigns a new meaning to an identifier like `JButton`, you lose any computational power that was created for the identifier in the API code. The programmers at Sun Microsystems, Oracle, the Java Community Process, and the OpenJDK Project did all the work of writing Java code to handle buttons. If you assign your own meaning to `JButton`, you're turning your back on all the progress made in creating the API.

To see the list of Java keywords, visit this book's website: `www.allmycode.com/JavaForDummies`.

Checking Out Java Code for the First Time

The first time you look at somebody else's Java program, you may tend to feel a bit queasy. The realization that you don't understand something (or many things) in the code can make you nervous. I've written hundreds (maybe thousands) of Java programs, but I still feel insecure when I start reading someone else's code.

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

In Listing 3-1, you get a blast of Java code. (Like all novice programmers, you're expected to gawk humbly at the code.) Hidden in the code, I've placed some important ideas, which I explain in detail in the next section. These ideas include the use of classes, methods, and Java statements.

The Java class

Because Java is an object-oriented programming language, your primary goal is to describe classes and objects. (If you're not convinced about this, read the sections on object-oriented programming in Chapter 1.)

On those special days when I'm feeling sentimental, I tell people that Java is more pure in its object-orientation than many other so-called object-oriented languages. I say this because, in Java, you can't do anything until you create a class of some kind. It's like being on *Jeopardy!* and hearing Alex Trebek say, "Let's go to a commercial" and then interrupting him by saying, "I'm sorry, Alex. You can't issue an instruction without putting your instruction inside a class."

The code in Listing 3-1 is a Java program, and that program describes a class. I wrote the program, so I get to make up a name for my new class. I chose the name `Displayer` because the program displays a line of text on the computer screen. That's why the first line in Listing 3-1 contains the words `class Displayer`. (See Figure 3-2.)

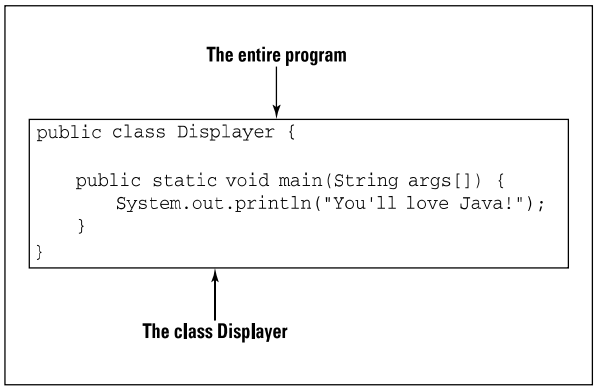

FIGURE 3-2: A Java program is a class.

The first two words in Listing 3-1, `public` and `class`, are Java keywords. (See the section "The words in a Java program," earlier in this chapter.) No matter who writes a Java program, the words `public` and `class` are always used in the same way. On the other hand, `Displayer` in Listing 3-1 is an identifier. I made up the word `Displayer` while I was writing this chapter. `Displayer` is the name of a particular class — the class that I'm creating by writing this program.

LISTING 3-1: The Simplest Java Program

```
public class Displayer {

    public static void main(String args[]) {
        System.out.println("You'll love Java!");
    }
}
```

You don't have to type the code in Listing 3-1 (or in any of this book's listings). To download all the code in this book, visit the book's website (`www.allmycode.com/JavaForDummies`).

When you run the program from Listing 3-1, the computer displays `You'll love Java!` (Figure 3-1 shows the output of the Displayer program when you use the Eclipse IDE.) Now, I admit that writing and running a Java program is a lot of work just to get `You'll love Java!` to appear on somebody's computer screen, but every endeavor has to start somewhere.

FIGURE 3-1: I use Eclipse to run the program in Listing 3-1.

To see how to run the code in Listing 3-1, visit this book's website (`www.allmycode.com/JavaForDummies`).

In the following section, you do more than just admire the program's output. After you read the following section, you actually understand what makes the program in Listing 3-1 work.

Understanding a Simple Java Program

This section presents, explains, analyzes, dissects, and otherwise demystifies the Java program shown previously in Listing 3-1.

This book is filled with talk about classes, but for the best description of a Java class (the reason for using the word `class` in Listing 3-1), visit Chapter 7. The word `public` means that other Java classes (classes other than the `Displayer` class in Listing 3-1) can use the features declared in Listing 3-1. For more details about the meaning of `public` and the use of the word `public` in a Java program, see Chapters 7 and 14.

CROSS REFERENCE

tHE jAVA PROGRAMMING LANGUAGE IS cASe-sEnsITiVE. If you change a lowercase letter in a word to an UpperCase letter, you can change the word's meaning. cHANGING case can make the entire word go from being meaningful to being meaningless. In the first line of Listing 3-1, you can't replace `class` with `Class`. iF YOU DO, THE WHOLE PROGRAM STOPS WORKING. The same holds true, to some extent, for the name of a file containing a particular class. For example, the name of the class in Listing 3-1 is `Displayer`, starting with an uppercase letter `D`. So it's a good idea to save the code of Listing 3-1 in a file named `Displayer.java`, starting with an uppercase letter `D`.

WARNING

Normally, if you define a class named `DogAndPony`, the class's Java code is in a file named `DogAndPony.java`, spelled and capitalized exactly the same way that the class name is spelled and capitalized. In fact, this file-naming convention is mandatory for most examples in this book.

The Java 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 what 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 what you're supposed to do.

In this scenario, using the word *method* wouldn't be 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 associate with the method.

If you believe all that (and I hope you do), 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 did, the program might include a `fixTheAlternator` method. The list of instructions in my `fixTheAlternator` method would look something like the text in Listing 3-2.

Don't scrutinize Listings 3-2 and 3-3 too carefully. All the code in Listings 3-2 and 3-3 is fake! I made up this code so that it looks a lot like real Java code, but it's not real. What's more important, the code in Listings 3-2 and 3-3 isn't meant to illustrate all the rules about Java. So, if you have a grain of salt handy, take it with Listings 3-2 and 3-3.

WARNING

LISTING 3-2: 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 3-2), 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 3-3.

LISTING 3-3: A Method Call

```
fixTheAlternator(junkyOldFord);
```

Now that you have a basic understanding of what a method is and how it works, you can dig a little deeper into some useful terminology:

- If I'm being lazy, I refer to the code in Listing 3-2 as a *method*. If I'm not being lazy, I refer to this code as a *method declaration*.
- The method declaration in Listing 3-2 has two parts. The first line (the part with `fixTheAlternator` in it, up to but not including the open curly brace) is

or

```
How to follow the main instructions for a Displayer:
    Print "You'll love Java!" on the screen.
```

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.

Look back at Figure 3-1. When the `Displayer` program runs, the computer automatically finds the program's `main` method and executes any instructions inside the method's body. In the `Displayer` program, the `main` method's body has only one instruction. That instruction tells the computer to print `You'll love Java!` on the screen. So in Figure 3-1, `You'll love Java!` appears on the computer screen.


REMEMBER

The instructions in a method aren't executed until the method is called into action. But, if you give a method the name *main,* that method is called into action automatically.


TECHNICAL STUFF

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

How you finally tell the computer to do something

Buried deep in the heart of Listing 3-1 is the single line that actually issues a direct instruction to the computer. The line, which is highlighted in Figure 3-4, tells the computer to display `You'll love Java!` This line is a *statement.* In Java, a statement is a direct instruction that tells the computer to do something (for example, display this text, put 7 in that memory location, make a window appear).

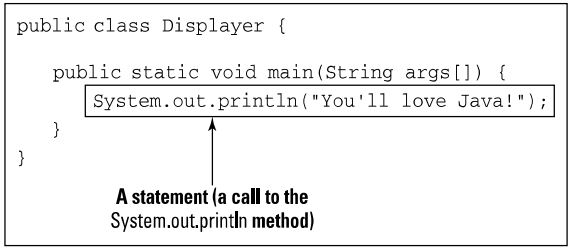

FIGURE 3-4: A Java statement.

a *method header.* The rest of Listing 3-2 (the part surrounded by curly braces) is a *method body.*

» The term *method declaration* distinguishes the list of instructions in Listing 3-2 from the instruction in Listing 3-3, which is known as a *method call.*


REMEMBER

A *method's declaration* tells the computer what happens 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

Figure 3-3 has a copy of the code from Listing 3-1. The bulk of the code contains the declaration of a method named `main`. (Just look for the word *main* in the code's method header.) For now, don't worry about the other words in the method header: `public`, `static`, `void`, `String`, and `args`. I explain these words in the next several chapters.

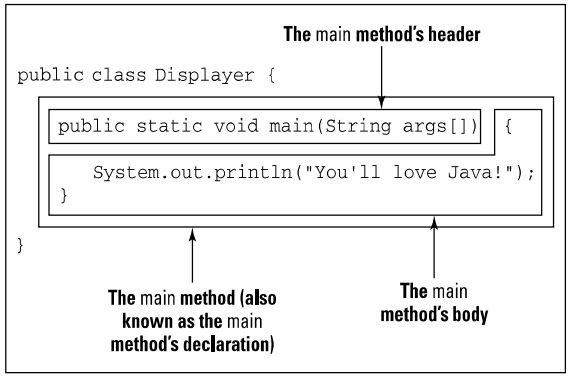

FIGURE 3-3: The `main` method.

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

```
How to make biscuits:
    Heat the oven.
    Roll the dough.
    Bake the rolled dough.
```

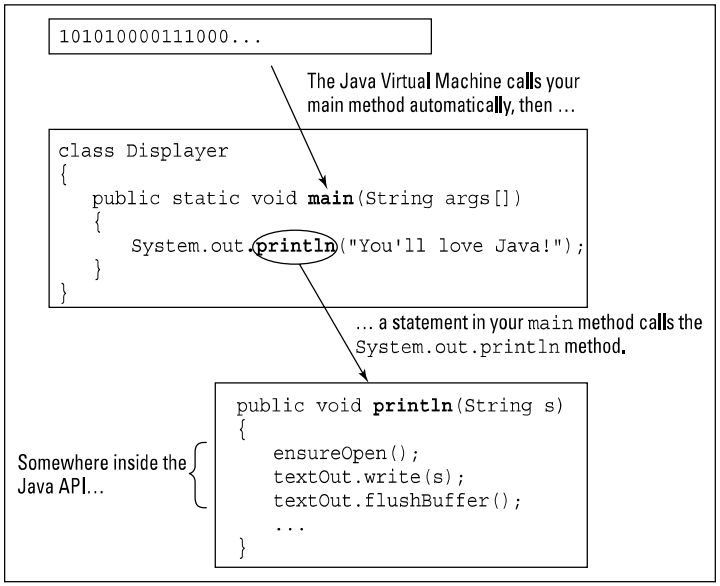

FIGURE 3-5: Calling the `System.out.println` method.

In Java, each statement (like the boxed line in Figure 3-4) ends with a semicolon. Other lines in Figure 3-4 don't end with semicolons, because the other lines in Figure 3-4 aren't statements. For instance, the method header (the line with the word *main* in it) doesn't directly tell the computer to do anything. The method header announces, "Just in case you ever want to do `main`, the next few lines of code tell you how to do it."

REMEMBER

Every complete Java statement ends with a semicolon.

Curly braces

Long ago, or maybe not so long ago, your schoolteachers told you how useful outlines are. With an outline, you can organize thoughts and ideas, help people see forests instead of trees, and generally show that you're a member of the Tidy Persons Club. Well, a Java program is like an outline. The program in Listing 3-1 starts with a header line that says, "Here comes a class named `Displayer`." After that header, a subheader announces, "Here comes a method named `main`."

REMEMBER

In `System.out.println`, the next-to-last character is a lowercase letter *l*, not a digit *1*.

Of course, Java has different kinds of statements. A method call, which I introduce in the earlier "The Java method" section, is one of the many kinds of Java statements. Listing 3-3 shows you what a method call looks like, and Figure 3-4 also contains a method call that looks like this:

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

When the computer executes this statement, the computer calls a method named `System.out.println` into action. (Yes, in Java, a name can have dots in it. The dots mean something.)

WARNING

I said it already, but it's worth repeating: In `System.out.println`, the next-to-last character is a lowercase letter *l* (*as in the word* line), not a digit *1* (*as in the number* one). If you use a digit *1*, your code won't work. Just think of `println` as a way of saying "print line" and you won't have any problem.

To learn the meaning behind the dots in Java names, see Chapter 7.

Figure 3-5 illustrates the `System.out.println` situation. Actually, two methods play active roles in the running of the `Displayer` program. Here's how they work:

- **There's a declaration for a `main` method.** I wrote the `main` method myself. This `main` method is called automatically whenever I run the `Displayer` 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 item 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, see the sections "The grammar and the common names" and "The words in a Java program," earlier in this chapter.

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 3-1.

REMEMBER

Never lose sight of the fact that a Java program is, first and foremost, an outline.

If you put curly braces in the wrong places or omit curly braces where the braces should be, your program probably won't work at all. If your program works, it'll probably work incorrectly.

If you don't indent lines of code in an informative manner, your program will still work correctly, but neither you nor any other programmer will be able to figure out what you were thinking when you wrote the code.

If you're a visual thinker, you can picture outlines of Java programs in your head. One friend of mine visualizes an actual numbered outline morphing into a Java program. (See Figure 3-6.) Another person, who shall remain nameless, uses more bizarre imagery. (See Figure 3-7.)

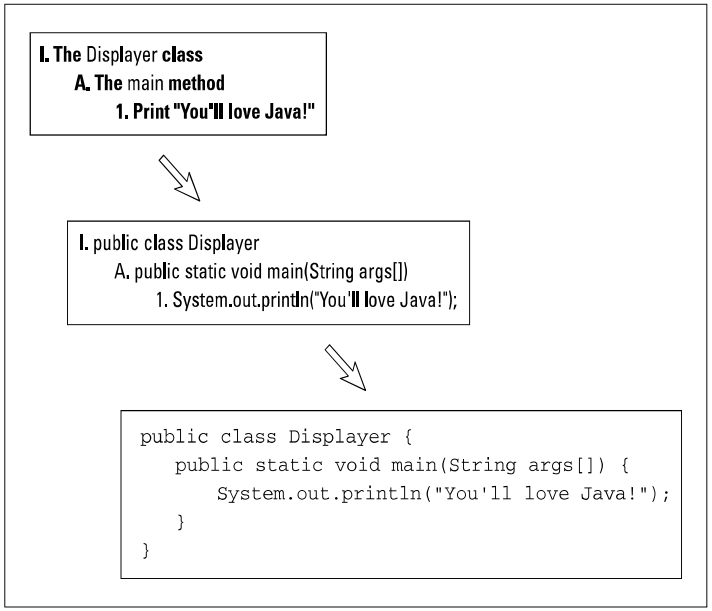

FIGURE 3-6: An outline turns into a Java program.

I appreciate a good excuse as much as the next guy, but failing to indent your Java code is inexcusable. In fact, many Java IDEs have tools to indent your code automatically. Visit this book's website (`www.allmycode.com/JavaForDummies`) for more information.

Now, if a Java program is like an outline, why doesn't a program *look* like an outline? What takes the place of the Roman numerals, capital letters, and other items? The answer is twofold:

- In a Java program, curly braces enclose meaningful units of code.
- You, the programmer, can (and should) indent lines so that other programmers can see at a glance the outline form of your code.

In an outline, everything is subordinate to the item in Roman numeral *I*. In a Java program, everything is subordinate to the top line — the line with `class` in it. To indicate that everything else in the code is subordinate to this `class` line, you use curly braces. Everything else in the code goes inside these curly braces. (See Listing 3-4.)

LISTING 3-4: Curly Braces for a Java Class

```
public class Displayer {

    public static void main(String args[]) {
        System.out.println("You'll love Java!");
    }
}
```

In an outline, some stuff is subordinate to a capital letter *A* item. In a Java program, some lines are subordinate to the method header. To indicate that something is subordinate to a method header, you use curly braces. (See Listing 3-5.)

LISTING 3-5: Curly Braces for a Java Method

```
public class Displayer {

    public static void main(String args[]) {
        System.out.println("You'll love Java!");
    }
}
```

In an outline, some items are at the bottom of the food chain. In the `Displayer` class, the corresponding line is the line that begins with `System.out.println`. Accordingly, this `System.out.println` line goes inside all the other curly braces and is indented more than any other line.

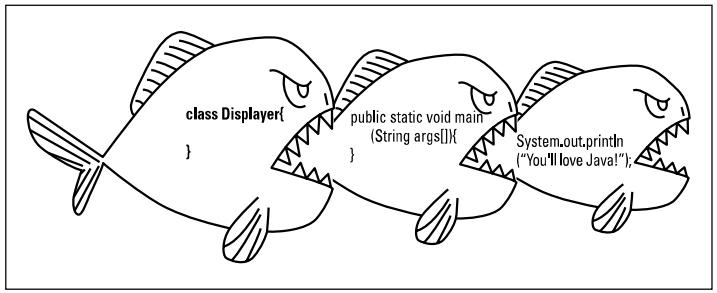

FIGURE 3-7: A class is bigger than a method; a method is bigger than a statement.

TRY IT OUT

Here are some things for you to try to help you understand the material in this section. 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 (`www.allmycode.com/JavaForDummies`). You can also email me with your questions (`JavaForDummies@allmycode.com`).

- If you've downloaded the code from this book's website, import Listing 3-1 (from the downloaded 03-01 folder) into your IDE. If you don't plan to download the code, create a new project in your IDE. In the new project, create a class named `Displayer` with the code from Listing 3-1. With the downloaded project, or with your own, newly created project, run the program and look for the words `You'll love Java!` in the output.
- Try running the code in Listing 3-1 with the text `"You'll love Java!"` changed to `"No more baked beans!"`. What happens?
- Try to run the code in Listing 3-1 with the word `public` (all lowercase) changed to `Public` (starting with an uppercase letter). What happens?
- Try to run the code in Listing 3-1 with the word `main` (all lowercase) changed to `Main` (starting with an uppercase letter). What happens?
- Try to run the code in Listing 3-1 with the word `System` (starting with an uppercase letter) changed to `system` (all lowercase). What happens?
- Try to run the code in Listing 3-1 with the indentation changed. For example, don't indent any lines. Also, for good measure, remove the line breaks between the first curly brace and the word `public` (so that the code reads `public class Displayer { public ...`). What happens?
- Try to run the code in Listing 3-1 with the word `println` changed to `print1n` (with the digit `1` near the end). What happens?
- Try to run the code in Listing 3-1 with the semicolon missing. What happens?

- Try to run the code in Listing 3-1 with additional semicolons added at the ends of some of the lines. What happens?
- Try to run the code in Listing 3-1 with the text `"You'll love Java!"` changed to `" Use a straight quote \", not a curly quote \u201D".` What happens?

And Now, a Few Comments

People gather around campfires to hear the old legend about a programmer whose laziness got her into trouble. To maintain this programmer's anonymity, I call her Jane Pro. Jane worked many months to create the holy grail of computing: a program that thinks on its own. If completed, this program could work independently, learning new things without human intervention. Day after day, night after night, Jane Pro labored to give the program that spark of creative, independent thought.

One day, when she was almost finished with the project, she received a disturbing piece of paper mail from her health insurance company. No, the mail wasn't about a serious illness. It was about a routine office visit. The insurance company's claim form had a place for Jane's date of birth, as if her date of birth had changed since the last time she sent in a claim. She had absentmindedly scribbled 2016 as her year of birth, so the insurance company refused to pay the bill.

Jane dialed the insurance company's phone number. Within 20 minutes, she was talking to a live person. "I'm sorry," said the live person. "To resolve this issue, you must dial a different number." Well, you can guess what happened next. "I'm sorry. The other operator gave you the wrong number." And then, "I'm sorry. You must call back the original phone number."

Five months later, Jane's ear ached, but after 800 hours on the phone, she had finally gotten a tentative promise that the insurance company would eventually reprocess the claim. Elated as she was, she was anxious to get back to her programming project. Could she remember what all those lines of code were supposed to be doing?

No, she couldn't. Jane stared and stared at her own work and, like a dream that doesn't make sense the next morning, the code was completely meaningless to her. She had written a million lines of code, and not one line was accompanied by an informative explanatory comment. She had left no clues to help her understand what she'd been thinking, so in frustration, she abandoned the whole project.

Adding comments to your code

Listing 3-6 has an enhanced version of this chapter's sample program. In addition to all the keywords, identifiers, and punctuation, Listing 3-6 has text that's meant for human beings to read.

LISTING 3-6: Three Kinds of Comments

```
/*
 * Listing 3-6 in "Java For Dummies, 7th Edition"
 *
 * Copyright 2017 Wiley Publishing, Inc.
 * All rights reserved.
 */

/**
 * The Displayer class displays text
 * on the computer screen.
 *
 * @author  Barry Burd
 * @version 1.0 1/24/17
 * @see     java.lang.System
 */
public class Displayer {

    /**
     * The main method is where
     * execution of the code begins.
     *
     * @param  args   (See Chapter 11.)
     */
    public static void main(String args[]) {
        System.out.println("I love Java!");  //I? You?
    }
}
```

A *comment* is a special section of text, inside a program, whose purpose is to help people understand the program. A comment is part of a good program's documentation.

The Java programming language has three kinds of comments:

- **Traditional comments:** The first five lines of Listing 3-6 form one *traditional* comment. The comment begins with `/*` and ends with `*/`. Everything between the opening `/*` and the closing `*/` is for human eyes only. No information about `"Java For Dummies, 7th Edition"` or `Wiley Publishing, Inc.` is translated by the compiler.

  To read about compilers, see Chapter 2.

  The second, third, fourth, and fifth lines in Listing 3-6 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 3-6 because, for some reason that I don't entirely understand, most Java programmers add these extra asterisks.

- **End-of-line comments:** The text `//I? You?` in Listing 3-6 is an end-of-line comment. An *end-of-line* comment starts with two slashes and goes to the end of a line of type. Once again, the compiler doesn't translate the text inside the end-of-line comment.

- **Javadoc comments:** A *javadoc* comment begins with a slash and two asterisks (`/**`). Listing 3-6 has two javadoc comments: one with the text `The Displayer class ...` and another with the text `The main method is where ....`

  A *javadoc* comment, which is a special kind of traditional comment, is meant to be read by people who never even look at the Java code. But that doesn't make sense. How can you see the javadoc comments in Listing 3-6 if you never look at Listing 3-6?

- Well, a certain program called *javadoc* (what else?) can find all the javadoc comments in Listing 3-6 and turn these comments into a nice-looking web page. Figure 3-8 shows the page.

CROSS REFERENCE

Javadoc comments are great. Here are several great things about them:

- The only person who has to look at a piece of Java code is the programmer who writes the code. Other people who use the code can find out what the code does by viewing the automatically generated web page.
- Because other people don't look at the Java code, other people don't make changes to the Java code. (In other words, other people don't introduce errors into the existing Java code.)
- Because other people don't look at the Java code, other people don't have to decipher the inner workings of the Java code. All these people need to know about the code is what they read on the code's web page.

- The programmer doesn't create two separate files — some Java code over here and some documentation about the code over there. Instead, the programmer creates one piece of Java code and embeds the documentation (in the form of javadoc comments) right inside the code.
- Best of all, the generation of web pages from javadoc comments is automatic. So everyone's documentation has the same format. No matter whose Java code you use, you find out about that code by reading a page like the one in Figure 3-8. That's good because the format in Figure 3-8 is familiar to anyone who uses Java.

You can generate your own web pages from the javadoc comments that you put in your code. To discover how, visit this book's website (`www.allmycode.com/JavaForDummies`).

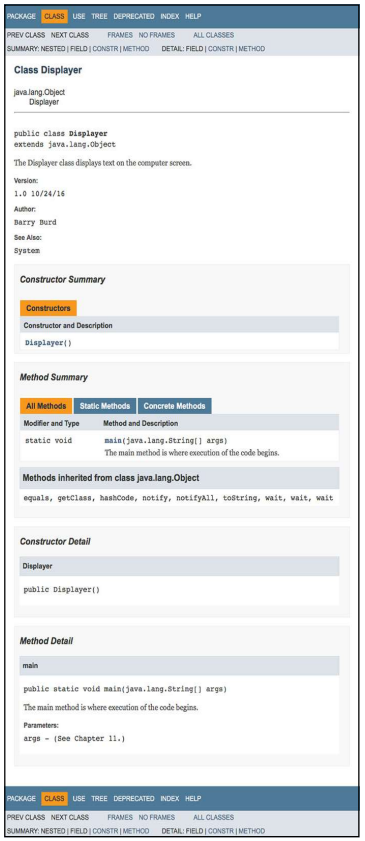

FIGURE 3-8: The javadoc page generated from the code in Listing 3-6.

What's Barry's excuse?

For years I've been telling my students to put comments in their code, and for years I've been creating sample code (like the code in Listing 3-1) with no comments in it. Why?

Three little words: *Know your audience.* When you write complicated, real-life code, your audience is other programmers, information technology managers, and people who need help deciphering what you've done. When I write simple samples of code for this book, my audience is you — the novice Java programmer. Instead of reading my comments, your best strategy is to stare at my Java statements — the statements that Java's compiler deciphers. That's why I put so few comments in this book's listings.

Besides, I'm a little lazy.

Using comments to experiment with your code

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 try removing 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 instance, you turn the statement

```
System.out.println("I love Java!");
```

into the comment

```
// System.out.println("I love Java!");
```

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

Traditional comments aren't very useful for commenting out code. The big problem is that you can't put one traditional comment inside of another. Suppose that you want to comment out the following statements:

```
System.out.println("Parents,");
System.out.println("pick your");
/*
```

2 Writing Your Own Java Programs

```
 * Intentionally displays on four separate lines
 */
System.out.println("battles");
System.out.println("carefully!");
```

If you try to turn this code into one traditional comment, you get the following mess:

```
/*
  System.out.println("Parents,");
  System.out.println("pick your");
  /*
   * Intentionally displays on four separate lines
   */
  System.out.println("battles");
  System.out.println("carefully!");
*/
```

The first `*/` (after `Intentionally displays`) ends the traditional comment prematurely. Then the `battles` and `carefully` statements aren't commented out, and the last `*/` chokes the compiler. You can't nest traditional comments inside one another. Because of this, I recommend end-of-line comments as tools for experimenting with your code.

Most IDEs can comment out sections of your code for you automatically. For details, visit this book's website (`www.allmycode.com/JavaForDummies`).

IN THIS CHAPTER

- Assigning values to things
- Making things store certain types of values
- Applying operators to get new values

Chapter 4

Making the Most of Variables and Their Values

The following conversation between Mr. Van Doren and Mr. Barasch never took place:

Charles: A sea squirt eats its brain, turning itself from an animal into a plant.

Jack: Is that your final answer, Charles?

Charles: Yes, it is.

Jack: How much money do you have in your account today, Charles?

Charles: I have fifty dollars and twenty-two cents in my checking account.

Jack: Well, you better call the IRS, because your sea squirt answer is correct. You just won a million dollars to add to your checking account. What do you think of that, Charles?

Charles: I owe it all to honesty, diligence, and hard work, Jack.

Some aspects of this dialogue can be represented in Java by a few lines of code.

IN THIS PART . . .

Create new values and modify existing values.

Put decision-making into your application's logic.

Repeat things as needed when your program runs.

Varying a Variable

No matter how you acquire your million dollars, you can use a variable to tally your wealth. Listing 4-1 shows the code.

LISTING 4-1: Using a Variable

```
amountInAccount = 50.22;
amountInAccount = amountInAccount + 1000000.00;
```

You don't have to type the code in Listing 4-1 (or in any of this book's listings). To download all the code in this book, visit the book's website (`www.allmycode.com/JavaForDummies`).

The code in Listing 4-1 makes use of the `amountInAccount` variable. A *variable* is a placeholder. You can stick a number like 50.22 into a variable. After you place a number in the variable, you can change your mind and put a different number 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 4-1 gives a before-and-after picture of the code in Listing 4-1. After the first statement in Listing 4-1 is executed, the variable `amountInAccount` has the number 50.22 in it. Then, after the second statement of Listing 4-1 is executed, the `amountInAccount` variable suddenly has 1000050.22 in it. When you think about a variable, picture a place in the computer's memory where wires and transistors store 50.22, 1000050.22, or whatever. On the left side of Figure 4-1, imagine that the box with 50.22 in it is surrounded by millions of other such boxes.

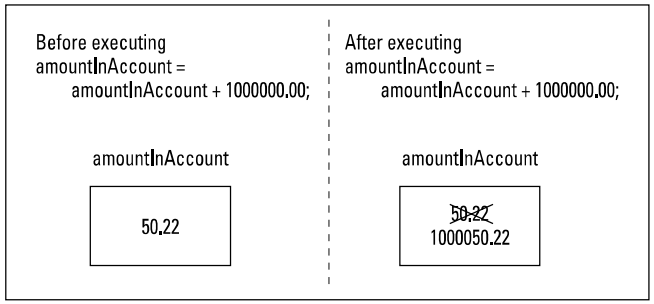

FIGURE 4-1: A variable (before and after).

Now you need some terminology. The thing stored in a variable is a *value*. A variable's value can change during the run of a program (when Jack gives you a million bucks, for instance). The value that's stored in a variable isn't necessarily a number. (For instance, you can create a variable that always stores a letter.) The kind of value that's stored in a variable is a variable's *type*.

You can read more about types in the section "The types of values that variables may have," later in this chapter.

TECHNICAL STUFF

A subtle, almost unnoticeable difference exists 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, `amountInAccount` is a variable name, and all the memory storage associated with `amountInAccount` (including the type that `amountInAccount` has and whatever value `amountInAccount` currently represents) 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. In preparing Listing 4-1, I made up the name *amountInAccount*.

CROSS REFERENCE

For more information on the kinds of names in a Java program, see Chapter 3.

Before the sun sets on Listing 4-1, you need to notice one more part of the listing. The listing has `50.22` and `1000000.00` in it. Anybody in his or her right mind would call these things *numbers*, but in a Java program it helps to call these things *literals*.

And what's so literal about `50.22` and `1000000.00`? Well, think about the variable `amountInAccount` in Listing 4-1. The variable `amountInAccount` stands for 50.22 some of the time, but it stands for 1000050.22 the rest of the time. You could use the word *number* to talk about `amountInAccount`. But really, what `amountInAccount` stands for depends on the fashion of the moment. On the other hand, 50.22 literally stands for the value 50 22/100.

REMEMBER

A variable's value changes; a literal's value doesn't.

TIP

Starting with Java 7, you can add underscores to numeric literals. Instead of using the plain old `1000000.00` in Listing 4-1, you can write `amountInAccount = amountInAccount + 1_000_000.00`. Unfortunately, you can't easily do what you're most tempted to do. You can't write `1,000,000.00` (as you would in the United States), nor can you write `1.000.000,00` (as you would in Germany). If you want to display a number such as `1,000,000.00` in the program's output, you have to use some fancy formatting tricks. For more information about formatting, check Chapters 10 and 11.

Assignment statements

Statements like the ones in Listing 4-1 are called *assignment statements.* In an assignment statement, you assign a value to something. In many cases, this something is a variable.

I recommend getting into the habit of reading assignment statements from right to left. Figure 4-2 illustrates the action of the first line in Listing 4-1.

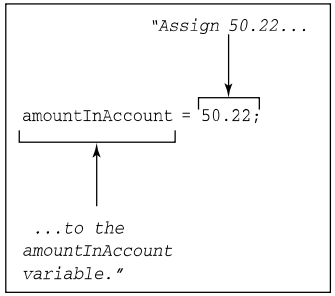

FIGURE 4-2: The action of the first line in Listing 4-1.

The second line in Listing 4-1 is just a bit more complicated. Figure 4-3 illustrates the action of the second line in Listing 4-1.

REMEMBER

In an assignment statement, the thing being assigned a value is always on the left side of the equal sign.

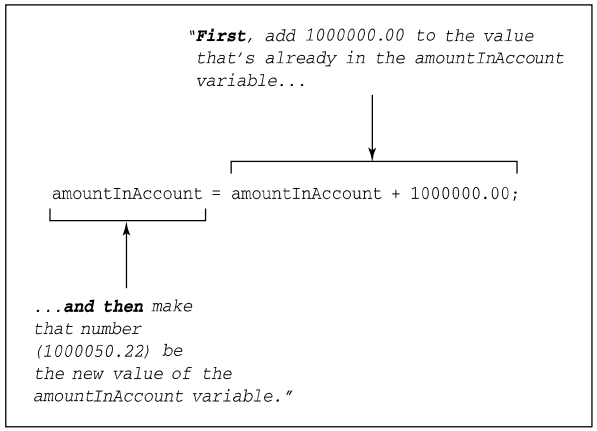

FIGURE 4-3: The action of the second line in Listing 4-1.

The types of values that variables may have

Have you seen the TV commercials that make you think you're flying among the circuits inside a computer? Pretty cool, eh? These commercials show 0s (zeros) and 1s (ones) sailing by because 0s and 1s are the only things that computers can deal with. When you think a computer is storing the letter *J*, the computer is really storing 01001010. Everything inside the computer is a sequence of 0s and 1s. As every computer geek knows, a 0 or 1 is called a *bit.*

As it turns out, the sequence 01001010, which stands for the letter *J*, can also stand for the number 74. The same sequence can also stand for $1.0369608636003646 \times 10^{-43}$. In fact, if the bits are interpreted as screen pixels, the same sequence can be used to represent the dots shown in Figure 4-4. The meaning of 01001010 depends on the way the software interprets this sequence of 0s and 1s.

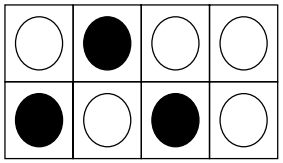

FIGURE 4-4: An extreme close-up of eight black and white screen pixels.

How do you tell the computer what 01001010 stands for? The answer is in the concept of type. The *type* of a variable is the range of values that the variable is permitted to store.

I copied the lines from Listing 4-1 and put them into a complete Java program. The program is in Listing 4-2. When I run the program in Listing 4-2, I get the output shown in Figure 4-5.

LISTING 4-2: A Program Uses amountInAccount

```
public class Millionaire {

    public static void main(String args[]) {
        double amountInAccount;

        amountInAccount = 50.22;
        amountInAccount = amountInAccount + 1000000.00;
```

(continued)

LISTING 4-2: *(continued)*

```
        System.out.print("You have $");
        System.out.print(amountInAccount);
        System.out.println(" in your account.");
    }
}
```

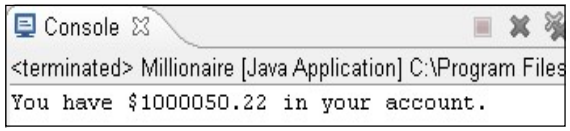

FIGURE 4-5: Running the program in Listing 4-2.

In Listing 4-2, look at the first line in the body of the `main` method:

```
double amountInAccount;
```

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 *amountInAccount* in my program." This line reserves the name *amountInAccount* for your use in the program.

In this variable declaration, the word *double* is a Java keyword. This word *double* tells the computer what kinds of values you intend to store in `amountInAccount`. In particular, the word *double* stands for numbers between -1.8×10^{308} and 1.8×10^{308}. (These are enormous numbers with 308 zeros before the decimal point. Only the world's richest people write checks with 308 zeros in them. The second of these numbers is one-point-eight gazazzo-zillion-kaskillion. The number 1.8×10^{308}, a constant defined by the International Bureau of Weights and Measures, is the number of eccentric computer programmers between Sunnyvale, California, and the M31 Andromeda Galaxy.)

More important than the humongous range of the `double` keyword's numbers is the fact that a `double` value can have digits beyond the decimal point. After you declare `amountInAccount` to be of type `double`, you can store all sorts of numbers in `amountInAccount`. You can store `50.22`, `0.02398479`, or `-3.0`. In Listing 4-2, if I hadn't declared `amountInAccount` to be of type `double`, I may not have been able to store `50.22`. Instead, I would have had to store plain old `50`, without any digits beyond the decimal point.

Another type — type `float` — also allows you to have digits beyond the decimal point. But `float` values aren't as accurate as `double` values.

DIGITS BEYOND THE DECIMAL POINT

Java has two different types that have digits beyond the decimal point: type `double` and type `float`. So what's the difference? When you declare a variable to be of type `double`, you're telling the computer to keep track of 64 bits when it stores the variable's values. When you declare a variable to be of type `float`, the computer keeps track of only 32 bits.

You could change Listing 4-2 and declare `amountInAccount` to be of type `float`.

```
float amountInAccount;
```

Surely, 32 bits are enough to store a small number like 50.22, right? Well, they are and they aren't. You could easily store 50.00 with only 32 bits. Heck, you could store 50.00 with only 6 bits. The size of the number doesn't matter. The accuracy matters. In a 64-bit double variable, you're using most of the bits to store stuff beyond the decimal point. To store the .22 part of 50.22, you need more than the measly 32 bits that you get with type `float`.

Do you really believe what you just read — that it takes more than 32 bits to store .22? To help convince you, I made a few changes to the code in Listing 4-2. I made `amountInAccount` be of type `float`. Then I changed the first three statements inside the main method as follows:

```
float amountInAccount;

amountInAccount = 50.22F;
amountInAccount = amountInAccount + 1000000.00F;
```

(To understand why I used the letter `F` in `50.22F` and `1000000.00F`, see Table 4-1, later in this chapter.) The output I got was

```
You have $1000050.25 in your account.
```

Compare this with the output in Figure 4-5. When I switch from type `double` to type `float`, Charles has an extra three cents in his account. By changing to the 32-bit `float` type, I've clobbered the accuracy in the `amountInAccount` variable's hundredths place. That's bad.

Another difficulty with `float` values is purely cosmetic. Look again at the literals, `50.22` and `1000000.00`, in Listing 4-2. The Laws of Java say that literals like these take up 64 bits each. So, if you declare `amountInAccount` to be of type `float`, you'll run into trouble. You'll have trouble stuffing those 64-bit literals into your little 32-bit

(continued)

TRY IT OUT

Run the code in Listing 4-2 to make sure that it runs correctly on your computer. Then see what happens when you make the following changes:

- Add thousands-separators to the number `1000000.00` in the code. For example, if you live in the United States, where the thousands-separator is a comma, change the number to `1,000,000.00` and see what happens. (***Hint:*** Nothing good happens.)
- Try using underscores as thousands-separators in the code. That is, change `1000000.00` to `1_000_000.00` and see what happens.
- Add a currency symbol to the number `50.22` in the code. For example, if you live in the United States, where the currency symbol is $, see what happens when you change the first assignment statement to `amountInAccount = $50.22`.
- Listing 4-2 has two `System.out.print` statements and one `System.out.println` statement. Change all three to `System.out.println` statements and then run the program.
- The code in Listing 4-2 displays one line of text in its output. Using the `amountInAccount` variable, add statements to the program so that it displays a second line of text. Have the second line of text be "Now you have even more! You have 2000000.00 in your account."

Numbers without decimal points

"In 1995, the average family had 2.3 children."

At this point, a wise guy always remarks that no real family has exactly 2.3 children. Clearly, whole numbers have a role in this world. Therefore, in Java, you can declare a variable to store nothing but whole numbers. Listing 4-3 shows a program that uses whole number variables.

LISTING 4-3: Using the int Type

```
public class ElevatorFitter {

    public static void main(String args[]) {
        int weightOfAPerson;
        int elevatorWeightLimit;
        int numberOfPeople;
```

(continued)

(continued)

amountInAccount variable. To compensate, you can switch from `double` literals to `float` literals by adding an `F` to each `double` literal, but a number with an extra `F` at the end looks funny.

```
float amountInAccount;
amountInAccount = 50.22F;
amountInAccount = amountInAccount + 1000000.00F;
```

To experiment with numbers, visit `http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html`. The page takes any number you enter and shows you how the number would be represented as 32 bits and as 64 bits.

TIP

In many situations, you have a choice. You can declare certain values to be either `float` values or `double` values. But don't sweat the choice between `float` and `double`. For most programs, just use `double`. With today's fancy processors, the space you save using the `float` type is almost never worth the loss of accuracy. (For more details, see the nearby sidebar, "Digits beyond the decimal point.")

The big million-dollar jackpot in Listing 4-2 is impressive. But Listing 4-2 doesn't illustrate the best way to deal with dollar amounts. In a Java program, the best way to represent currency is to shun the `double` and `float` types and opt instead for a type named `BigDecimal`. For more information, see this book's website (`www.allmycode.com/JavaForDummies`).

Displaying text

The last three statements in Listing 4-2 use a neat formatting trick. You want to display several different items on a single line on the screen. You put these items in separate statements. All but the last of the 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 executing `System.out.print`, 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. (Refer to Figure 4-5.)

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.

LISTING 4-3: *(continued)*

```
        weightOfAPerson = 150;
        elevatorWeightLimit = 1400;
        numberOfPeople = elevatorWeightLimit / weightOfAPerson;

        System.out.print("You can fit ");
        System.out.print(numberOfPeople);
        System.out.println(" people on the elevator.");
    }
}
```

The story behind the program in Listing 4-3 takes some heavy-duty explaining. Here goes:

You have a hotel elevator whose weight capacity is 1,400 pounds. One weekend the hotel hosts the Brickenchicker family reunion. A certain branch of the Brickenchicker family has been blessed with identical dectuplets (ten siblings, all with the same physical characteristics). Normally, each of the Brickenchicker dectuplets weighs exactly 145 pounds. But on Saturday the family has a big catered lunch, and, because lunch included strawberry shortcake, each of the Brickenchicker dectuplets now weighs 150 pounds. Immediately after lunch, all ten of the Brickenchicker dectuplets arrive at the elevator at exactly the same time. (Why not? All ten of them think alike.) So, the question is, how many of the dectuplets can fit on the elevator?

Now remember, if you put one ounce more than 1,400 pounds of weight on the elevator, the elevator cable breaks, plunging all dectuplets on the elevator to their sudden (and costly) deaths.

The answer to the Brickenchicker riddle (the output of the program of Listing 4-3) is shown in Figure 4-6.

FIGURE 4-6: Save the Brickenchickers.

At the core of the Brickenchicker elevator problem, you have whole numbers — numbers with no digits beyond the decimal point. When you divide 1,400 by 150, you get 9⅓, but you shouldn't take the ⅓ seriously. No matter how hard you try,

you can't squeeze an extra 50 pounds' worth of Brickenchicker dectuplet onto the elevator. This fact is reflected nicely in Java. In Listing 4-3, all three variables (`weightOfAPerson`, `elevatorWeightLimit`, and `numberOfPeople`) 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 4-3), you get another `int`. When you divide 1,400 by 150, you get 9 — not 9⅓. You see this in Figure 4-6. Taken together, the following statements display 9 onscreen:

```
numberOfPeople = elevatorWeightLimit / weightOfAPerson;

System.out.print(numberOfPeople);
```

TRY IT OUT

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`.

Combining declarations and initializing variables

Look back at Listing 4-3. In that listing, you see three variable declarations — one for each of the program's three `int` variables. I could have done the same thing with just one declaration:

```
int weightOfAPerson, elevatorWeightLimit, numberOfPeople;
```

FOUR WAYS TO STORE WHOLE NUMBERS

Java has four types of whole numbers. The types are `byte`, `short`, `int`, and `long`. Unlike the complicated story about the accuracy of types `float` and `double`, the only thing that matters when you choose among the whole number types is the size of the number you're trying to store. If you want to use numbers larger than 127, don't use `byte`. To store numbers larger than 32767, don't use `short`.

Most of the time, you'll use `int`. But if you need to store numbers larger than 2147483647, forsake `int` in favor of `long`. (A `long` number can be as big as 9223372036854775807.) For the whole story, see Table 4-1, a little later in this chapter.

If two variables have completely different types, you can't create both variables in the same declaration. For instance, to create an `int` variable named *weightOfFred* and a `double` variable named *amountInFredsAccount*, you need two separate variable declarations.

WARNING

You can give variables their starting values in a declaration. In Listing 4-3, for instance, one declaration can replace several lines in the `main` method (all but the calls to `print` and `println`):

```
int weightOfAPerson = 150, elevatorWeightLimit = 1400,
  numberOfPeople = elevatorWeightLimit/weightOfAPerson;
```

When you do this, you don't say that you're assigning values to variables. The pieces of the declarations with equal signs in them aren't really called assignment statements. Instead, you say that you're *initializing* the variables. Believe it or not, keeping this distinction in mind is helpful.

Like everything else in life, initializing a variable has advantages and disadvantages:

- **When you combine six lines of Listing 4-3 into just one declaration, the code becomes more concise.** Sometimes concise code is easier to read. Sometimes it's not. As a programmer, it's your judgment call.
- **By initializing a variable, you might automatically avoid certain programming errors.** For an example, see Chapter 7.
- **In some situations, you have no choice. The nature of your code forces you either to initialize or not to initialize.** For an example that doesn't lend itself to variable initialization, see the deleting-evidence program in Chapter 6.

Experimenting with JShell

The programs in Listings 4-2 and 4-3 both begin with the same old, tiresome refrain:

```
public class SomethingOrOther {

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


A Java program requires this verbose introduction because

- In Java the entire program is a class.
- The `main` method is called into action automatically when the program begins running.

I explain all of this in Chapter 3.

Anyway, retyping this boilerplate code into an editor window can be annoying, especially when your goal is to test the effect of executing a few simple statements. To fix this problem, the stewards of Java came up with a new tool in Java 9. They call it *JShell*.

Instructions for launching JShell differ from one computer to the next. For instructions that work on your computer, visit this book's website (`www.allmycode.com/JavaForDummies`).

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.

JShell is only one example of a language's *Read Evaluate Print Loop* (REPL). Many programming languages have REPLs and, with Java 9, the Java language finally has a REPL of its own.

In Figure 4-7, I use JShell to find out how Java responds to the assignment statements in Listings 4-2 and 4-3.

When you run JShell, the dialogue goes something like this:

```
jshell> You type a statement
JShell responds

jshell> You type another statement
JShell responds
```

For example, in Figure 4-7, I type `double amountInAccount` and then press Enter. JShell responds by displaying

```
amountInAccount ==> 0.0
```

```
jshell> double amountInAccount
amountInAccount ==> 0.0

jshell> amountInAccount = 50.22
amountInAccount ==> 50.22

jshell> amountInAccount = amountInAccount + 1000000.00
amountInAccount ==> 1000050.22

jshell> int weightOfAPerson, elevatorWeightLimit
weightOfAPerson ==> 0
elevatorWeightLimit ==> 0

jshell> weightOfAPerson = 150;
weightOfAPerson ==> 150

jshell> elevatorWeightLimit = 1400
elevatorWeightLimit ==> 1400

jshell> elevatorWeightLimit / weightOfAPerson
$8 ==> 9

jshell> $8 + 1
$9 ==> 10

jshell> 42 + 7
$10 ==> 49

jshell>
```

FIGURE 4-7: An intimate conversation between me and JShell.

Here are a few things to notice about JShell:

- **You don't have to type an entire Java program.**

  Typing a few statements such as

  ```
  double amountInAccount
  amountInAccount = 50.22
  amountInAccount = amountInAccount + 1000000.00
  ```

  does the trick. It's like running the code snippet in Listing 4-1 (except that Listing 4-1 doesn't declare `amountInAccount` to be a `double`).

- **In JShell, semicolons are (to a large extent) optional.**

  In Figure 4-7, I type a semicolon at the end of only one of my nine lines.

  For some advice about using semicolons in JShell, see Chapter 5.

CROSS REFERENCE

- **JShell responds immediately after you type each line.**

  After I declare `amountInAccount` to be `double`, JShell responds by telling me that the `amountInAccount` variable has the value 0.0. After I type `amountInAccount = amountInAccount + 1000000.00`, JShell tells me that the new value of `amountInAccount` is 1000050.22.

- **You can mix statements from many different Java programs.**

  In Figure 4-7, I mix statements from the programs in Listings 4-2 and 4-3. JShell doesn't care.

- **You can ask JShell for the value of an expression.**

  You don't have to assign the expression's value to a variable. For example, in Figure 4-7, I type

  ```
  elevatorWeightLimit / weightOfAPerson
  ```

  JShell responds by telling me that the value of `elevatorWeightLimit / weightOfAPerson` is 9. JShell makes up a temporary name for that value. In Figure 4-7, the name happens to be $8. So, on the next line in Figure 4-7, I ask for the value of `$8 +1`, and JShell gives me the answer 10.

- **You can even get answers from JShell without using variables.**

  On the last line in Figure 4-7, I ask for the value of 42 + 7, and JShell generously answers with the value 49.

TIP

While you're running JShell, you don't have to retype commands that you've already typed. If you press the up-arrow key once, JShell shows you the command that you typed most recently. If you press the up-arrow key twice, JShell shows you the next-to-last command that you typed. And so on. When JShell shows you a command, you can use your left- and right-arrow keys to move to any character in the middle of the command. You can modify characters in the command. Finally, when you press Enter, JShell executes your newly modified command.

To end your run of JShell, you type **/exit** (starting with a slash). But `/exit` is only one of many commands you can give to JShell. To ask JShell what other kinds of commands you can use, type **/help**.

With JShell, you can test your statements before you put them into a full-blown Java program. That makes JShell a truly useful tool.

TRY IT OUT

Visit this book's website (`www.allmycode.com/JavaForDummies`) for instructions on launching JShell on your computer. After launching JShell, type a few lines of code from Figure 4-7. See what happens when you type some slightly different lines.

TABLE 4-1: Java's Primitive Types

Type Name	What a Literal Looks Like	Range of Values
Whole number types		
`byte`	`(byte)42`	–128 to 127
`short`	`(short)42`	–32768 to 32767
`int`	`42`	–2147483648 to 2147483647
`long`	`42L`	–9223372036854775808 to 9223372036854775807
Decimal number types		
`float`	`42.0F`	-3.4×10^{38} to 3.4×10^{38}
`double`	`42.0`	-1.8×10^{308} to 1.8×10^{308}
Character type		
`char`	`'A'`	Thousands of characters, glyphs, and symbols
Logical type		
`boolean`	`true`	true, false

The types that you shouldn't ignore are `int`, `double`, `char`, and `boolean`. Previous sections in this chapter cover the `int` and `double` types. So the next two sections cover `char` and `boolean` types.

The char type

Several decades ago, people thought computers existed only for doing big number-crunching calculations. Nowadays, nobody thinks that way. So, if you haven't been in a cryogenic freezing chamber for the past 20 years, you know that computers store letters, punctuation symbols, and other characters.

The Java type that's used to store characters is called *char*. Listing 4-4 has a simple program that uses the `char` type. Figure 4-8 shows the output of the program in Listing 4-4.

What Happened to All the Cool Visual Effects?

The programs in Listings 4-2 and 4-3 are text-based. A *text-based* program has no windows, no dialog boxes — nothing of that kind. All you see is line after line of plain, unformatted text. The user types something, and the computer displays a response beneath each line of input.

The opposite of a text-based program is a *graphical user interface* (GUI) program. A GUI program has windows, text fields, buttons, and other visual goodies.

As visually unexciting as text-based programs are, they contain the basic concepts for all computer programming. Also, text-based programs are easier for the novice programmer to read, write, and understand than the corresponding GUI programs. So, in this book I take a three-pronged approach:

- **Text-based examples:** I introduce most of the new concepts with these examples.
- **The `DummiesFrame` class:** Alongside the text-based examples, I present GUI versions using the `DummiesFrame` class, which I created especially for this book. (I introduce the `DummiesFrame` class in Chapter 7.)
- **GUI programming techniques:** I describe some of the well-known techniques in Chapters 9, 10, 14, and 16. I even have a tiny GUI example in this chapter. (See the later section "The Molecules and Compounds: Reference Types.")

With this careful balance of drab programs and sparkly programs, you're sure to learn Java.

The Atoms: Java's Primitive Types

The words *int* and *double* that I describe in the previous sections are examples of *primitive types* (also known as *simple* types) in Java. The Java language has exactly eight primitive types. As a newcomer to Java, you can pretty much ignore all but four of these types. (As programming languages go, Java is nice and compact that way.) Table 4-1 shows the complete list of primitive types.

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), you need to use something called a String.

For a look at Java's `String` type, see the section "The Molecules and Compounds: Reference Types," later in this chapter.


WARNING

If you're used to writing programs in other languages, you may be aware of something called ASCII character encoding. Most languages use ASCII; Java uses Unicode. In the old ASCII representation, each character takes up only 8 bits, but in Unicode, each character takes up 8, 16, or 32 bits. Whereas ASCII stores the letters of the Roman (English) alphabet, Unicode has room for characters from most of the world's commonly spoken languages. The only problem is that some of the Java API methods are geared specially toward 16-bit Unicode. Occasionally, this bites you in the back (or it bytes you in the back, as the case may be). If you're using a method to write `Hello` on the screen and `H e l l o` shows up instead, check the method's documentation for mention of Unicode characters.

It's worth noticing that the two methods, `Character.toUpperCase` and `System.out.println`, are used quite differently in Listing 4-4. The method `Character.toUpperCase` is called as part of an initialization or an assignment statement, but the method `System.out.println` is called on its own. To find out more about this topic, see the explanation of return values in Chapter 7.

The boolean type

A variable of type `boolean` stores one of two values: `true` or `false`. Listing 4-5 demonstrates the use of a `boolean` variable. Figure 4-9 shows the output of the program in Listing 4-5.

LISTING 4-5: Using the boolean Type

```
public class ElevatorFitter2 {

    public static void main(String args[]) {
        System.out.println("True or False?");
        System.out.println("You can fit all ten of the");
        System.out.println("Brickenchicker dectuplets");
        System.out.println("on the elevator:");
        System.out.println();

        int weightOfAPerson = 150;
```

(continued)

LISTING 4-4: Using the char Type

```
public class CharDemo {

  public static void main(String args[]) {
     char myLittleChar = 'b';
     char myBigChar = Character.toUpperCase(myLittleChar);
     System.out.println(myBigChar);
  }
}
```

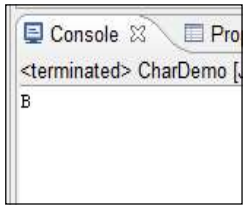

FIGURE 4-8: An exciting run of the program of Listing 4-4 as it appears in the Eclipse Console view.

In Listing 4-4, the first initialization stores the letter *b* in the variable `myLittleChar`. In the initialization, notice how *b* is surrounded by single quote marks. In Java, every `char` literal starts and ends with a single quote mark.


REMEMBER

In a Java program, single quote marks surround the letter in a `char` literal.

If you need help sorting out the terms *assignment*, *declaration*, and *initialization*, see the "Combining declarations and initializing variables" section, earlier in this chapter.

In the second initialization of Listing 4-4, the program calls an API method whose name is *Character.toUpperCase*. The `Character.toUpperCase` method does just what its name suggests — the method produces the uppercase equivalent of the letter *b*. This uppercase equivalent (the letter *B*) is assigned to the `myBigChar` variable, and the *B* that's in `myBigChar` prints onscreen.


CROSS REFERENCE

For an introduction to the Java application programming interface (API), see Chapter 3.

If you're tempted to write the following statement,

```
char myLittleChars = 'barry';  //Don't do this
```

LISTING 4-5: *(continued)*

```
        int elevatorWeightLimit = 1400;
        int numberOfPeople = elevatorWeightLimit / weightOfAPerson;

        boolean allTenOkay = numberOfPeople >= 10;

        System.out.println(allTenOkay);
    }
}
```

```
True or False?
You can fit all ten of the
Brickenchicker dectuplets
on the elevator:

false
```

FIGURE 4-9: The Brickenchicker dectuplets strike again.

In Listing 4-5, the `allTenOkay` variable is of type `boolean`. To find a value for the `allTenOkay` variable, the program checks to see whether `numberOfPeople` is greater than or equal to ten. (The symbols `>=` stand for *greater than or equal to.*)

At this point, it pays to be fussy about terminology. Any part of a Java program that has a value is an *expression.* If you write

```
weightOfAPerson = 150;
```

then `150` is an expression (an expression whose value is the quantity `150`). If you write

```
numberOfEggs = 2 + 2;
```

then `2 + 2` is an expression (because `2 + 2` has the value `4`). If you write

```
int numberOfPeople = elevatorWeightLimit / weightOfAPerson;
```

then `elevatorWeightLimit / weightOfAPerson` is an expression. (The value of the expression `elevatorWeightLimit / weightOfAPerson` depends on whatever values the variables `elevatorWeightLimit` and `weightOfAPerson` have when the code containing the expression is executed.)

REMEMBER

Any part of a Java program that has a value is an *expression.*

In Listing 4-5, the code `numberOfPeople >= 10` is an expression. The expression's value depends on the value stored in the `numberOfPeople` variable. But, as you know from seeing the strawberry shortcake at the Brickenchicker family's catered lunch, the value of `numberOfPeople` isn't greater than or equal to ten. As a result, the value of `numberOfPeople >= 10` is `false`. So, in the statement in Listing 4-5, in which `allTenOkay` is assigned a value, the `allTenOkay` variable is assigned a `false` value.

TIP

In Listing 4-5, I call `System.out.println()` with nothing inside the parentheses. When I do this, Java adds a line break to the program's output. In Listing 4-5, `System.out.println()` tells the program to display a blank line.

The Molecules and Compounds: Reference Types

By combining simple things, you get more complicated things. That's the way things always go. Take some of Java's primitive types, whip them together to make a primitive type stew, and what do you get? A more complicated type called a *reference type.*

The program in Listing 4-6 uses reference types. Figure 4-10 shows you what happens when you run the program in Listing 4-6.

LISTING 4-6: **Using Reference Types**

```
import javax.swing.JFrame;

public class ShowAFrame {

    public static void main(String args[]) {
        JFrame myFrame = new JFrame();
        String myTitle = "Blank Frame";

        myFrame.setTitle(myTitle);
        myFrame.setSize(300, 200);
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setVisible(true);
    }
}
```

The program in Listing 4-6 uses two references types. Both types are defined in the Java API. One of the types (the one that you'll use all the time) is called *String*. The other type (the one that you can use to create GUIs) is called *JFrame*.

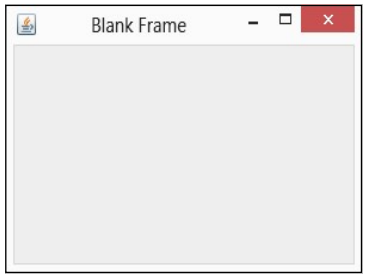

FIGURE 4-10: An empty frame.

A `String` is a bunch of characters. It's like having several `char` values in a row. So, with the `myTitle` variable declared to be of type `String`, assigning `"Blank Frame"` to the `myTitle` variable makes sense in Listing 4-6. The `String` class is declared in the Java API.

REMEMBER

In a Java program, double quote marks surround the letters in a `String` literal.

A Java *JFrame* is a lot like a window. (The only difference is that you call it a JFrame instead of a window.) To keep Listing 4-6 short and sweet, I decided not to put anything in my frame — no buttons, no fields, nothing.

Even with a completely empty frame, Listing 4-6 uses tricks that I don't describe until later in this book. So don't try reading and interpreting every word of Listing 4-6. The big thing to get from Listing 4-6 is that the program has two variable declarations. In writing the program, I made up two variable names: `myTitle` and `myFrame`. According to the declarations, `myTitle` is of type `String`, and `myFrame` is of type `JFrame`.

You can look up `String` and `JFrame` in Java's API documentation. But, even before you do, I can tell you what you'll find. You'll find that `String` and `JFrame` are the names of Java classes. So that's the big news. Every class is the name of a reference type. You can reserve `amountInAccount` for `double` values by writing

```
double amountInAccount;
```

or by writing

```
double amountInAccount = 50.22;
```


You can also reserve `myFrame` for a `JFrame` value by writing

```
JFrame myFrame;
```

or by writing

```
JFrame myFrame = new JFrame();
```

To review the notion of a Java class, see the sections on object-oriented programming (OOP) in Chapter 1.

REMEMBER

Every Java class is a reference type. If you declare a variable to have some type that's not a primitive type, the variable's type is (most of the time) the name of a Java class.

Now, when you declare a variable to have type `int`, you can visualize what that declaration means in a fairly straightforward way. It means that, somewhere inside the computer's memory, a storage location is reserved for that variable's value. In the storage location is a bunch of bits. The arrangement of the bits ensures that a certain whole number is represented.

That explanation is fine for primitive types like `int` or `double`, but what does it mean when you declare a variable to have a reference type? What does it mean to declare variable `myFrame` to be of type `JFrame`?

Well, what does it mean to declare *i thank You God* to be an E. E. Cummings poem? What would it mean to write the following declaration?

```
EECummingsPoem ithankYouGod;
```

It means that a class of things is `EECummingsPoem`, and `ithankYouGod` refers to an instance of that class. In other words, `ithankYouGod` is an object belonging to the `EECummingsPoem` class.

Because `JFrame` is a class, you can create objects from that class. (If you don't believe me, read some of my paragraphs about classes and objects in Chapter 1.) Each object (each instance of the `JFrame` class) is an actual frame — a window that appears on the screen when you run the code in Listing 4-6. By declaring the variable `myFrame` to be of type `JFrame`, you're reserving the use of the name `myFrame`. This reservation tells the computer that `myFrame` can refer to an actual `JFrame`-type object. In other words, `myFrame` can become a nickname for one of the windows that appears on the computer screen. Figure 4-11 illustrates the situation.

PRIMITIVE TYPE STEW

While I'm on the subject of frames, what's a frame, anyway? A *frame* is a window that has a certain height and width and a certain location on your computer's screen. Therefore, deep inside the declaration of the `Frame` class, you can find variable declarations that look something like this:

```
int width;
int height;
int x;
int y;
```

Here's another example — `Time`. An instance of the `Time` class may have an hour (a number from 1 to 12), a number of minutes (from 0 to 59), and a letter (*a* for a.m.; *p* for p.m.).

```
int hour;
int minutes;
char amOrPm;
```

Notice that this high-and-mighty thing called a Java API class is neither high nor mighty. A class is just a collection of declarations. Some of those declarations are the declarations of variables. Some of those variable declarations use primitive types, and other variable declarations use reference types. These reference types, however, come from other classes, and the declarations of those classes have variables. The chain goes on and on. Ultimately, everything comes, in one way or another, from the primitive types.

An Import Declaration

It's always good to announce your intentions up front. Consider the following classroom lecture:

> *Today, in our History of Film course, we'll be discussing the career of actor* ***Lionel Herbert Blythe Barrymore****.*
>
> *Born in Philadelphia,* ***Barrymore*** *appeared in more than 200 films, including* It's a Wonderful Life, Key Largo, and Dr. Kildare's Wedding Day. *In addition,* ***Barrymore*** *was a writer, composer, and director. Barrymore did the voice of Ebenezer Scrooge every year on radio . . . .*

REMEMBER

When you declare *`ClassName variableName;`*, you're saying that a certain variable can refer to an instance of a particular class.

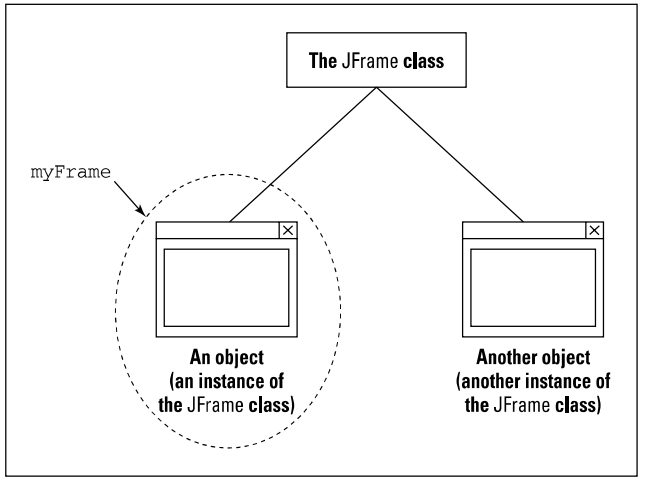

FIGURE 4-11: The variable `myFrame` refers to an instance of the `JFrame` class.

TECHNICAL STUFF

In Listing 4-6, the phrase `JFrame myFrame` reserves the use of the name `myFrame`. On that same line of code, the phrase `new JFrame()` creates a new object (an instance of the `JFrame` class). Finally, that line's equal sign makes `myFrame` refer to the new object. Knowing that the two words `new JFrame()` create an object can be very important. For a more thorough explanation of objects, see Chapter 7.

TRY IT OUT

Try these things:

- Run the code in Listing 4-6 on your computer.
- Before running the code in Listing 4-6, comment out the `myFrame.setVisible(true)` statement by putting two forward slashes (`//`) immediately to the left of the statement. Does anything happen when you run the modified code?
- Experiment with the code in Listing 4-6 by changing the order of the statements inside the body of the `main` method. What rearrangements of these statements are okay, and which aren't?

The details of this import stuff can be pretty nasty. But fortunately, many IDEs have convenient helper features for import declarations. For details, see this book's website (`www.allmycode.com/JavaForDummies`).

No single section in this book can present the entire story about import declarations. To begin untangling some of the import declaration's subtleties, see Chapters 5, 9, and 10.

Creating New Values by Applying Operators

What could be more comforting than your old friend the plus sign? It was the first topic you learned about in elementary school math. Almost everybody knows how to add 2 and 2. In fact, in English usage, adding 2 and 2 is a metaphor for something that's easy to do. Whenever you see a plus sign, a cell in your brain says, "Thank goodness — it could be something much more complicated."

Java has a plus sign. You can use it for several purposes. You can use the plus sign to add two numbers, like this:

```
int apples, oranges, fruit;
apples = 5;
oranges = 16;
fruit = apples + oranges;
```

You can also use the plus sign to paste `String` values together:

```
String startOfChapter =
    "It's three in the morning. I'm dreaming about the"+
    "history course that I failed in high school.";
System.out.println(startOfChapter);
```

This can be handy because in Java, you're not allowed to make a `String` straddle from one line to another. In other words, the following code wouldn't work:

```
String thisIsBadCode =
    "It's three in the morning. I'm dreaming about the
     history course that I failed in high school.";
System.out.println(thisIsBadCode);
```


Interesting stuff, heh? Now compare these paragraphs with a lecture in which the instructor doesn't begin by introducing the subject:

> *Welcome once again to the History of Film.*
>
> *Born in Philadelphia,* ***Lionel Barrymore*** *appeared in more than 200 films, including* It's a Wonderful Life, Key Largo, *and* Dr. Kildare's Wedding Day. *In addition,* ***Barrymore (not Ethel, John, or Drew)*** *was a writer, composer, and director.* ***Lionel Barrymore*** *did the voice of Ebenezer Scrooge every year on radio . . . .*

Without a proper introduction, a speaker may have to remind you constantly that the discussion is about Lionel Barrymore and not about any other Barrymore. The same is true in a Java program. Look again at Listing 4-6:

```
import javax.swing.JFrame;

public class ShowAFrame {

    public static void main(String args[]) {
        JFrame myFrame = new JFrame();
```

In Listing 4-6, you announce in the introduction (in the import declaration) that you're using `JFrame` in your Java class. You clarify what you mean by `JFrame` with the full name `javax.swing.JFrame`. (Hey! Didn't the first lecturer clarify with the full name "Lionel Herbert Blythe Barrymore"?) After announcing your intentions in the import declaration, you can use the abbreviated name *JFrame* in your Java class code.

If you don't use an import declaration, you have to repeat the full `javax.swing.JFrame` name wherever you use the name *JFrame* in your code. For example, without an import declaration, the code of Listing 4-6 would look like this:

```
public class ShowAFrame {

    public static void main(String args[]) {
        javax.swing.JFrame myFrame = new javax.swing.JFrame();
        String myTitle = "Blank Frame";

        myFrame.setTitle(myTitle);
        myFrame.setSize(3200, 200);
        myFrame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
        myFrame.setVisible(true);
    }
}
```

LISTING 4-7: Making Change

```
import static java.lang.System.out;

public class MakeChange {

    public static void main(String args[]) {
        int total = 248;
        int quarters = total / 25;
        int whatsLeft = total % 25;

        int dimes = whatsLeft / 10;
        whatsLeft = whatsLeft % 10;

        int nickels = whatsLeft / 5;
        whatsLeft = whatsLeft % 5;

        int cents = whatsLeft;

        out.println("From " + total + " cents you get");
        out.println(quarters + " quarters");
        out.println(dimes + " dimes");
        out.println(nickels + " nickels");
        out.println(cents + " cents");
    }
}
```

Figure 4-12 shows a run of the code in Listing 4-7. You start with a total of 248 cents. Then

```
quarters = total / 25
```

divides 248 by 25, giving 9. That means you can make 9 quarters from 248 cents. Next,

```
whatsLeft = total % 25
```

divides 248 by 25 again and puts only the remainder, 23, into `whatsLeft`. Now you're ready for the next step, which is to take as many dimes as you can out of cents.

```
From 248 cents you get
9 quarters
2 dimes
0 nickels
3 cents
```

FIGURE 4-12: Change for $2.48.

TECHNICAL STUFF

The correct way to say that you're pasting `String` values together is to say that you're *concatenating* `String` values.

You can even use the plus sign to paste numbers next to `String` values:

```
int apples, oranges, fruit;
apples = 5;
oranges = 16;
fruit = apples + oranges;
System.out.println("You have" + fruit + "pieces of fruit.");
```

Of course, the old minus sign is available, too (but not for `String` values):

```
apples = fruit - oranges;
```

Use an asterisk (`*`) for multiplication and a slash (`/`) for division:

```
double rate, pay;
int hours;

rate = 6.25;
hours = 35;
pay = rate * hours;
System.out.println(pay);
```

For an example using division, refer to Listing 4-3.

WARNING

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`. To get past this, make either (or both) of the numbers you're dividing `double` values. If you put `System.out.println(11.0 / 4)` in your program, the computer prints `2.75`.

Another useful arithmetic operator is 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 this because 4 goes into 11 who-cares-how-many times with a remainder of 3. The remainder operator turns out to be fairly useful. Listing 4-7 has an example.

TRY IT OUT

Find the values of the following expressions by typing each expression in JShell (if you have trouble launching JShell, create a Java program that displays the value of each of these expressions):

```
» 5 / 4
» 5 / 4.0
» 5.0 / 4
» 5.0 / 4.0
» "5" + "4"
» 5 + 4
» " " + 5 + 4
```

Initialize once, assign often

Listing 4-7 has three lines that put values into the variable `whatsLeft`:

```
int whatsLeft = total % 25;

whatsLeft = whatsLeft % 10;

whatsLeft = whatsLeft % 5;
```

Only one of these lines is a declaration. The other two lines are assignment statements. That's good because you can't declare the same variable more than once (not without creating something called a *block*). If you goof and write

```
int whatsLeft = total % 25;

int whatsLeft = whatsLeft % 10;
```

in Listing 4-7, you see an error message (such as `Duplicate variable whatsLeft` or `Variable 'whatsLeft' is already defined`) when you try to compile your code.

CROSS REFERENCE

To find out what a block is, see Chapter 5. Then, for some honest talk about redeclaring variables, see Chapter 10.

TECHNICAL STUFF

The code in Listing 4-7 makes change in U.S. 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` class gives you more than simply a set of coins adding up to 248 cents. The `MakeChange` class gives you the *smallest number of coins* that add up to 248 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. In fact, I'm looking for examples. If your country's coinage prevents `MakeChange` from always giving the best answer, please, send me an email (`JavaForDummies@allmycode.com`).

IMPORT DECLARATIONS: THE UGLY TRUTH

Notice the import declaration at the top of Listing 4-7:

```
import static java.lang.System.out;
```

Compare this with the import declaration at the top of Listing 4-6:

```
import javax.swing.JFrame;
```

By adding the `import static java.lang.System.out;` line to Listing 4-7, I can make the rest of the code a bit easier to read, and I can avoid having long Java statements that start on one line and continue on another. But you never have to do that. If you remove the `import static java.lang.System.out;` line and pepper the code liberally with `System.out.println`, the code works just fine.

Here's a question: Why does one declaration include the word *static* and the other declaration doesn't? Well, to be honest, I wish I hadn't asked!

For the real story about *static*, you have to read part of Chapter 10. And frankly, I don't recommend skipping ahead to that chapter's *static* section if you take medicine for a heart condition, if you're pregnant or nursing, or if you have no previous experience with object-oriented programming. For now, rest assured that Chapter 10 is easy to read after you've made the journey through Part 3 of this book. And when you have to decide whether to use the word *static* in an import declaration, remember these hints:

- The vast majority of import declarations in Java program do not use the word *static*.
- In this book, I never use *import static* to import anything except `System.out`. (Well, almost never . . . .)
- Most import declarations don't use the word *static* because most declarations import classes. Unfortunately, `System.out` is not the name of a class.

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, four such operators exist — two increment operators and two decrement operators. The increment operators add 1, and the decrement operators subtract 1. The increment operators use double plus signs (++), and the decrement operators use double minus signs (--). To see how they work, you need some examples. The first example is shown in Figure 4-13.

Figure 4-14 shows a run of the program in Figure 4-13. In this horribly uneventful run, the count of bunnies prints three times.

The double plus signs go by two names, depending on where you put them. When you put the ++ before a variable, the ++ is called the *preincrement* operator. (The *pre* stands for *before.*)

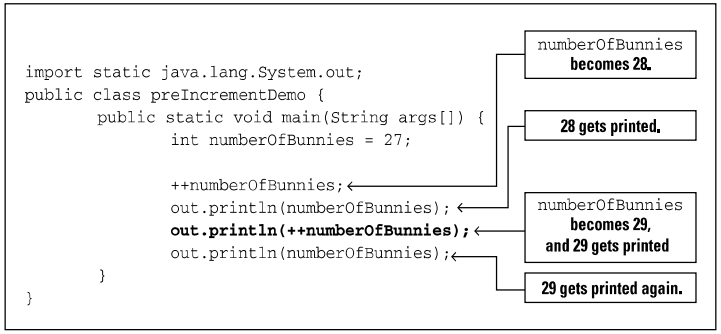

FIGURE 4-13: Using preincrement.

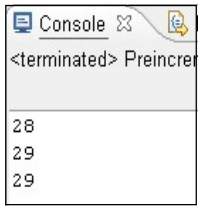

FIGURE 4-14: A run of the code in Figure 4-13.

The word *before* has two meanings:

- You put ++ before the variable.
- The computer adds 1 to the variable's value before the variable is used in any other part of the statement.

To understand this, look at the bold line in Figure 4-13. The computer adds 1 to `numberOfBunnies` (raising the value of `numberOfBunnies` to 29) and then prints 29 onscreen.

REMEMBER

With `out.println(++numberOfBunnies)`, the computer adds 1 to `numberOfBunnies` before printing the new value of `numberOfBunnies` onscreen.

An alternative to preincrement is *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.

To see more clearly how postincrement works, look at the bold line in Figure 4-15. The computer prints the old value of `numberOfBunnies` (which is 28) on the screen, and then the computer adds 1 to `numberOfBunnies`, which raises the value of `numberOfBunnies` to 29.

REMEMBER

With `out.println(numberOfBunnies++)`, the computer adds 1 to `numberOfBunnies` after printing the old value that `numberOfBunnies` already had.

Figure 4-16 shows a run of the code in Figure 4-15. Compare Figure 4-16 with the run in Figure 4-14:

- With preincrement in Figure 4-14, the second number is 29.
- With postincrement in Figure 4-16, the second number is 28.

  In Figure 4-16, 29 doesn't show onscreen until the end of the run, when the computer executes one last `out.println(numberOfBunnies)`.

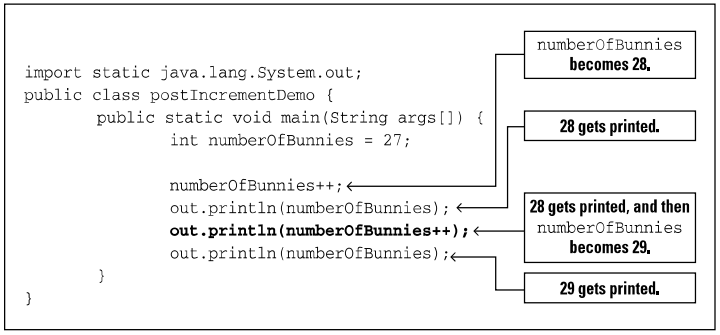

FIGURE 4-15: Using postincrement.

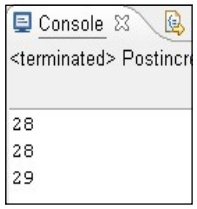

FIGURE 4-16: A run of the code in Figure 4-15.

TIP

Are you trying to decide between using preincrement or postincrement? Try no longer. Most programmers use postincrement. In a typical Java program, you often see things like `numberOfBunnies++`. You seldom see things like `++numberOfBunnies`.

In addition to preincrement and postincrement, Java has two operators that use `--`. These operators are called *predecrement* and *postdecrement:*

- With predecrement (`--numberOfBunnies`), the computer subtracts 1 from the variable's value before the variable is used in the rest of the statement.
- With postdecrement (`numberOfBunnies--`), the computer subtracts 1 from the variable's value after the variable is used in the rest of the statement.

TECHNICAL STUFF

Instead of writing `++numberOfBunnies`, you could achieve the same effect by writing `numberOfBunnies = numberOfBunnies + 1`. So some people conclude that Java's `++` and `--` operators are for saving keystrokes — to keep those poor fingers from overworking themselves. This is entirely incorrect. The best reason for using `++` is to avoid the inefficient and error-prone practice of writing the same variable name, such as `numberOfBunnies`, twice in the same statement. If you write `numberOfBunnies` only once (as you do when you use `++` or `--`), the computer has to figure out what `numberOfBunnies` means only once. On top of that, when you write `numberOfBunnies` only once, you have only one chance (instead of two chances) to type the variable name incorrectly. With simple expressions like `numberOfBunnies++`, these advantages hardly make a difference. But with more complicated expressions, such as `inventoryItems[(quantityReceived--*itemsPerBox+17)]++`, the efficiency and accuracy that you gain by using `++` and `--` are significant.

STATEMENTS AND EXPRESSIONS

You can describe the pre- and postincrement and pre- and postdecrement operators in two ways: the way everyone understands them and the right way. The way that I explain the concept in most of this section (in terms of time, with *before* and *after*) is the way that everyone understands it. 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 a 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, 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. (I discuss statements in Chapter 3, and I describe expressions elsewhere in this chapter.) Which category does `numberOfBunnies++` belong to? The surprising answer is both. The Java code `numberOfBunnies++` is both a statement and an expression.

Assume that, before the computer executes the code `out.println(numberOfBunnies++)`, the value of `numberOfBunnies` is 28.

- As a statement, `numberOfBunnies++` tells the computer to add 1 to `numberOfBunnies`.
- As an expression, the value of `numberOfBunnies++` is 28, not 29.

So even though the computer adds 1 to `numberOfBunnies`, the code `out.println(numberOfBunnies++)` really means `out.println(28)`.

Now, almost everything you just read about `numberOfBunnies++` is true about `++numberOfBunnies`. The only difference is that as an expression, `++numberOfBunnies` behaves in a more intuitive way.

- As a statement, `++numberOfBunnies` tells the computer to add 1 to `numberOfBunnies`.
- As an expression, the value of `++numberOfBunnies` is 29.

So, with `out.println(++numberOfBunnies)`, the computer adds 1 to the variable `numberOfBunnies`, and the code `out.println(++numberOfBunnies)` really means `out.println(29)`.

by anything you want. You can do other cool operations, too. Listing 4-8 has a smorgasbord of assignment operators (the things with equal signs). Figure 4-17 shows the output from running Listing 4-8.

LISTING 4-8: Assignment Operators

```
public class UseAssignmentOperators {

    public static void main(String args[]) {
        int numberOfBunnies = 27;
        int numberExtra = 53;

        numberOfBunnies += 1;
        System.out.println(numberOfBunnies);

        numberOfBunnies += 5;
        System.out.println(numberOfBunnies);

        numberOfBunnies += numberExtra;
        System.out.println(numberOfBunnies);

        numberOfBunnies *= 2;
        System.out.println(numberOfBunnies);

        System.out.println(numberOfBunnies -= 7);

        System.out.println(numberOfBunnies = 100);
    }
}
```

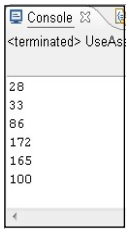

FIGURE 4-17: A run of the code in Listing 4-8.

Listing 4-8 shows how versatile Java's assignment operators are. With the assignment operators, you can add, subtract, multiply, or divide a variable by any number. Notice how `+= 5` adds 5 to `numberOfBunnies`, and how `*= 2` multiplies `numberOfBunnies` by 2. You can even use another expression's value (in Listing 4-8, `numberExtra`) as the number to be applied.

TRY IT OUT

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) {
    int i = 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);
    i++;
    i = i++ + ++i;
    System.out.println(i);
    i = i++ + i++;
    System.out.println(i);
  }
}
```

Type the boldface text, one line after another, into JShell, and see how JShell responds.

```
jshell> int i = 8
jshell> i++
jshell> i
jshell> i
jshell> i++
jshell> i
jshell> ++i
jshell> i
```

Assignment operators

If you read the preceding section, which is 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 `numberOfBunnies++++` and still call yourself a Java programmer? Well, you can't. If you try it, an error message appears when you try to compile your code.

What can you do? 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

IN THIS CHAPTER

- Writing statements that choose between alternatives
- Forming logical conditions
- Putting statements inside one another
- Choosing among many alternatives

Chapter 5

Controlling Program Flow with Decision-Making Statements

The TV show *Dennis the Menace* aired on CBS from 1959 to 1963. I remember one episode in which Mr. Wilson was having trouble making an important decision. I think it was something about changing jobs or moving to a new town. Anyway, I can still see that shot of Mr. Wilson sitting in his yard, sipping lemonade, and staring into nowhere for the whole afternoon. Of course, the annoying character Dennis was constantly interrupting Mr. Wilson's peace and quiet. That's what made this situation funny.

What impressed me about this episode (the reason I remember it clearly even now) was Mr. Wilson's dogged intent in making the decision. This guy wasn't going about his everyday business, roaming around the neighborhood while thoughts about the decision wandered in and out of his mind. He was sitting quietly in his yard, making marks carefully and logically on his mental balance sheet. How many people actually make decisions this way?

At that time, I was still pretty young. I'd never faced the responsibility of making a big decision that affected my family and me. But I wondered what such a decision-making process would be like. Would it help to sit there like a stump for

The last two lines in Listing 4-8 demonstrate a special feature of Java's assignment operators. You can use an assignment operator as part of a larger Java statement. In the next-to-last line of Listing 4-8, the operator subtracts 7 from `numberOfBunnies`, decreasing the value of `numberOfBunnies` from `172` to `165`. Then the whole assignment business is stuffed into a call to `System.out.println`, so 165 prints onscreen.

Lo and behold, the last line of Listing 4-8 shows how you can do the same thing with Java's plain old equal sign. The thing that I call an assignment statement near the start of this chapter is really one of the assignment operators that I describe in this section. Therefore, whenever you assign a value to something, you can make that assignment be part of a larger statement.


TECHNICAL STUFF

Each use of an assignment operator does double duty as a statement and an expression. In all cases, the expression's value equals whatever value you assign. For example, before executing the code `System.out.println(numberOfBunnies -= 7)`, the value of `numberOfBunnies` is `172`. As a statement, `numberOfBunnies -= 7` tells the computer to subtract 7 from `numberOfBunnies` (so the value of `numberOfBunnies` goes from `172` to `165`). As an expression, the value of `numberOfBunnies -= 7` is `165`. So the code `System.out.println(numberOfBunnies -= 7)` really means `System.out.println(165)`. The number 165 displays on the computer screen.

For a richer explanation of this kind of thing, see the sidebar "Statements and expressions," earlier in this chapter.


TRY IT OUT

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) {
    int i = 10;

    i += 2;
    i -= 5;
    i *= 6;

    System.out.println(i);
    System.out.println(i += 3);
    System.out.println(i /= 2);
  }
}
```

```
        out.println("Thank you for playing.");

        keyboard.close();
    }
}
```

```
Enter an int from 1 to 10: 2
**********
*You win.*
**********
Thank you for playing.


Enter an int from 1 to 10: 4
You lose.
The random number was 10.
Thank you for playing.
```

FIGURE 5-1: Two runs of the guessing game.

The program in Listing 5-1 plays a guessing game with the user. The program gets a number (a guess) from the user and then generates a random number between 1 and 10. If the number that the user entered is the same as the random number, the user wins. Otherwise, the user loses and the program tells the user what the random number was.

She controlled keystrokes from the keyboard

Taken together, the lines

```
import java.util.Scanner;

        Scanner keyboard = new Scanner(System.in);

        int inputNumber = keyboard.nextInt();
```

in Listing 5-1 get whatever number the user types on the computer's keyboard. The last of the three lines puts this number into a variable named *inputNumber*. If these lines look complicated, don't worry: You can copy these lines almost word for word whenever you want to read from the keyboard. Include the first two lines (the `import` and `Scanner` lines) just once in your program. Later in your program, wherever the user types an `int` value, include a line with a call to `nextInt` (as in the last of the preceding three lines of code).

hours on end? Would I make my decisions by the careful weighing and tallying of options? Or would I shoot in the dark, take risks, and act on impulse? Only time would tell.

Making Decisions (Java if Statements)

When you're writing computer programs, you're constantly hitting forks in roads. Did the user correctly type the password? If yes, let the user work; if no, kick the bum out. So the Java programming language needs a way of making a program branch in one of two directions. Fortunately, the language has a way: It's called an `if` statement.

Guess the number

Listing 5-1 illustrates the use of an `if` statement. Two runs of the program in Listing 5-1 are shown in Figure 5-1.

LISTING 5-1: A Guessing Game

```
import static java.lang.System.out;
import java.util.Scanner;
import java.util.Random;

public class GuessingGame {

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

        out.print("Enter an int from 1 to 10: ");

        int inputNumber = keyboard.nextInt();
        int randomNumber = new Random().nextInt(10) + 1;

        if (inputNumber == randomNumber) {
            out.println("**********");
            out.println("*You win.*");
            out.println("**********");
        } else {
            out.println("You lose.");
            out.print("The random number was ");
            out.println(randomNumber + ".");
        }
```

Of all the names in these three lines of code, the only two names that I coined myself are *inputNumber* and *keyboard*. All the other names are part of Java. So, if I want to be creative, I can write the lines this way:

```
import java.util.Scanner;

        Scanner readingThingie = new Scanner(System.in);

        int valueTypedIn = readingThingie.nextInt();
```

I can also beef up my program's import declarations, as I do later on in Listings 5-2 and 5-3. Other than that, I have very little leeway.

As you read on in this book, you'll start recognizing the patterns behind these three lines of code, so I don't clutter up this section with all the details. For now, you can just copy these three lines and keep the following in mind:

- **When you import `java.util.Scanner`, you don't use the word *static*.**

  But importing `Scanner` is different from importing `System.out`. When you import `java.lang.System.out`, you use the word `static`. (Refer to Listing 5-1.) The difference creeps into the code because `Scanner` is the name of a class, and `System.out` isn't the name of a class.

  CROSS REFERENCE

  For a quick look at the use of the word *static* in import declarations, see the sidebar in Chapter 4 about import declarations: the ugly truth. For a more complete story about the word, see Chapter 10.

- **Typically (on a desktop or laptop computer), the name *System.in* stands for the keyboard.**

  To get characters from some place other than the keyboard, you can type something other than `System.in` inside the parentheses.

  CROSS REFERENCE

  What else can you put inside the `new Scanner(...)` parentheses? For some ideas, see Chapter 8.

  In Listing 5-1, I make the arbitrary decision to give one of my variables the name `keyboard`. The name `keyboard` reminds you, the reader, that this variable refers to a bunch of plastic buttons in front of your computer. Naming something `keyboard` doesn't tell Java anything about plastic buttons or about user input. On the other hand, the name `System.in` always tells Java about those plastic buttons. The code `Scanner keyboard = new Scanner(System.in)` in Listing 5-1 connects the name `keyboard` with the plastic buttons that we all know and love.

- **When you expect the user to type an `int` value (a whole number of some kind), use `nextInt()`.**

  If you expect the user to type a `double` value (a number containing a decimal point), use `nextDouble()`. If you expect the user to type **true** or **false**, use `nextBoolean()`. If you expect the user to type a word like *Barry*, *Java*, or *Hello*, use `next()`.

  WARNING

  Decimal points vary from one country to another. In the United States, *10.5* (with a period) represents ten-and-a-half, but in France, *10,5* (with a comma) represents ten-and-a-half. In the Persian language, a decimal point looks like a slash (but it sits a bit lower than the digit characters). Your computer's operating system stores information about the country you live in, and Java reads that information to decide what ten-and-a-half looks like. If you run a program containing a `nextDouble()` method call, and Java responds with an `InputMismatchException`, check your input. You might have input 10.5 when your country's conventions require 10,5 (or another way of representing ten-and-a-half). For more information, see the "Where on Earth do you live?" sidebar in Chapter 8.

  CROSS REFERENCE

  For an example in which the user types a word, see Listing 5-3, later in this chapter. For an example in which the user types a single character, see Listing 6-4, in Chapter 6. For an example in which a program reads an entire line of text (all in one big gulp), see Chapter 8.

- **You can get several values from the keyboard, one after another.**

  To do this, use the `keyboard.nextInt()` code several times.

  To see a program that reads more than one value from the keyboard, go to Listing 5-4, later in this chapter.

- Whenever you use Java's `Scanner`, you should call the `close` method after your last `nextInt` call (or your last `nextDouble` call, or your last `next`*`Whatever`* call).

  In Listing 5-1, the `main` method's last statement is

  ```
  keyboard.close();
  ```

  This statement does some housekeeping to disconnect the Java program from the computer keyboard. (The amount of required housekeeping is more than you might think!) If I omit this statement from Listing 5-1, nothing terrible happens. Java's Virtual Machine usually cleans up after itself very nicely. But using `close()` to explicitly detach from the keyboard is good practice, and some IDEs display warnings if you omit the `keyboard.close()` statement. In this book's example, I always remember to close my `Scanner` variables.

In Chapter 13, I show you a more reliable way to incorporate the `keyboard.close()` statement in your Java program.

When your program calls `System.out.println`, your program uses the computer's screen. So why don't you call a `close` method after all your `System.out.println` calls? The answer is subtle. In Listing 5-1, your own code connects to the keyboard by calling `new Scanner(System.in)`. So, later in the program, your code cleans up after itself by calling the `close` method. But with `System.out.println`, your own code doesn't create a connection to the screen. (The `out` variable refers to a `PrintStream`, but you don't call `new PrintStream()` to prepare for calling `System.out.println`.) Instead, the Java Virtual Machine connects to the screen on your behalf. The Java Virtual Machine's code (which you never have to see) contains a call to `new PrintStream()` in preparation for your calling `System.out.println`. So, because it's a well-behaved piece of code, the Java Virtual Machine eventually calls `out.close()` without any effort on your part.

Creating randomness

Achieving real randomness is surprisingly difficult. Mathematician Persi Diaconis says that if you flip a coin several times, always starting with the head side up, you're likely to toss heads more often than tails. If you toss several more times, always starting with the tail side up, you'll likely toss tails more often than heads. In other words, coin tossing isn't really fair.*

Computers aren't much better than coins and human thumbs. A computer mimics the generation of random sequences, but in the end the computer just does what it's told and does all of this in a purely deterministic fashion. So in Listing 5-1, when the computer executes

```
import java.util.Random;

        int randomNumber = new Random().nextInt(10) + 1;
```

the computer appears to give a randomly generated number — a whole number between 1 and 10. But it's all a fake. The computer only follows instructions. It's not really random, but without bending a computer over backward, it's the best that anyone can do.

*Diaconis, Persi. "The Search for Randomness." American Association for the Advancement of Science annual meeting. Seattle. 14 Feb. 2004.

Once again, I ask you to take this code on blind faith. Don't worry about what new `Random().nextInt` means until you have more experience with Java. Just copy this code into your own programs and have fun with it. And if the numbers from 1 to 10 aren't in your flight plans, don't fret. To roll an imaginary die, write the statement

```
int rollEmBaby = new Random().nextInt(6) + 1;
```

With the execution of this statement, the variable `rollEmBaby` gets a value from 1 to 6.

The if statement

At the core of Listing 5-1 is a Java `if` statement. This `if` statement represents a fork in the road. (See Figure 5-2.) The computer follows one of two prongs — the prong that prints `You win` or the prong that prints `You lose`. The computer decides which prong to take by testing the truth or falsehood of a *condition*. In Listing 5-1, the condition being tested is

```
inputNumber == randomNumber
```

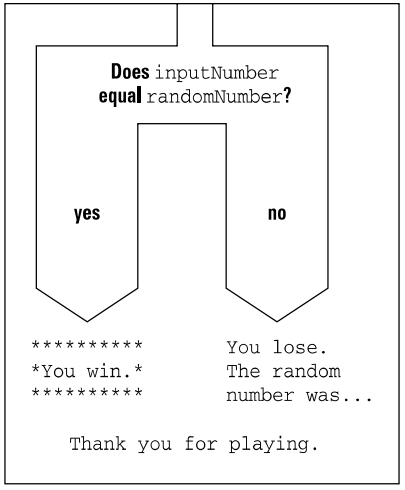

FIGURE 5-2: An `if` statement is like a fork in the road.

Does the value of `inputNumber` equal the value of `randomNumber`? When the condition is true, the computer does the stuff between the condition and the word *else*. When the condition turns out to be false, the computer does the stuff after the

word *else*. Either way, the computer goes on to execute the last `println` call, which displays `Thank you for playing`.


REMEMBER

The condition in an `if` statement must be enclosed in parentheses. However, a line like `if (inputNumber == randomNumber)` is not a complete statement (just as "If I had a hammer" isn't a complete sentence). So this line `if (inputNumber == randomNumber)` shouldn't end with a semicolon.


TECHNICAL STUFF

Sometimes, when I'm writing about a condition that's being tested, I slip into using the word *expression* instead of *condition*. That's okay because every condition is an expression. An expression is something that has a value and, sure enough, every condition has a value. The condition's value is either `true` or `false`. (For revealing information about expressions and values like `true` and `false`, see Chapter 4.)

The double equal sign

In Listing 5-1, in the `if` statement's condition, notice the use of the double equal sign. Comparing two numbers to see whether they're the same isn't the same as setting something equal to something else. That's why the symbol to compare for equality isn't the same as the symbol that's used in an assignment or an initialization. In an `if` statement's condition, you can't replace the double equal sign with a single equal sign. If you do, your program just won't work. (You almost always get an error message when you try to compile your code.)

On the other hand, if you never make the mistake of using a single equal sign in a condition, you're not normal. Not long ago, while I was teaching an introductory Java course, I promised that I'd swallow my laser pointer if no one made the single equal sign mistake during any of the lab sessions. This wasn't an idle promise. I knew I'd never have to keep it. As it turned out, even if I had ignored the first ten times anybody made the single equal sign mistake during those lab sessions, I would still be laser-pointer-free. Everybody mistakenly uses the single equal sign several times in a programming career.


TIP

The trick is not to avoid making the single-equal-sign mistake; the trick is to catch the mistake whenever you make it.

Brace yourself

The `if` statement in Listing 5-1 has two halves: a top half and a bottom half. I have names for these two parts of an `if` statement. I call them the *if part* (the top half) and the *else part* (the bottom half).

The `if` part in Listing 5-1 seems to have more than one statement in it. I make this happen by enclosing the three statements of the `if` part in a pair of curly braces. When I do this, I form a block. A *block* is a bunch of statements scrunched together by a pair of curly braces.

With this block, three calls to `println` are tucked away safely inside the `if` part. With the curly braces, the rows of asterisks and the words `You win` display only when the user's guess is correct.

This business with blocks and curly braces applies to the else part as well. In Listing 5-1, whenever `inputNumber` doesn't equal `randomNumber`, the computer executes three `print/println` calls. To convince the computer that all three of these calls are inside the `else` clause, I put these calls into a block. That is, I enclose these three calls in a pair of curly braces.


TECHNICAL STUFF

Strictly speaking, Listing 5-1 has only one statement between the `if` and the `else` statements and only one statement after the `else` statement. The trick is that when you place a bunch of statements inside curly braces, you get a block; and a block behaves, in all respects, like a single statement. In fact, the official Java documentation lists blocks as one of the many kinds of statements. So, in Listing 5-1, the block that prints `You win` and asterisks is a single statement that has, within it, three smaller statements.

Indenting if statements in your code

Notice how, in Listing 5-1, the `print` and `println` calls inside the `if` statement are indented. (This includes both the `You win` and `You lose` statements. The `print` and `println` calls that come after the word `else` are still part of the `if` statement.) 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 that neither you nor anyone else can make sense of your code if you don't indent your statements in some logical fashion. In Listing 5-1, the indenting of the `print` and `println` statements helps your eye (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.

Many Java IDEs have tools to indent your code automatically. In fact, code indentation is one of my favorite IDE features. So don't walk — run — to a computer, and visit this book's website (`www.allmycode.com/JavaForDummies`) for more information on what Java IDEs can offer.

The `if` statement in Listing 5-2 has no `else` part. When `inputNumber` is the same as `randomNumber`, the computer prints `You win`. When `inputNumber` is different from `randomNumber`, the computer doesn't print `You win`.

Listing 5-2 illustrates another new idea. With an `import` declaration for `System.in`, I can reduce `new Scanner(System.in)` to the shorter new `Scanner(in)`. Adding this `import` declaration is hardly worth the effort. In fact, I do more typing with the `import` declaration than without it. Nevertheless, the code in Listing 5-2 demonstrates that it's possible to import `System.in`.

```
Enter an int from 1 to 10: 4
*You win.*
That was a very good guess :-)
The random number was 4.
Thank you for playing.


Enter an int from 1 to 10: 4
That was a very good guess :-)
The random number was 6.
Thank you for playing.
```

FIGURE 5-3: Two runs of the game in Listing 5-2.

TRY IT OUT

In Chapter 4, Listing 4-5 tells you whether you can or cannot fit ten people on an elevator. A run of the listing's code looks something like this:

```
True or False?
You can fit all ten of the
Brickenchicker dectuplets
on the elevator:

false
```

Use what you know about Java's `if` statements to make the program's output more natural. Depending on the value of the program's `elevatorWeightLimit` variable, the output should be either

```
You can fit all ten of the
Brickenchicker dectuplets
on the elevator.
```

or

```
You can't fit all ten of the
Brickenchicker dectuplets
on the elevator.
```


WARNING

When you write `if` statements, you may be tempted to chuck out the window all the rules about curly braces and simply rely on indentation. This works in other programming languages, such as Python and Haskell, but it doesn't work in Java. If you indent three statements after the word `else` and forget to enclose those statements in curly braces, the computer thinks that the `else` part includes only the first of the three statements. What's worse, the indentation misleads you into believing that the `else` part includes all three statements. This makes it more difficult for you to figure out why your code isn't behaving the way you think it should. Watch those braces!

Elseless in Ifrica

Okay, so the title of this section is contrived. Big deal! The idea is that you can create an `if` statement without the `else` part. Take, for instance, the code in Listing 5-1, shown earlier. Maybe you'd rather not rub it in whenever the user loses the game. The modified code in Listing 5-2 shows you how to do this (and Figure 5-3 shows you the result).

LISTING 5-2: A Kinder, Gentler Guessing Game

```
import static java.lang.System.in;
import static java.lang.System.out;
import java.util.Scanner;
import java.util.Random;

public class DontTellThemTheyLost {

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

        out.print("Enter an int from 1 to 10: ");

        int inputNumber = keyboard.nextInt();
        int randomNumber = new Random().nextInt(10) + 1;

        if (inputNumber == randomNumber) {
            out.println("*You win.*");
        }

        out.println("That was a very good guess :-)");
        out.print("The random number was ");
        out.println(randomNumber + ".");
        out.println("Thank you for playing.");

        keyboard.close();
    }
}
```

Using Blocks in JShell

Chapter 4 introduces Java 9's interactive JShell environment. You type a statement, and JShell responds immediately by executing the statement. That's fine for simple statements, but what happens when you have a statement inside of a block?

In JShell, you can start typing a statement with one or more blocks. JShell doesn't respond until you finish typing the entire statement — blocks and all. To see how it works, look over this conversation that I had recently with JShell:

```
jshell> import static java.lang.System.out

jshell> import java.util.Random

jshell> int randomNumber = new Random().nextInt(10) + 1
randomNumber ==> 4

jshell> int inputNumber = 4
inputNumber ==> 4

jshell> if (inputNumber == randomNumber) {
   ...>     out.println("*You win.*");
   ...> }
*You win.*

jshell>
```

In this dialogue, I've set the text that I type in bold. JShell's responses aren't set in bold.

When I type `if (inputNumber == randomNumber) {` and press Enter, JShell doesn't do much. JShell only displays a `...>` prompt, which indicates that whatever lines I've typed don't form a complete statement. I have to respond by typing the rest of the `if` statement.

When I finish the `if` statement with a close curly brace, JShell finally acknowledges that I've typed an entire statement. JShell executes the statement and (in this example) displays `*You win.*`.

Notice the semicolon at the end of the `out.println` line:

» When you type a statement that's not inside of a block, JShell lets you omit the semicolon at the end of the statement.

» When you type a statement that's inside of a block, JShell (like the plain old Java in Listing 5-2) doesn't let you omit the semicolon.


TIP

When you type a block in JShell, you always have the option of typing the entire block on one line, with no line breaks, like so:

```
if (inputNumber == randomNumber) { out.println("*You win.*"); }
```

Forming Conditions with Comparisons and Logical Operators

The Java programming language has plenty of little squiggles and doodads for your various condition-forming needs. This section tells you all about them.

Comparing numbers; comparing characters

Table 5-1 shows you the operators that you can use to compare one value with another.

TABLE 5-1 Comparison Operators

Operator Symbol	Meaning	Example
`==`	is equal to	`numberOfCows == 5`
`!=`	is not equal to	`buttonClicked != panicButton`
`<`	is less than	`numberOfCows < 5`
`>`	is greater than	`myInitial > 'B'`
`<=`	is less than or equal to	`numberOfCows <= 5`
`>=`	is greater than or equal to	`myInitial >= 'B'`

You can use all of Java's comparison operators to compare numbers and characters. When you compare numbers, things go pretty much the way you think they should go. But when you compare characters, things are a little strange. Comparing uppercase letters with one another is no problem. Because the letter *B* comes alphabetically before *H*, the condition `'B' < 'H'` is true. Comparing lowercase

```
What's the password? swordfish
You typed >>swordfish<<

The word you typed is not
stored in the same place as
the real password, but that's
no big deal.

The word you typed has the
same characters as the real
password. You can use our
precious system.
```

FIGURE 5-4: The results of using == and using Java's equals method.

LISTING 5-3: Checking a Password

```
import static java.lang.System.*;
import java.util.Scanner;

public class CheckPassword {

    public static void main(String args[]) {

        out.print("What's the password?");

        Scanner keyboard = new Scanner(in);
        String password = keyboard.next();

        out.println("You typed >>" + password + "<<");
        out.println();

        if (password == "swordfish") {
            out.println("The word you typed is stored");
            out.println("in the same place as the real");
            out.println("password. You must be a");
            out.println("hacker.");
        } else {
            out.println("The word you typed is not");
            out.println("stored in the same place as");
            out.println("the real password, but that's");
            out.println("no big deal.");
        }
        out.println();

        if (password.equals("swordfish")) {
            out.println("The word you typed has the");
            out.println("same characters as the real");
```

(continued)

letters with one another is also okay. What's strange is that when you compare an uppercase letter with a lowercase letter, the uppercase letter is always smaller. So, even though `'Z' < 'A'` is `false`, `'Z' < 'a'` is `true`.

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 smaller than each lowercase letter.

WARNING

Be careful when you compare two numbers for equality (with `==`) or inequality (with `!=`). After you do some calculations and obtain two `double` values or two `float` values, the values that you have are seldom dead-on equal to one another. (The problem comes from those pesky digits beyond the decimal point.) For instance, the Fahrenheit equivalent of 21 degrees Celsius is 69.8, and when you calculate `9.0 / 5 * 21 + 32` by hand, you get 69.8. But the condition `9.0 / 5 * 21 + 32 == 69.8` turns out to be false. That's because, when the computer calculates `9.0 / 5 * 21 + 32`, it gets 69.80000000000001, not 69.8.

Comparing objects

When you start working with objects, you find that you can use `==` and `!=` to compare objects with one another. For instance, a button you see on the computer screen is an object. You can ask whether the thing that was just mouse-clicked is a particular button on your screen. You do this with Java's equality operator:

```
if (e.getSource() == bCopy) {
    clipboard.setText(which.getText());
```

CROSS REFERENCE

To find out more about responding to button clicks, read Chapter 16.

The big gotcha with Java's comparison scheme comes when you compare two strings. (For a word or two about Java's `String` type, see the section about reference types in Chapter 4.) When you compare two strings with one another, you don't want to use the double equal sign. Using the double equal sign would ask, "Is this string stored in exactly the same place in memory as that other string?" Usually, that's not what you want to ask. Instead, you usually want to ask, "Does this string have the same characters in it as that other string?" To ask the second question (the more appropriate question), Java's `String` type has a method named `equals`. (Like everything else in the known universe, this `equals` method is defined in the Java API, short for application programming interface.) The `equals` method compares two strings to see whether they have the same characters in them. For an example using Java's `equals` method, see Listing 5-3. (Figure 5-4 shows a run of the program in Listing 5-3.)

When comparing strings with one another, use the `equals` method — not the double equal sign.

Importing everything in one fell swoop

The first line of Listing 5-3 illustrates a lazy way of importing both `System.out` and `System.in`. To import everything that `System` has to offer, you use the asterisk wildcard character (`*`). In fact, importing `java.lang.System.*` is like having about 30 separate import declarations, including `System.in`, `System.out`, `System.err`, `System.nanoTime`, and many other `System` things.

The use of an asterisk in an `import` declaration is generally considered bad programming practice, so I don't do it often in this book's examples. But for larger programs — programs that use dozens of names from the Java API — the lazy asterisk trick is handy.


TECHNICAL STUFF

You can't toss an asterisk anywhere you want inside an `import` declaration. For example, you can't import everything starting with `java` by writing `import java.*`. You can substitute an asterisk only for the name of a class or for the name of something static that's tucked away inside a class. For more information about asterisks in `import` declarations, see Chapter 9. For information about `static` things, see Chapter 10.

Java's logical operators

Mr. Spock would be pleased: Java has all the operators that you need for mixing and matching logical tests. The operators are shown in Table 5-2.

TABLE 5-2 Logical Operators

Operator Symbol	What It Means	Example
`&&`	and	`5 < x && x < 10`
`||`	or	`x < 5 || 10 < x`
`!`	not	`!password.equals("swordfish")`

You can use these operators to form all kinds of elaborate conditions. Listing 5-4 has an example.

LISTING 5-3: *(continued)*

```
            out.println("password. You can use our");
            out.println("precious system.");
        } else {
            out.println("The word you typed doesn't");
            out.println("have the same characters as");
            out.println("the real password. You can't");
            out.println("use our precious system.");
        }

        keyboard.close();
    }
}
```

In Listing 5-3, the call `keyboard.next()` grabs whatever word the user types on the computer keyboard. The code shoves this word into the variable named *password*. Then the program's `if` statements use two different techniques to compare `password` with `"swordfish"`.

The more appropriate of the two techniques uses Java's `equals` method. The `equals` method looks funny because when you call it, you put a dot after one string and put the other string in parentheses. But that's the way you have to do it.

In calling Java's `equals` method, it doesn't matter which string gets the dot and which gets the parentheses. For instance, in Listing 5-3, you could have written

```
if ("swordfish".equals(password))
```

The method would work just as well.


TECHNICAL STUFF

A call to Java's `equals` method looks imbalanced, but it's not. There's a reason behind the apparent imbalance between the dot and the parentheses. The idea is that you have two objects: the `password` object and the `"swordfish"` object. Each of these two objects is of type `String`. (However, `password` is a variable of type `String`, and `"swordfish"` is a `String` literal.) When you write `password.equals("swordfish")`, you're calling an `equals` method that belongs to the `password` object. When you call that method, you're feeding `"swordfish"` to the method as the method's parameter (pun intended).


CROSS REFERENCE

You can read more about methods belonging to objects in Chapter 7.

FIGURE 5-5: Several runs of the code from Listing 5-4.

from Listing 5-3. The big difference is, while `keyboard.next()` displays dull-looking text in a console, `JOptionPane.showInputDialog("Username:")` displays a fancy dialog box containing a text field and buttons. (Compare Figures 5-4 and 5-5.) When the user clicks OK, the computer takes whatever text is in the text field and hands that text over to a variable. In fact, Listing 5-4 uses `JOptionPane.showInputDialog` twice — once to get a value for the username variable and a second time to get a value for the password variable.

Near the end of Listing 5-4, I use a slight variation on the `JOptionPane` business,

```
JOptionPane.showMessageDialog(null, "You're in.");
```

With `showMessageDialog`, I show a very simple dialog box — a box with no text field. (Again, see Figure 5-5.)

Like thousands of other names, the name `JOptionPane` is defined in Java's API. (To be more specific, `JOptionPane` is defined inside something called `javax.swing`, which in turn is defined inside Java's API.) So, to use the name `JOptionPane` throughout Listing 5-4, I import `javax.swing.JOptionPane` at the top of the listing.

LISTING 5-4: Checking Username and Password

```
import javax.swing.JOptionPane;

public class Authenticator {

    public static void main(String args[]) {

        String username = JOptionPane.showInputDialog("Username:");
        String password = JOptionPane.showInputDialog("Password:");

        if (
              username != null && password != null &&
              (
                (username.equals("bburd") && password.equals("swordfish")) ||
                (username.equals("hritter") && password.equals("preakston"))
              )
           )
        {
            JOptionPane.showMessageDialog(null, "You're in.");
        } else {
            JOptionPane.showMessageDialog(null, "You're suspicious.");
        }
    }
}
```

Several runs of the program of Listing 5-4 are shown in Figure 5-5. When the username is *bburd* and the password is *swordfish* or when the username is *hritter* and the password is *preakston*, the user gets a nice message. Otherwise, the user is a bum who gets the nasty message that he or she deserves.

Confession: Figure 5-5 is a fake! To help you read the usernames and passwords, I added an extra statement to Listing 5-4. The extra statement (`UIManager.put("TextField.font", new Font("Dialog", Font.BOLD, 14))`) enlarges each text field's font size. Yes, I modified the code before creating the figure. Shame on me!

Listing 5-4 illustrates a new way to get user input; namely, to show the user an input dialog box. The statement

```
String password = JOptionPane.showInputDialog("Password:");
```

in Listing 5-4 performs more or less the same task as the statement

```
String password = keyboard.next();
```

```
if (
      you didn't press Cancel in the username dialog and
      you didn't press Cancel in the password dialog and
      (
        (you typed bburd in the username dialog and
         you typed swordfish in the password dialog) or
        (you typed hritter in the username dialog and
         you typed preakston in the password dialog)
      )
   )
```


WARNING

In Listing 5-4, the comparisons `username != null` and `password != null` are not optional. If you forget to include these and click Cancel when the program runs, you get a nasty `NullPointerException` message, and the program comes crashing down before your eyes. The word `null` represents nothing, and in Java, you can't compare nothing to a string like `"bburd"` or `"swordfish"`. In Listing 5-4, the purpose of the comparison `username != null` is to prevent Java from moving on to check `username.equals("bburd")` whenever you happen to click Cancel. Without this preliminary `username != null` test, you're courting trouble.


TECHNICAL STUFF

The last couple of `null`s in Listing 5-4 are different from the others. In the code `JOptionPane.showMessageDialog (null, "You're in.")`, the word `null` stands for "no other dialog box." In particular, the call `showMessageDialog` tells Java to pop up a new dialog box, and the word `null` indicates that the new dialog box doesn't grow out of any existing dialog box. One way or another, Java insists that you say something about the origin of the newly popped dialog box. (For some reason, Java doesn't insist that you specify the origin of the `showInputDialog` box. Go figure!) Anyway, in Listing 5-4, having a `showMessageDialog` box pop up from nowhere is quite useful.

(Conditions in parentheses)

Keep an eye on those parentheses! When you're combining conditions with logical operators, it's better to waste typing effort and add unneeded parentheses than to goof up your result by using too few parentheses. Take, for example, the expression

```
2 < 5 || 100 < 6 && 27 < 1
```

By misreading this expression, you might conclude that the expression is false. That is, you could wrongly read the expression as meaning (*something-or-other*) `&&` `27 < 1`. Because `27 < 1` is false, you would conclude that the whole expression is false. The fact is that, in Java, any `&&` operator is evaluated before


TIP

In Listing 5-4, `JOptionPane.showInputDialog` works nicely because the user's input (username and password) are mere strings of characters. If you want the user to input a number (an `int` or a `double`, for example), you have to do some extra work. For example, to get an `int` value from the user, type something like `int numberOfCows = Integer.parseInt(JOptionPane.showInputDialog("How many cows?"))`. The extra `Integer.parseInt` stuff forces your text field's input to be an `int` value. To get a `double` value from the user, type something like `double fractionOfHolsteins = Double.parseDouble(JOptionPane.showInputDialog("Holsteins:"))`. The extra `Double.parseDouble` business forces your text field's input to be a `double` value.

Vive les nuls!

The French translations of *For Dummies* books are books *Pour les Nuls*. So a "dummy" in English is a "nul" in French.* But in Java, the word `null` means "nothing." When you see

```
if (
      username != null
```

in Listing 5-4, you can imagine that you see

```
if (
      username isn't nothing
```

or

```
if (
      username has any value at all
```

To find out how username can have no value, see the last row in Figure 5-5. When you click Cancel in the first dialog box, the computer hands `null` to your program. So, in Listing 5-4, the variable username becomes `null`. The comparisons `username != null` checks to make sure that you haven't clicked Cancel in the program's first dialog box. The comparison `password != null` performs the same kind of check for the program's second dialog box. When you see the `if` statement in Listing 5-4, you can imagine that you see the following:

*In Russian, a "dummy" is a "чайник" which, when interpreted literally, means a "teapot." So in Russian, this book is "Java For Teapots." I've never been called a "teapot," and I'm not sure how I'd react if I were.

Anyway, when Java finally checks `username.equals("hritter")`, your program aborts with an ugly `NullPointerException` message. You've made Java angry by trying to apply `.equals` to a `null` username. (Psychiatrists have recommended anger management sessions for Java, but Java's insurance plan refuses to pay for the sessions.)


TRY IT OUT

Make some changes to the code in Listing 5-4:

- Add a third username/password combination to the list of acceptable logins.
- Modify the `if` statement's condition so that an all-uppercase entry for either username is acceptable. In other words, the input `BBURD` yields the same result as `bburd` and the input `HRITTER` yields the same result as `hritter`.
- In Listing 5-4, change

```
username != null && password != null
```

to

```
!(username == null || password == null)
```

Does the program still work? Why, or why not?

- In Listing 5-4, change

```
username != null && password != null
```

to

```
!(username == null && password == null)
```

This is almost the same as the previous experiment. The only difference is the use of `&&` instead of `||` between the two `== null` tests.

Does the program still work? Why, or why not?

Building a Nest

Have you seen those cute Russian matryoshka nesting dolls? Open one, and another one is inside. Open the second, and a third one is inside it. You can do the same thing with Java's `if` statements. (Talk about fun!) Listing 5-5 shows you how.

any `||` operator. So the expression really asks whether `2 < 5 ||` (*something-or-other*). Because `2 < 5` is true, the whole expression is true.

To change the expression's value from `true` to `false`, you can put the expression's first two comparisons in parentheses, like this:

```
(2 < 5 || 100 < 6) && 27 < 1
```


TIP

Java's `||` operator is *inclusive*. This means that you get a `true` value whenever the thing on the left side is true, the thing on the right side is true, or both things are true. For instance, the expression `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`.

In Listing 5-4, the `if` statement's condition has more than a dozen parentheses. What happens if you omit two of them?

```
if (
      username != null && password != null &&
      // open parenthesis omitted
          (username.equals("bburd") && password.equals("swordfish")) ||
          (username.equals("hritter") && password.equals("preakston"))
      // close parenthesis omitted
   )
```

Java tries to interpret your wishes by grouping everything before the "or" (the `||` operator):

```
if (
      username != null && password != null &&
      (username.equals("bburd") && password.equals("swordfish"))

      ||

      (username.equals("hritter") && password.equals("preakston"))
   )
```

When the user clicks Cancel and username is `null`, Java says, "Okay! The stuff before the `||` operator is false, but maybe the stuff after the `||` operator is true. I'll check the stuff after the `||` operator to find out whether it's true." (Java often talks to itself. The psychiatrists are monitoring this situation.)

LISTING 5-5: Nested if Statements

```
import static java.lang.System.out;
import java.util.Scanner;

public class Authenticator2 {

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

        out.print("Username: ");
        String username = keyboard.next();

        if (username.equals("bburd")) {
            out.print("Password: ");
            String password = keyboard.next();

            if (password.equals("swordfish")) {
                out.println("You're in.");
            } else {
                out.println("Incorrect password");
            }

        } else {
            out.println("Unknown user");
        }

        keyboard.close();
    }
}
```

Figure 5-6 shows several runs of the code in Listing 5-5. The main idea is that to log on, you have to pass two tests. (In other words, two conditions must be true.) The first condition tests for a valid username; the second condition tests for the correct password. If you pass the first test (the username test), you march right into another `if` statement that performs a second test (the password test). If you fail the first test, you never make it to the second test. Figure 5-7 shows the overall plan.

WARNING

The code in Listing 5-5 does a good job with nested `if` statements, but it does a terrible job with real-world user authentication. First, never show a password in plain view (without asterisks to masquerade the password). Second, don't handle passwords without encrypting them. Third, don't tell the malicious user which of the two words (the username or the password) was entered incorrectly. Fourth . . . well, I could go on and on. The code in Listing 5-5 just isn't meant to illustrate good username/password practices.


```
Username: bburd
Password: swordfish
You're in.


Username: bburd
Password: catfish
Incorrect password


Username: jschmoe
Unknown user
```

FIGURE 5-6: Three runs of the code in Listing 5-5.

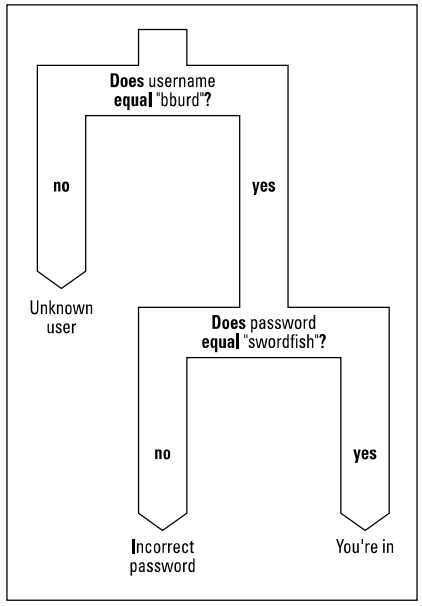

FIGURE 5-7: Don't try eating with this fork.

TRY IT OUT

Modify the program in Listing 5-4 so that, if the user clicks Cancel for either the username or the password, the program replies with a `Not enough information` message.

Choosing among Many Alternatives (Java switch Statements)

I'm the first to admit that I hate making decisions. If things go wrong, I would rather have the problem be someone else's fault. Writing the previous sections (on making decisions with Java's `if` statement) knocked the stuffing right out of me. That's why my mind boggles as I begin this section on choosing among many alternatives. What a relief it is to have that confession out of the way!

Your basic switch statement

Now, it's time to explore situations in which you have a decision with many branches. Take, for instance, the popular campfire song "Al's All Wet." (For a review of the lyrics, see the "Al's All Wet" sidebar.) You're eager to write code that prints this song's lyrics. Fortunately, you don't have to type all the words over and over again. Instead, you can take advantage of the repetition in the lyrics.

"AL'S ALL WET"

Sung to the tune of "Gentille Alouette":

Al's all wet. Oh, why is Al all wet? Oh,

Al's all wet 'cause he's standing in the rain.

Why is Al out in the rain?

That's because he has no brain.

Has no brain, has no brain,

In the rain, in the rain.

Ohhhhhhhh ...

Al's all wet. Oh, why is Al all wet? Oh,

Al's all wet 'cause he's standing in the rain.

Why is Al out in the rain?

That's because he is a pain.

He's a pain, he's a pain,

Has no brain, has no brain,

In the rain, in the rain.

Ohhhhhhhh ...

Al's all wet. Oh, why is Al all wet? Oh,

Al's all wet 'cause he's standing in the rain.

Why is Al out in the rain?

'Cause this is the last refrain.

Last refrain, last refrain,

He's a pain, he's a pain,

Has no brain, has no brain,

In the rain, in the rain.

Ohhhhhhhh ...

Al's all wet. Oh, why is Al all wet? Oh,

Al's all wet 'cause he's standing in the rain.

—*Harriet Ritter and Barry Burd*

```
            default:
               out.println("No such verse. Please try again.");
               break;
            }

            out.println("Ohhhhhhhh ... .");

            keyboard.close();
         }
}
```

Figure 5-8 shows two runs of the program in Listing 5-6. (Figure 5-9 illustrates the program's overall idea.) First, the user types a number, like the number 2. Then execution of the program reaches the top of the `switch` statement. The computer checks the value of the `verse` variable. When the computer determines that the `verse` variable's value is `2`, the computer checks each case of the `switch` statement. The value `2` doesn't match the topmost case, so the computer proceeds to the middle of the three cases. The value posted for the middle case (the number 2) matches the value of the `verse` variable, so the computer executes the statements that come immediately after case 2. These two statements are

```
out.println("That's because he is a pain.");
break;
```

```
Which verse? 2
That's because he is a pain.
Ohhhhhhhh. . . .


Which verse? 6
No such verse. Please try again.
Ohhhhhhhh. . . .
```

FIGURE 5-8: Running the code of Listing 5-6 two times.

The first of the two statements displays the line `That's because he is a pain.` on the screen. The second statement is called a `break` statement. (What a surprise!) When the computer encounters a `break` statement, the computer jumps out of whatever `switch` statement it's in. So, in Listing 5-6, the computer skips right past the case that would display `'Cause this is the last refrain`. In fact, the computer jumps out of the entire `switch` statement and goes straight to the statement just after the end of the `switch` statement. The computer displays `Ohhhhhhhh ...` because that's what the statement after the `switch` statement tells the computer to do.

A complete program to display the "Al's All Wet" lyrics won't come until Chapter 6. In the meantime, assume that you have a variable named `verse`. The value of `verse` is `1`, `2`, `3`, or `4`, depending on which verse of "Al's All Wet" you're trying to print. You could have a big, clumsy bunch of `if` statements that checks each possible `verse` number:

```
if (verse == 1) {
    out.println("That's because he has no brain.");
}
if (verse == 2) {
    out.println("That's because he is a pain.");
}
if (verse == 3) {
    out.println("'Cause this is the last refrain.");
}
```

But that approach seems wasteful. Why not create a statement that checks the value of `verse` just once and then takes an action based on the value it finds? Fortunately, just such a statement exists. It's called a *switch* statement. Listing 5-6 has an example of a `switch` statement.

LISTING 5-6: A switch Statement

```
import static java.lang.System.out;
import java.util.Scanner;

public class JustSwitchIt {

    public static void main(String args[]) {
       Scanner keyboard = new Scanner(System.in);
       out.print("Which verse? ");
       int verse = keyboard.nextInt();

       switch (verse) {
       case 1:
          out.println("That's because he has no brain.");
          break;
       case 2:
          out.println("That's because he is a pain.");
          break;
       case 3:
          out.println("'Cause this is the last refrain.");
          break;
```

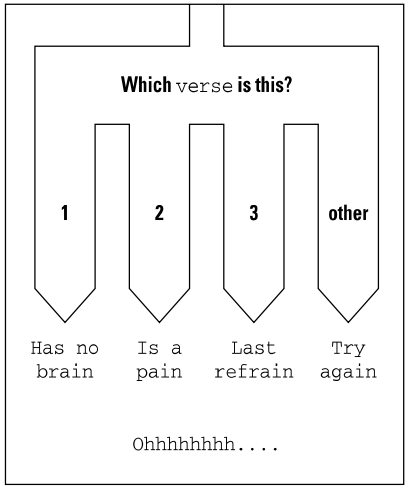

FIGURE 5-9: The big fork in the code of Listing 5-6.

If the pesky user asks for verse 6, the computer bypasses cases 1, 2, and 3. The computer goes straight to the default. In the default, the computer displays `No such verse. Please try again`, and then breaks out of the `switch` statement. After the computer is out of the `switch` statement, the computer displays `Ohhhhhhhh ...`

TIP

You don't really need to put a `break` at the very end of a `switch` statement. In Listing 5-6, the last `break` (the `break` that's part of the default) is just for the sake of overall tidiness.

To break or not to break

In every Java programmer's life, a time comes when he or she forgets to use `break` statements. At first, the resulting output is confusing, but then the programmer remembers fall-through. The term *fall-through* describes what happens when you end a case without a `break` statement. What happens is that execution of the code falls right through to the next case in line. Execution keeps falling through until you eventually reach a `break` statement or the end of the entire `switch` statement.

Usually, when you're using a `switch` statement, you don't want fall-through, so you pepper `break` statements throughout the `switch` statements. But, occasionally, fall-through is just the thing you need. Take, for instance, the "Al's All Wet" song. (The classy lyrics are shown in the sidebar bearing the song's name.) Each

verse of "Al's All Wet" adds new lines in addition to the lines from previous verses. This situation (accumulating lines from one verse to another) cries out for a `switch` statement with fall-through. Listing 5-7 demonstrates the idea.

LISTING 5-7: A switch Statement with Fall-Through

```
import static java.lang.System.out;
import java.util.Scanner;

public class FallingForYou {

    public static void main(String args[]) {
        Scanner keyboard = new Scanner(System.in);
        out.print("Which verse? ");
        int verse = keyboard.nextInt();

        switch (verse) {
        case 3:
            out.print("Last refrain, ");
            out.println("last refrain,");
        case 2:
            out.print("He's a pain, ");
            out.println("he's a pain,");
        case 1:
            out.print("Has no brain, ");
            out.println("has no brain,");
        }

        out.println("In the rain, in the rain.");
        out.println("Ohhhhhhhh...");
        out.println();

        keyboard.close();
    }
}
```

Figure 5-10 shows several runs of the program in Listing 5-7. Because the switch has no `break` statements in it, fall-through happens all over the place. For instance, when the user selects verse 2, the computer executes the two statements in case 2:

```
out.print("He's a pain, ");
out.println("he's a pain,");
```

Then the computer marches right on to execute the two statements in case 1:

```
out.print("Has no brain, ");
out.println("has no brain,");
```

That's good because the song's second verse has all these lines in it.

```
Which verse? 1
Has no brain, has no brain,
In the rain, in the rain.
Ohhhhhhhh...


Which verse? 2
He's a pain, he's a pain,
Has no brain, has no brain,
In the rain, in the rain.
Ohhhhhhhh...


Which verse? 3
Last refrain, last refrain,
He's a pain, he's a pain,
Has no brain, has no brain,
In the rain, in the rain.
Ohhhhhhhh...


Which verse? 6
In the rain, in the rain.
Ohhhhhhhh...
```

FIGURE 5-10: Running the code of Listing 5-7 four times.

Notice what happens when the user asks for verse 6. The `switch` statement in Listing 5-7 has no case 6 and no default, so none of the actions inside the `switch` statement is executed. Even so, with statements that print `In the rain, in the rain` and `Ohhhhhhhh ...` right after the `switch` statement, the computer displays something when the user asks for verse 6.

Strings in a switch statement

In Listings 5-6 and 5-7, shown earlier, the variable `verse` (an `int` value) steers the `switch` statement to one case or another. An `int` value inside a `switch` statement works in any version of Java, old or new. (For that matter, `char` values and a few other kinds of values have worked in Java's `switch` statements ever since Java was a brand-new language.)

Starting with Java 7, you can set it up so that the case to be executed in a `switch` statement depends on the value of a particular string. Listing 5-8 illustrates the use of strings in `switch` statements. Figure 5-11 shows a run of the code in Listing 5-8.

LISTING 5-8: A switch Statement with a String

```
import static java.lang.System.out;
import java.util.Scanner;

public class SwitchIt7 {

    public static void main(String args[]) {
       Scanner keyboard = new Scanner(System.in);
       out.print("Which verse (one, two or three)? ");
       String verse = keyboard.next();

       switch (verse) {
       case "one":
          out.println("That's because he has no brain.");
          break;
       case "two":
          out.println("That's because he is a pain.");
          break;
       case "three":
          out.println("'Cause this is the last refrain.");
          break;
       default:
          out.println("No such verse. Please try again.");
          break;
       }

       out.println("Ohhhhhhhh... .");

       keyboard.close();
    }
}
```

```
Which verse (one, two or three)? two
That's because he is a pain.
Ohhhhhhhh. . . .
```

FIGURE 5-11: Running the code of Listing 5-8.

IN THIS CHAPTER

- Using basic looping
- Counting as you loop
- Repeating relentlessly (until the user gives you a clear answer)

Chapter 6

Controlling Program Flow with Loops

In 1966, the company that brings you Head & Shoulders shampoo made history. On the back of the bottle, the directions for using the shampoo read, "LATHER-RINSE-REPEAT." Never before had a complete set of directions (for doing anything, let alone shampooing your hair) been summarized so succinctly. People in the direction-writing business hailed this as a monumental achievement. Directions like these stood in stark contrast to others of the time. (For instance, the first sentence on a can of bug spray read, "Turn this can so that it points away from your face." Duh!)

Aside from their brevity, the thing that made the Head & Shoulders directions so cool was that, with three simple words, it managed to capture a notion that's at the heart of all instruction-giving — the notion of repetition. That last word, *REPEAT*, took an otherwise bland instructional drone and turned it into a sophisticated recipe for action.

The fundamental idea is that when you're following directions, you don't just follow one instruction after another. Instead, you take turns in the road. You make decisions ("If HAIR IS DRY, then USE CONDITIONER") and you go into loops ("LATHER-RINSE, and then LATHER-RINSE again."). In computer programming, you use decision-making and looping all the time. This chapter explores looping in Java.

TRY IT OUT

Get some practice with `if` statements and `switch` statements!

- Write a program that inputs the name of a month and outputs the number of days in that month. In this first version of the program, assume that February always has 28 days.
- Make your code even better! Have the user input a month name, but also have the user input `yes` or `no` in response to the question `Is it a leap year?`

```
        out.print("You win after ");
        out.println(numGuesses + " guesses.");

        keyboard.close();
    }
}
```

```
       ************
Welcome to the Guessing Game
       ************

Enter an int from 1 to 10: 2

Try again...
Enter an int from 1 to 10: 5

Try again...
Enter an int from 1 to 10: 8

Try again...
Enter an int from 1 to 10: 3
You win after 4 guesses.
```

FIGURE 6-1: Play until you drop.

In Figure 6-1, the user makes four guesses. Each time around, the computer checks to see whether the guess is correct. An incorrect guess generates a request to try again. For a correct guess, the user gets a rousing `You win`, along with a tally of the number of guesses he or she made. The computer repeats several statements, checking each time through to see whether the user's guess is the same as a certain randomly generated number. Each time the user makes a guess, the computer adds 1 to its tally of guesses. When the user makes the correct guess, the computer displays that tally. Figure 6-2 illustrates the flow of action.

When you look over Listing 6-1, you see the code that does all this work. At the core of the code is a thing called a *while statement* (also known as a *while loop*). Rephrased in English, the `while` statement says:

```
while the inputNumber is not equal to the randomNumber
keep doing all the stuff in curly braces: {

}
```

The stuff in curly braces (the stuff that repeats) is the code that prints `Try again` and `Enter an int ...`, gets a value from the keyboard, and adds 1 to the count of the user's guesses.

Repeating Instructions Over and Over Again (Java while Statements)

Here's a guessing game for you. The computer generates a random number from 1 to 10. The computer asks you to guess the number. If you guess incorrectly, the game continues. As soon as you guess correctly, the game is over. Listing 6-1 shows the program to play the game, and Figure 6-1 shows a round of play.

LISTING 6-1: A Repeating Guessing Game

```
import static java.lang.System.out;
import java.util.Scanner;
import java.util.Random;

public class GuessAgain {

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

        int numGuesses = 0;
        int randomNumber = new Random().nextInt(10) + 1;

        out.println("       ************        ");
        out.println("Welcome to the Guessing Game");
        out.println("       ************        ");
        out.println();

        out.print("Enter an int from 1 to 10: ");
        int inputNumber = keyboard.nextInt();
        numGuesses++;

        while (inputNumber != randomNumber) {
            out.println();
            out.println("Try again...");
            out.print("Enter an int from 1 to 10: ");
            inputNumber = keyboard.nextInt();
            numGuesses++;
        }
```

REMEMBER

With code of the kind shown in Listing 6-1, the computer never jumps out in mid-loop. When the computer finds that `inputNumber` isn't equal to `randomNumber`, the computer marches on and executes all five statements inside the loop's curly braces. The computer performs the test again (to see whether `inputNumber` is still not equal to `randomNumber`) only after it fully executes all five statements in the loop.

TRY IT OUT

I have two things for you to try:

- Modify the program in Listing 6-1 so that the randomly generated number is a number from 1 and 100. To make life bearable for the game player, have the program give a hint whenever the player guesses incorrectly. Hints such as `Try a higher number` or `Try a lower number` are very helpful.
- Write a program in which the user types `int` values, one after another. The program stops looping when the user types a number that isn't positive (for example, the number 0 or the number –17). After all the looping, the program displays the largest number that the user typed. For example, if the user types the numbers

```
7
25
3
9
0
```

the program displays the number 25.

Repeating a Certain Number of Times (Java for Statements)

"Write *I will not talk in class* on the blackboard 100 times."

What your teacher really meant was this:

```
Set the count to 0.
As long as the count is less than 100,
    Write 'I will not talk in class' on the blackboard,
    Add 1 to the count.
```

Fortunately, you didn't know about loops and counters at the time. If you pointed out all this stuff to your teacher, you'd have gotten into a lot more trouble than you were already in.

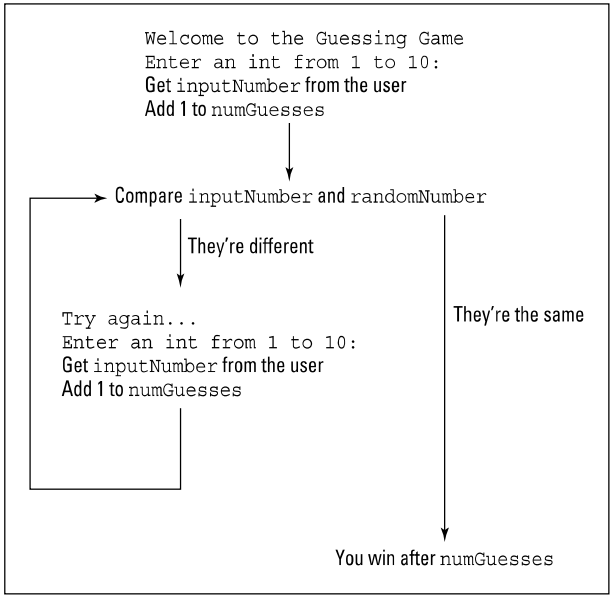

FIGURE 6-2: Around and around you go.

TIP

When you're dealing with counters, like `numGuesses` in Listing 6-1, you may easily become confused and be off by 1 in either direction. You can avoid this headache by making sure that the `++` statements stay close to the statements whose events you're counting. For example, in Listing 6-1, the variable `numGuesses` starts with a value of `0`. That's because, when the program starts running, the user hasn't made any guesses. Later in the program, right after each call to `keyboard.nextInt`, is a `numGuesses++` statement. That's how you do it — you increment the counter as soon as the user enters another guess.

The statements in curly braces are repeated as long as `inputNumber != randomNumber` is true. Each repetition of the statements in the loop is called an iteration of the loop. In Figure 6-1, the loop undergoes three iterations. (If you don't believe that Figure 6-1 has exactly three iterations, count the number of `Try again` printings in the program's output. A `Try again` appears for each incorrect guess.)

When, at long last, the user enters the correct guess, the computer goes back to the top of the `while` statement, checks the condition in parentheses, and finds itself in double double-negative land. The not equal `(!=)` relationship between `inputNumber` and `randomNumber` no longer holds. In other words, the `while` statement's condition, `inputNumber != randomNumber`, is false. Because the `while` statement's condition is false, the computer jumps past the `while` loop and goes on to the statements just below the `while` loop. In these two statements, the computer prints `You win` after four guesses.

One way or another, 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 is so 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.* Listings 6-2 and 6-3 illustrate the use of the `for` statement. Listing 6-2 has a rock-bottom simple example, and Listing 6-3 has a more exotic example. Take your pick.

LISTING 6-2: The World's Most Boring for Loop

```
import static java.lang.System.out;

public class Yawn {

    public static void main(String args[]) {

        for (int count = 1; count <= 10; count++) {
            out.print("The value of count is ");
            out.print(count);
            out.println(".");
        }

        out.println("Done!");
    }
}
```

Figure 6-3 shows you what you get when you run the program of Listing 6-2. (You get exactly what you deserve.) The `for` statement in Listing 6-2 starts by setting the `count` variable to `1`. Then the statement tests to make sure that `count` is less than or equal to 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 `The value of count is 1`.) Finally, the `for` statement does that last thing inside its parentheses — it adds 1 to the value of `count`.

With `count` now equal to 2, the `for` statement checks again to make sure that count is less than or equal to 10. (Yes, 2 is smaller than 10.) Because the test turns out okay, the `for` statement marches back into the curly braced statements and prints `The value of count is 2` on the screen. Finally, the `for` statement does that last thing inside its parentheses — it adds 1 to the value of `count`, increasing the value of `count` to 3.


```
The value of count is 1.
The value of count is 2.
The value of count is 3.
The value of count is 4.
The value of count is 5.
The value of count is 6.
The value of count is 7.
The value of count is 8.
The value of count is 9.
The value of count is 10.
Done!
```

FIGURE 6-3: Counting to ten.

And so on. This whole thing repeats until, after ten iterations, the value of `count` finally reaches 11. When this happens, the check for `count` being less than or equal to ten fails, and the loop's execution ends. The computer jumps to whatever statement comes immediately after the `for` statement. In Listing 6-2, the computer prints `Done!` as its output. Figure 6-4 illustrates the whole process.

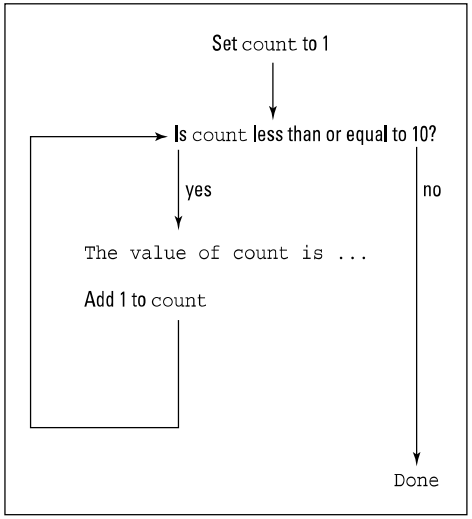

FIGURE 6-4: The action of the `for` loop in Listing 6-2.

The anatomy of a for statement

After the word *for,* you always put three things in parentheses. The first of these three is called an *initialization,* the second is an *expression,* and the third is an *update:*

```
for ( initialization ; expression ; update )
```

Each of the three items in parentheses plays its own distinct role:

- The **initialization** is executed once, when the run of your program first reaches the `for` statement.
- The **expression** is evaluated several times (before each iteration).
- 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:

```
int count = 1
for count <= 10 {
    out.print("The value of count is ");
    out.print(count);
    out.println(".");
    count++;
}
```

You can't write a real `for` statement this way. Even so, this is the order in which the parts of the statement are executed.

WARNING

If you declare a variable in the initialization of a `for` loop, you can't use that variable outside the loop. For instance, in Listing 6-2, you get an error message if you try putting `out.println(count)` after the end of the loop.

TIP

Anything that can be done with a `for` loop can also be done with a `while` loop. Choosing to use a `for` loop is a matter of style and convenience, not necessity.

TRY IT OUT

Would you like some practice? Try these experiments and challenges:

- A `for` statement's initialization may have several parts. A `for` statement's update may also have several parts. To find out how, enter the following lines in Java's JShell, or add the lines to a small Java program:

```
import static java.lang.System.out
for (int i = 0, j = 10; i < j; i++, j--) {out.println(i + " " + j);}
```

- What's the output of the following code?

```
int total = 0;
for (int i = 0; i < 10; i++) {
  total += i;
}
System.out.println(total);
```


In this code, the variable `total` is called an *accumulator* because it accumulates (adds up) a bunch of values inside the loop.

- In mathematics, the exclamation point (!) means *factorial* — the number you get when you multiply all the positive `int` values up to and including a certain number. For example, 3! is $1 \times 2 \times 3$, which is 6. And 5! is $1 \times 2 \times 3 \times 4 \times 5$, which is 120.

  Write a program in which the user types a positive `int` value (call it *n*), and Java displays the value of *n*! as its output.

- Without running the following code, try to predict what the code's output will be:

```
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 experiment is a slight variation on the code in the previous experiment. 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();
}
```

- Write a program that uses loops to display three copies of the following pattern, one after another:

```
*
**
***
****
*****
```

The world premiere of "Al's All Wet"

Listing 6-2 is very nice, but the program in that listing doesn't do anything interesting. For a more eye-catching example, see Listing 6-3. In Listing 6-3, I make

good on a promise I make in Chapter 5. The program in Listing 6-3 prints all the lyrics of the hit single "Al's All Wet." (You can find the lyrics in Chapter 5.)

LISTING 6-3: The Unabridged "Al's All Wet" Song

```
import static java.lang.System.out;

public class AlsAllWet {

   public static void main(String args[]) {

      for (int verse = 1; verse <= 3; verse++) {
        out.print("Al's all wet. ");
        out.println("Oh, why is Al all wet? Oh,");
        out.print("Al's all wet 'cause ");
        out.println("he's standing in the rain.");
        out.println("Why is Al out in the rain?");

        switch (verse) {
        case 1:
          out.println("That's because he has no brain.");
          break;
        case 2:
          out.println("That's because he is a pain.");
          break;
        case 3:
          out.println("'Cause this is the last refrain.");
          break;
        }

        switch (verse) {
        case 3:
          out.println("Last refrain, last refrain,");
        case 2:
          out.println("He's a pain, he's a pain,");
        case 1:
          out.println("Has no brain, has no brain,");
        }

        out.println("In the rain, in the rain.");
        out.println("Ohhhhhhhh...");
        out.println();
      }
```



```
      out.print("Al's all wet. ");
      out.println("Oh, why is Al all wet? Oh,");
      out.print("Al's all wet 'cause ");
      out.println("he's standing in the rain.");
   }
}
```

Listing 6-3 is nice because it combines many of the ideas from Chapters 5 and 6. In Listing 6-3, two `switch` statements are nested inside a `for` loop. One of the `switch` statements uses `break` statements; the other `switch` statement uses fall-through. As the value of the for loop's counter variable (`verse`) goes from `1` to `2` and then to `3`, all the cases in the `switch` statements are executed. When the program is near the end of its run and execution has dropped out of the `for` loop, the program's last four statements print the song's final verse.

TECHNICAL STUFF

When I boldly declare that a `for` statement is for counting, I'm stretching the truth just a bit. Java's `for` statement is very versatile. You can use a `for` statement in situations that have nothing to do with counting. For instance, a statement with no update part, such as `for (i = 0; i < 10; )`, just keeps on going. The looping ends when some action inside the loop assigns a big number to the variable `i`. You can even create a `for` statement with nothing inside the parentheses. The loop `for ( ; ; )` runs forever, which is good if the loop controls a serious piece of machinery. 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.

TRY IT OUT

Look! I have some experiments for you to try!

» Listing 6-3 uses `break` statements to jump out of a `switch`. But a `break` statement can also play a role inside a loop. To find out how it works, run a program containing the following code:

```
Scanner keyboard = new Scanner(System.in);
while (true) {
  System.out.print("Enter an int value: ");
  int i = keyboard.nextInt();
  if (i == 0) {
    break;
  }
  System.out.println(i);
}
System.out.println("Done!");
keyboard.close();
```

The loop's condition is always `true`. It's like starting a loop with the line

```
while (1 + 1 == 2)
```

If it weren't for the `break` statement, the loop would run forever. Fortunately, when you execute the `break` statement, Java jumps to the code immediately after the loop.

» In addition to its `break` statement, Java has a `continue` statement. When you execute a `continue` statement, Java skips to the end of its loop and begins the next iteration of that loop. To see it in action, run a program containing the following code:

```
Scanner keyboard = new Scanner(System.in);

while (true) {
  System.out.print("Enter an int value: ");
  int i = keyboard.nextInt();
  if (i > 10) {
    continue;
  }
  if (i == 0) {
    break;

  }
  System.out.println(i);
}

System.out.println("Done!");
keyboard.close();
```

Repeating until You Get What You Want (Java do Statements)

Fools rush in where angels fear to tread.

—ALEXANDER POPE

Today, I want to be young and foolish (or, at the very least, foolish). Look back at Figure 6-2 and notice how Java's `while` loop works. When execution enters a `while` loop, the computer checks to make sure that the loop's condition is true. If the condition isn't true, the statements inside the loop are 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):

```
int twoPlusTwo = 2 + 2;

while (twoPlusTwo == 5) {
    out.println("Are you kidding?");
    out.println("2 + 2 doesn't equal 5");
    out.print("Everyone knows that");
    out.println(" 2 + 2 equals 3");
}
```

In spite of this silly `twoPlusTwo` example, the `while` statement turns out to be the most versatile 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. Take, for instance, the situation in which you're asking the user for a response. Maybe the user's response makes sense, but maybe it doesn't. If it doesn't, you want to ask again. Maybe the user's finger slipped, or perhaps the user didn't understand the question.

Figure 6-5 shows some runs of a program to delete a file. Before deleting the file, the program asks the user whether making the deletion is okay. If the user answers *y* or *n*, the program proceeds according to the user's wishes. But if the user enters any other character (any digit, uppercase letter, punctuation symbol, or whatever), the program asks the user for another response.

```
Delete evidence? (y/n) n
Sorry, buddy. Just asking.


Delete evidence? (y/n) u
Delete evidence? (y/n) Y
Delete evidence? (y/n) L
Delete evidence? (y/n) 8
Delete evidence? (y/n) .
Delete evidence? (y/n) y
Okay, here goes...
The evidence has been deleted.
```

FIGURE 6-5: Two runs of the code in Listing 6-4.

To write this program, you need a loop — a loop that repeatedly asks the user whether the file should be deleted. The loop keeps asking until the user gives a meaningful response. Now, the thing to notice is that the loop doesn't need to check anything before asking the user the first time. Indeed, before the user gives

```
do {
    out.print("Delete evidence? (y/n) ");
    reply = keyboard.findWithinHorizon(".", 0).charAt(0);
} while (reply != 'y' && reply != 'Y' && reply != 'n' && reply!='N');

if (reply == 'y' || reply == 'Y') {
```

Figure 6-6 shows the flow of control in the loop of Listing 6-4. With a `do` loop, the situation in the `twoPlusTwo` program (shown at the beginning of this section) can never happen. Because the `do` loop carries out its first action without testing a condition, every `do` loop is guaranteed to perform at least one iteration.

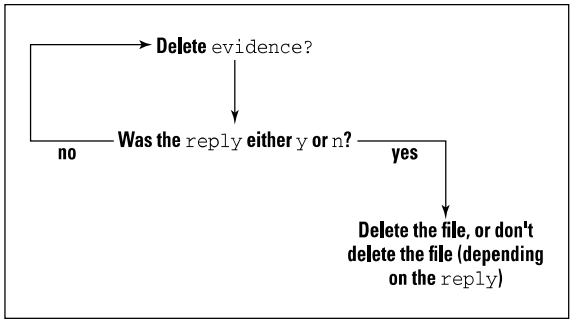

FIGURE 6-6: Here we go loop, do loop.

TIP

The location of Listing 6-4's `cookedBooks.txt` file on your computer's hard drive depends on several factors. If you create a `cookedBooks.txt` file in the wrong directory, the code in Listing 6-4 cannot delete your file. (More precisely, if `cookedBooks.txt` is in the wrong directory on your hard drive, the code in Listing 6-4 can't find the `cookedBooks.txt` file in preparation for deleting the file.) In most settings, you start testing Listing 6-4 by creating a project within your IDE. The new project lives in a folder on your hard drive, and the `cookedBooks.txt` file belongs directly inside that folder. For example, I have a project named `06-04`. That project lives on my hard drive in a folder named `06-04`. Inside that folder, I have a file named `cookedBooks.txt`. If you have trouble with this, add the following code to Listing 6-4 immediately after the `new File` statement:

```
try {
  out.println("Looking for " + evidence.getCanonicalPath());
} catch (java.io.IOException e) {
  e.printStackTrace();
}
```


the first response, the loop has nothing to check. The loop doesn't start with "as long as such-and-such is true, then get a response from the user." Instead, the loop just leaps ahead, gets a response from the user, and then checks the response to see whether it makes sense.

That's why the program in Listing 6-4 has a *do* loop (also known as a *do . . . while* loop). With a `do` loop, the program jumps right in, takes action, and then checks a condition to see whether the result of the action makes sense. If the result makes sense, execution of the loop is done. If not, the program goes back to the top of the loop for another go-round.

LISTING 6-4: To Delete or Not to Delete

```
import java.io.File;
import static java.lang.System.out;
import java.util.Scanner;

public class DeleteEvidence {

   public static void main(String args[]) {
      File evidence = new File("cookedBooks.txt");
      Scanner keyboard = new Scanner(System.in);
      char reply;

      do {
         out.print("Delete evidence? (y/n) ");
         reply = keyboard.findWithinHorizon(".",0).charAt(0);
      } while (reply != 'y' && reply != 'n');

      if (reply == 'y') {
         out.println("Okay, here goes...");
         evidence.delete();
         out.println("The evidence has been deleted.");
      } else {
         out.println("Sorry, buddy. Just asking.");
      }

      keyboard.close();
   }
}
```

Figure 6-5 shows two runs of the code in Listing 6-4. The program accepts lowercase letters *y* and *n*, but not the uppercase letters *Y* and *N*. To make the program accept uppercase letters, change the conditions in the code as follows:

When you run the code, Java tells you where, on your hard drive, the `cookedBooks.txt` file should be.

CROSS REFERENCE

For more information about files and their folders, see Chapter 8.

Reading a single character

Over in Listing 5-3 from Chapter 5, the user types a word on the keyboard. The `keyboard.next` method grabs the word and places the word into a `String` variable named *password*. Everything works nicely because a `String` variable can store many characters at once, and the `next` method can read many characters at once.

But in Listing 6-4, you're not interested in reading several characters. You expect the user to type one letter — either **y** or **n**. So you don't create a `String` variable to store the user's response. Instead, you create a `char` variable — a variable that stores just one symbol at a time.

The Java API doesn't have a `nextChar` method. To read something suitable for storage in a `char` variable, you have to improvise. In Listing 6-4, the improvisation looks like this:

```
keyboard.findWithinHorizon(".", 0).charAt(0)
```

You can use this code exactly as it appears in Listing 6-4 whenever you want to read a single character.

REMEMBER

A `String` variable can contain many characters or just one. But a `String` variable that contains only one character isn't the same as a `char` variable. No matter what you put in a `String` variable, `String` variables and `char` variables have to be treated differently.

File handling in Java

In Listing 6-4, the actual file-handling statements deserve some attention. These statements involve the use of classes, objects, and methods. Many of the meaty details about these things are in other chapters, like Chapters 7 and 9. Even so, I can't do any harm by touching on some highlights right here.

So, you can find a class in the Java language API named *java.io.File*. The statement

```
File evidence = new File("cookedBooks.txt");
```


creates a new object in the computer's memory. This object, formed from the `java.io.File` class, describes everything that the program needs to know about the disk file `cookedBooks.txt`. From this point on in Listing 6-4, the variable `evidence` refers to the disk file `cookedBooks.txt`.

The `evidence` object, as an instance of the `java.io.File` class, has a `delete` method. (What can I say? It's in the API documentation.) When you call `evidence.delete`, the computer gets rid of the file for you.

Of course, you can't get rid of something that doesn't already exist. When the computer executes

```
File evidence = new File("cookedBooks.txt");
```

Java doesn't check to make sure that you have a file named `cookedBooks.txt`. To force Java to do the checking, you have a few options. The simplest is to call the `exists` method. When you call `evidence.exists()`, the method looks in the folder where Java expects to find `cookedBooks.txt`. The call `evidence.exists()` returns true if Java finds `cookedBooks.txt` inside that folder. Otherwise, the call `evidence.exists()` returns false. Here's a souped-up version of Listing 6-4, with a call to `exists` included in the code:

```
import java.io.File;
import static java.lang.System.out;
import java.util.Scanner;

public class DeleteEvidence {

  public static void main(String args[]) {
    File evidence = new File("cookedBooks.txt");
    if (evidence.exists()) {
      Scanner keyboard = new Scanner(System.in);
      char reply;

      do {
        out.print("Delete evidence? (y/n) ");
        reply =
           keyboard.findWithinHorizon(".", 0).charAt(0);
      } while (reply != 'y' && reply != 'n');

      if (reply == 'y') {
        out.println("Okay, here goes...");
        evidence.delete();
        out.println("The evidence has been deleted.");
```

TRY IT OUT

Copy the code from Listing 6-1, but with the following change:

```
        out.print("Enter an int from 1 to 10: ");
        int inputNumber = keyboard.nextInt();
        numGuesses++;

        do {
            out.println();
            out.println("Try again...");
            out.print("Enter an int from 1 to 10: ");
            inputNumber = keyboard.nextInt();
            numGuesses++;
        } while (inputNumber != randomNumber);

        out.print("You win after ");
        out.println(numGuesses + " guesses.");
```

The code in Listing 6-1 has a `while` loop, but this modified code has a `do` loop. Does this modified code work correctly? Why, or why not?


```
        } else {
          out.println("Sorry, buddy. Just asking.");
        }

        keyboard.close();
      }
    }
}
```

Variable declarations and blocks

A bunch of statements surrounded by curly braces forms a block. If you declare a variable inside a block, you generally can't use that variable outside the block. For instance, in Listing 6-4, you get an error message if you make the following change:

```
do {
   out.print("Delete evidence? (y/n) ");
   char reply = keyboard.findWithinHorizon(".", 0).charAt(0);
} while (reply != 'y' && reply != 'n');

if (reply == 'y')
```

With the declaration `char reply` inside the loop's curly braces, no use of the name `reply` makes sense anywhere outside the braces. When you try to compile this code, you get three error messages — two for the `reply` words in `while (reply != 'y' && reply != 'n')` and a third for the `if` statement's `reply`.

So in Listing 6-4, your hands are tied. The program's first real use of the `reply` variable is inside the loop. But to make that variable available after the loop, you have to declare `reply` before the loop. In this situation, you're best off declaring the `reply` variable without initializing the variable. Very interesting!

CROSS REFERENCE

To read more about variable initializations, see Chapter 4. To find out more about blocks, see Chapter 5.

All versions of Java have the three kinds of loops described in this chapter (`while` loops, `for` loops, and `do ... while` loops). But newer Java versions (namely, Java 5 and beyond) have yet another kind of loop, called an *enhanced for loop*. For a look at Java's `enhanced for` loop, see Chapter 11.

3 Working with the Big Picture: Object-Oriented Programming

IN THIS CHAPTER

- Thinking like a real object-oriented programmer
- Passing values to and from methods
- Hiding details in your object-oriented code

Chapter 7

Thinking in Terms of Classes and Objects

As a computer book author, I've been told this over and over again: I shouldn't expect people to read sections and chapters in their logical order. People jump around, picking what they need and skipping what they don't feel like reading. With that in mind, I realize that you may have skipped Chapter 1. If that's the case, please don't feel guilty. You can compensate in just 60 seconds by reading the following information, culled from Chapter 1:

> *Because Java is an object-oriented programming language, your primary goal is to describe classes and objects. A class is the idea behind a certain kind of thing. An object is a concrete instance of a class. The programmer defines a class, and from the class definition, Java makes individual objects.*

Of course, you can certainly choose to skip over the 60-second summary paragraph. If that's the case, you may want to recoup some of your losses. You can do that by reading the following two-word summary of Chapter 1:

> *Classes; objects.*

IN THIS PART . . .

Find out what classes and objects are (without bending your brain out of shape).

Find out how object-oriented programming helps you reuse existing code (saving you time and money).

Be the emperor of your own virtual world by constructing brand-new objects.

Defining a Class (What It Means to Be an Account)

What distinguishes one bank account from another? If you ask a banker this question, you hear a long sales pitch. The banker describes interest rates, fees, penalties — the whole routine. Fortunately for you, I'm not interested in all that. Instead, I want to know how my account is different from your account. After all, my account is named *Barry Burd, trading as Burd Brain Consulting,* and your account is named *Jane Q. Reader, trading as Budding Java Expert.* My account has $24.02 in it. How about yours?

When you come right down to it, the differences between one account and another can be summarized as values of variables. Maybe there's a variable named `balance`. For me, the value of `balance` is `24.02`. For you, the value of `balance` is `55.63`. The question is, when writing a computer program to deal with accounts, how do I separate my `balance` variable from your `balance` variable?

The answer is to create two separate objects. Let one `balance` variable live inside one of the objects and let the other `balance` variable live inside the other object. While you're at it, put a `name` variable and an `address` variable in each of the objects. And there you have it: two objects, and each object represents an account. More precisely, each object is an instance of the `Account` class. (See Figure 7-1.)

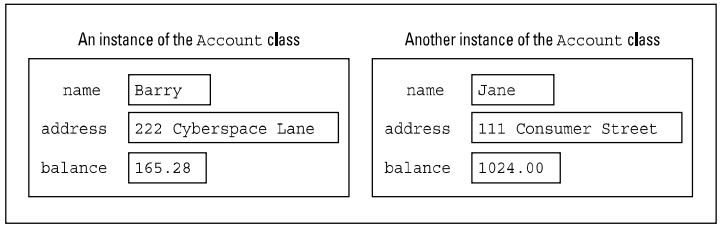

FIGURE 7-1: Two objects.

So far, so good. However, you still haven't solved the original problem. In your computer program, how do you refer to my `balance` variable, as opposed to your `balance` variable? Well, you have two objects sitting around, so maybe you have variables to refer to these two objects. Create one variable named *myAccount* and another variable named *yourAccount.* The `myAccount` variable refers to my object (my instance of the `Account` class) with all the stuff that's inside it. To refer to my balance, write

```
myAccount.balance
```


To refer to my name, write

```
myAccount.name
```

Then `yourAccount.balance` refers to the value in your object's `balance` variable, and `yourAccount.name` refers to the value of your object's `name` variable. To tell Java how much I have in my account, you can write

```
myAccount.balance = 24.02;
```

To display your name on the screen, you can write

```
out.println(yourAccount.name);
```

These ideas come together in Listings 7-1 and 7-2. Here's Listing 7-1:

LISTING 7-1: What It Means to Be an Account

```
public class Account {
    String name;
    String address;
    double balance;
}
```

The `Account` class in Listing 7-1 defines what it means to be an `Account`. In particular, Listing 7-1 tells you that each of the `Account` class's instances has three variables: `name`, `address`, and `balance`. This is consistent with the information in Figure 7-1. Java programmers have a special name for variables of this kind (variables that belong to instances of classes). Each of these variables — `name`, `address`, and `balance` — is called a *field.*

REMEMBER

A variable declared inside a class but not inside any particular method is a *field.* In Listing 7-1, the variables `name`, `address`, and `balance` are fields. Another name for a field is an *instance variable.*

If you've been grappling with the material in Chapters 4 through 6, the code for class `Account` (refer to Listing 7-1) may come as a big shock to you. Can you really define a complete Java class with only four lines of code (give or take a curly brace)? You certainly can. A class is a grouping of existing things. In the `Account` class of Listing 7-1, those existing things are two `String` values and a `double` value.

LISTING 7-2: Dealing with Account Objects

```
import static java.lang.System.out;

public class UseAccount {

    public static void main(String args[]) {
        Account myAccount;
        Account yourAccount;

        myAccount = new Account();
        yourAccount = new Account();

        myAccount.name = "Barry Burd";
        myAccount.address = "222 Cyberspace Lane";
        myAccount.balance = 24.02;

        yourAccount.name = "Jane Q. Public";
        yourAccount.address = "111 Consumer Street";
        yourAccount.balance = 55.63;

        out.print(myAccount.name);
        out.print(" (");
        out.print(myAccount.address);
        out.print(") has $");
        out.print(myAccount.balance);
        out.println();

        out.print(yourAccount.name);
        out.print(" (");
        out.print(yourAccount.address);
        out.print(") has $");
        out.print(yourAccount.balance);
    }
}
```

Taken together, the two classes — `Account` and `UseAccount` — form one complete program. The code in Listing 7-2 defines the `UseAccount` class, and the `UseAccount` class has a main method. This main method has variables of its own — `yourAccount` and `myAccount`.

In a way, the first two lines inside the main method of Listing 7-2 are misleading. Some people read `Account yourAccount` as if it's supposed to mean, "`yourAccount` is an `Account`," or "The variable `yourAccount` refers to an instance of the `Account` class." That's not really what this first line means. Instead, the line `Account yourAccount` means, "If and when I make the variable `yourAccount` refer to something, that something will be an instance of the `Account` class." So, what's the difference?

WARNING

The field declarations in Listing 7-1 have *default access*, which means that I didn't add a word before the type name `String`. The alternatives to default access are *public*, *protected*, and *private* access:

```
public String name;
protected String address;
private double balance;
```

Professional programmers shun the use of default access because default access doesn't shield a field from accidental misuse. But in my experience, you learn best when you learn about the simplest stuff first, and in Java, default access is the simplest stuff. In this book, I delay the discussion of private access until this chapter's section "Hiding Details with Accessor Methods." And I delay the discussion of protected access until Chapter 14. As you read this chapter's examples, please keep in mind that default access isn't the best thing to use in a Java program. And, if a professional programmer asks you where you learned to use default access, please lie and blame someone else's book.

Declaring variables and creating objects

A young fellow approaches me while I'm walking down the street. He tells me to print "You'll love Java!" so I print those words. If you must know, I print them with chalk on the sidewalk. But where I print the words doesn't matter. What matters is that some guy issues instructions, and I follow the instructions.

Later that day, an elderly woman sits next to me on a park bench. She says, "An account has a name, an address, and a balance." And I say, "That's fine, but what do you want me to do about it?" In response she just stares at me, so I don't do anything about her *account* pronouncement. I just sit there, she sits there, and we both do absolutely nothing.

Listing 7-1, shown earlier, is like the elderly woman. This listing defines what it means to be an `Account`, but the listing doesn't tell me to do anything with my account, or with anyone else's account. In order to do something, I need a second piece of code. I need another class — a class that contains a `main` method. Fortunately, while the woman and I sit quietly on the park bench, a young child comes by with Listing 7-2.

The difference is that simply declaring `Account yourAccount` doesn't make the `yourAccount` variable refer to an object. All the declaration does is reserve the variable name *yourAccount* so that the name can eventually refer to an instance of the `Account` class. The creation of an actual object doesn't come until later in the code, when Java executes `new Account()`.

CROSS REFERENCE

Technically, when Java executes `new Account()`, you're creating an object by calling the `Account` class's *constructor*. I have more to say about that in Chapter 9.

When Java executes the assignment `yourAccount = new Account()`, Java creates a new object (a new instance of the `Account` class) and makes the variable `yourAccount` refer to that new object. (The equal sign makes the variable refer to the new object.) Figure 7-2 illustrates the situation.

To test the claim that I made in the last few paragraphs, I added an extra line to the code of Listing 7-2. I tried to print `yourAccount.name` after declaring `yourAccount` but before calling `new Account()`:

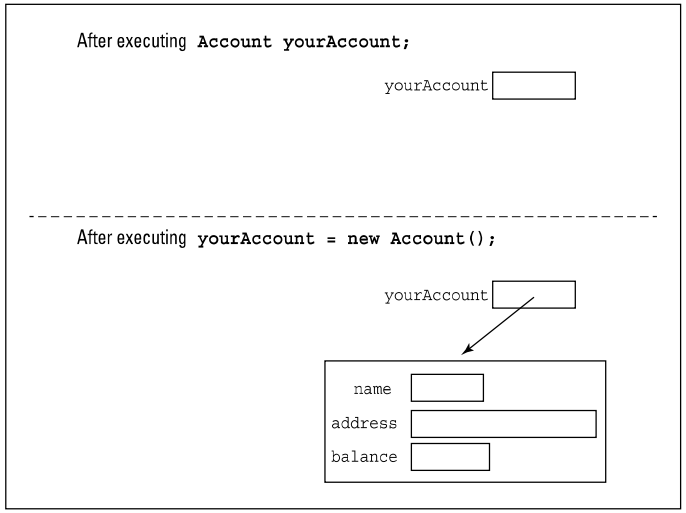

FIGURE 7-2: Before and after a constructor is called.

```
Account myAccount;
    Account yourAccount;

    out.println(yourAccount.name);

    myAccount = new Account();
    yourAccount = new Account();
```


When I tried to compile the new code, I got this error message: `variable yourAccount might not have been initialized`. That settles it. Before you do `new Account()`, you can't print the name variable of an object because an object doesn't exist.

REMEMBER

When a variable has a reference type, simply declaring the variable isn't enough. You don't get an object until you call a constructor and use the keyword `new`.

For information about reference types, see Chapter 4.

Initializing a variable

In Chapter 4, I announce that you can initialize a primitive type variable as part of the variable's declaration.

```
int weightOfAPerson = 150;
```

You can do the same thing with reference type variables, such as `myAccount` and `yourAccount` in Listing 7-2. You can combine the first four lines in the listing's main method into just two lines, like this:

```
Account myAccount = new Account();
Account yourAccount = new Account();
```

If you combine lines this way, you automatically avoid the `variable might not have been initialized` error that I describe in the preceding section. Sometimes you find a situation in which you can't initialize a variable. But when you can initialize, it's usually a plus.

Using an object's fields

After you've bitten off and chewed the `main` method's first four lines, the rest of the code in the earlier Listing 7-2 is sensible and straightforward. You have three lines that put values in the `myAccount` object's fields, three lines that put values in the `yourAccount` object's fields, and four lines that do some printing. Figure 7-3 shows the program's output.

```
Barry Burd (222 Cyberspace Lane) has $24.02
Jane Q. Public (111 Consumer Street) has $55.63
```

FIGURE 7-3: Running the code in Listings 7-1 and 7-2.

One program; several classes

Each program in Chapters 3 to 6 consists of a single class. That's great for a book's introductory chapters. But in real life, a typical program consists of hundreds or even thousands of classes. The program that spans Listings 7-1 and 7-2 consists of two classes. Sure, having two classes isn't like having thousands of classes, but it's a step in that direction.

In practice, most programmers put each class in a file of its own. When you create a program, such as the one in Listings 7-1 and 7-2, you create two files on your computer's hard drive. Therefore, when you download this section's example from the web, you get two separate files — `Account.java` and `UseAccount.java`.

For information about running a program consisting of more than one `.java` file in Eclipse, NetBeans, and IntelliJ IDEA, visit this book's website (`www.allmycode.com/JavaForDummies`).

Public classes

The first line of Listing 7-1 is

```
public class Account {
```

The `Account` class is public. A public class is available for use by all other classes. For example, if you write an ATMController program in some remote corner of cyberspace, then your ATMController program can contain code, such as `myAccount.balance = 24.02`, making use of the `Account` class declared in Listing 7-1. (Of course, your code has to know where in cyberspace I've stored the code in Listing 7-1, but that's another story.)

Listing 7-2 contains the code `myAccount.balance = 24.02`. You might say to yourself, "The `Account` class has to be public because another class (the code in Listing 7-2) uses the `Account` class." Unfortunately, the real lowdown about public classes is a bit more complicated. In fact, when the planets align themselves correctly, one class can make use of another class's code, even though the other class isn't public. (I cover the proper aligning of planets in Chapter 14.)

The dirty secret in this chapter's code is that declaring certain classes to be public simply makes me feel good. Yes, programmers do certain things to feel good. In Listing 7-1, my esthetic sense of goodness comes from the fact that an `Account` class is useful to many other programmers. When I create a class that declares something useful and nameable — an `Account`, an `Engine`, a `Customer`, a `BrainWave`, a `Headache`, or a `SevenLayerCake` class — I declare the class to be public.

The `UseAccount` class in Listing 7-2 is also public. When a class contains a `main` method, Java programmers tend to make the class public without thinking too much about who uses the class. So even if no other class makes use of my `main` method, I declare the `UseAccount` class to be public. Most of the classes in this book contain `main` methods, so most of the classes in this book are public.

WARNING

When you declare a class to be public, you must declare the class in a file whose name is exactly the same as the name of the class (but with the `.java` extension added). For example, if you declare `public class MyImportantCode`, you must put the class's code in a file named `MyImportantCode.java`, with uppercase letters `M`, `I`, and `C` and all other letters lowercase. This file-naming rule has an important consequence: If your code declares two public classes, your code must consist of at least two `.java` files. In other words, you can't declare two `public` classes in one `.java` file.

For more news about the word *public* and other such words, see Chapter 14.

TRY IT OUT

In this section, I create an `Account` class. You can create classes too.

- An `Organization` has a name (such as *XYZ Company*), an annual revenue (such as $100,000.00), and a `boolean` value indicating whether the organization is or is not a profit-making organization. Companies that manufacture and sell products are generally profit-making organizations; groups that provide aid to victims of natural disasters are generally not profit-making organizations.

  Declare your own `Organization` class. Declare another class that creates organizations and displays information about those organizations.

- A product for sale in a food store has several characteristics: a type of food (peach slices), a weight (500 grams), a cost ($1.83), a number of servings (4), and a number of calories per serving (70).

  Declare a `FoodProduct` class. Declare another class that creates `FoodProduct` instances and displays information about those instances.

Defining a Method within a Class (Displaying an Account)

Imagine a table containing the information about two accounts. (If you have trouble imagining such a thing, just look at Table 7-1.)

TABLE 7-1 Without Object-Oriented Programming

Name	Address	Balance
Barry Burd	222 Cyberspace Lane	24.02
Jane Q. Public	111 Consumer Street	55.63

In Table 7-1, each account has three things — a name, an address, and a balance. That's how things were done before object-oriented programming came along. But object-oriented programming involved a big shift in thinking. With object-oriented programming, each account can have a name, an address, a balance, and a way of being displayed.

In object-oriented programming, each object has its own built-in functionality. An account knows how to display itself. A string can tell you whether it has the same characters inside it as another string has. A `PrintStream` instance, such as `System.out`, knows how to do `println`. In object-oriented programming, each object has its own methods. These methods are little subprograms that you can call to have an object do things to (or for) itself.

And why is this a good idea? It's good because you're making pieces of data take responsibility for themselves. With object-oriented programming, all the functionality that's associated with an account is collected inside the code for the `Account` class. Everything you have to know about a string is located in the file `String.java`. Anything having to do with year numbers (whether they have two or four digits, for instance) is handled right inside the `Year` class. Therefore, if anybody has problems with your `Account` class or your `Year` class, he or she knows just where to look for all the code. That's great!

Imagine an enhanced account table. In this new table, each object has built-in functionality. Each account knows how to display itself on the screen. Each row of the table has its own copy of a `display` method. Of course, you don't need much imagination to picture this table. I just happen to have a table you can look at. It's Table 7-2.

TABLE 7-2 The Object-Oriented Way

Name	Address	Balance	Display
Barry Burd	222 Cyberspace Lane	24.02	`out.print ...`
Jane Q. Public	111 Consumer Street	55.63	`out.print ...`

An account that displays itself

In Table 7-2, each account object has four things — a name, an address, a balance, and a way of displaying itself on the screen. After you make the jump to object-oriented thinking, you'll never turn back. Listings 7-3 and 7-4 show programs that implement the ideas in Table 7-2.

LISTING 7-3: An Account Displays Itself

```
import static java.lang.System.out;

public class Account {
    String name;
    String address;
    double balance;

    public void display() {
        out.print(name);
        out.print(" (");
        out.print(address);
        out.print(") has $");
        out.print(balance);
    }
}
```

LISTING 7-4: Using the Improved Account Class

```
public class UseAccount {

    public static void main(String args[]) {
        Account myAccount = new Account();
        Account yourAccount = new Account();

        myAccount.name = "Barry Burd";
        myAccount.address = "222 Cyberspace Lane";
        myAccount.balance = 24.02;

        yourAccount.name = "Jane Q. Public";
        yourAccount.address = "111 Consumer Street";
        yourAccount.balance = 55.63;

        myAccount.display();
        System.out.println();
        yourAccount.display();
    }
}
```

A run of the code in Listings 7-3 and 7-4 looks just like a run for Listings 7-1 and 7-2. You can see the action earlier, in Figure 7-3.

In Listing 7-3, the `Account` class has four things in it: a name, an address, a balance, and a display method. These things match up with the four columns in Table 7-2. So each instance of the `Account` class has a name, an address, a balance, and a way of displaying itself. The way you call these things is nice and uniform. To refer to the name stored in `myAccount`, you write

```
myAccount.name
```

To get `myAccount` to display itself on the screen, you write

```
myAccount.display()
```

The only difference is the parentheses.


REMEMBER

When you call a method, you put parentheses after the method's name.

The display method's header

Look again at Listings 7-3 and 7-4. A call to the display method is inside the `UseAccount` class's `main` method, but the declaration of the `display` method is up in the `Account` class. The declaration has a header and a body. (See Chapter 3.) The header has three words and some parentheses:

- **The word *public* serves roughly the same purpose as the word *public* in Listing 7-1.** Roughly speaking, any code can contain a call to a public method, even if the calling code and the public method belong to two different classes. In this section's example, the decision to make the `display` method public is a matter of taste. Normally, when I create a method that's useful in a wide variety of applications, I declare the method to be public.
- **The word *void* tells Java that when the `display` method is called, the `display` method doesn't return anything to the place that called it.** To see a method that does return something to the place that called it, see the next section.
- **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. The `display` method in Listing 7-3 looks strange

because the parentheses in the method's header have nothing inside them. This nothingness indicates that no information is passed to the `display` method when you call it. For a meatier example, see the next section.


CROSS REFERENCE

Listing 7-3 contains the `display` method's declaration, and Listing 7-4 contains a call to the `display` method. Although Listings 7-3 and 7-4 contain different classes, both uses of `public` in Listing 7-3 are optional. To find out why, check out Chapter 14.


TRY IT OUT

In the previous section, you create `Organization` and `FoodProduct` classes. Add `display` methods to both of these classes and create separate classes that make use of these `display` methods.

Sending Values to and from Methods (Calculating Interest)

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 *goToTheSupermarketAndBuySome*. 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. Instead of getting 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 what you're going to ask your pal to buy.

And what happens when your friend returns from the supermarket? "Here's the bread you asked me to buy," says your friend. By carrying out your wishes, your friend returns something to you. You make a method call, and the method returns information (or a loaf of bread).

The thing returned to you is called the method's *return value*. The general type of thing that is returned to you is called the method's *return type*. These concepts are made more concrete in Listings 7-5 and 7-6.

LISTING 7-5: An Account That Calculates Its Own Interest

```
import static java.lang.System.out;

public class Account {
    String name;
    String address;
    double balance;

    public void display() {
        out.print(name);
        out.print(" (");
        out.print(address);
        out.print(") has $");
        out.print(balance);
    }

    public double getInterest(double percentageRate) {
        return balance * percentageRate / 100.00;
    }
}
```

LISTING 7-6: Calculating Interest

```
import static java.lang.System.out;

public class UseAccount {

    public static void main(String args[]) {
        Account myAccount = new Account();
        Account yourAccount = new Account();

        myAccount.name = "Barry Burd";
        myAccount.address = "222 Cyberspace Lane";
        myAccount.balance = 24.02;

        yourAccount.name = "Jane Q. Public";
        yourAccount.address = "111 Consumer Street";
        yourAccount.balance = 55.63;

        myAccount.display();
        out.print(" plus $");
        out.print(myAccount.getInterest(5.00));
        out.println(" interest ");

        yourAccount.display();

        double yourInterestRate = 7.00;
        out.print(" plus $");
        double yourInterestAmount = yourAccount.getInterest(yourInterestRate);
        out.print(yourInterestAmount);
        out.println(" interest ");
    }
}
```

Figure 7-4 shows the output of the code in Listings 7-5 and 7-6. In Listing 7-5, the `Account` class has a `getInterest` method. This `getInterest` method is called twice from the `main` method in Listing 7-6. The actual account balances and interest rates are different each time.

FIGURE 7-4: Running the code in Listings 7-5 and 7-6.

```
Barry Burd (222 Cyberspace Lane) has $24.02 plus $1.2009999999999998 interest
Jane Q. Public (111 Consumer Street) has $55.63 plus $3.8941000000000003 interest
```

- **In the first call, the balance is 24.02, and the interest rate is 5.00.** The first call, `myAccount.getInterest(5.00)`, refers to the `myAccount` object and to the values stored in the `myAccount` object's fields. (See Figure 7-5.) When this call is made, the expression `balance * percentageRate / 100.00` stands for 24.02 * 5.00 / 100.00.
- **In the second call, the balance is 55.63, and the interest rate is 7.00.** In the `main` method, just before this second call is made, the variable `yourInterestRate` is assigned the value `7.00`. The call itself, `yourAccount.getInterest(yourInterestRate)`, refers to the `yourAccount` object and to the values stored in the `yourAccount` object's fields. (Again, see Figure 7-5.) So, when the call is made, the expression `balance * percentageRate / 100.00` stands for 55.63 * 7.00 / 100.00.

By the way, the `main` method in Listing 7-6 contains two calls to `getInterest`. One call has the literal `5.00` in its parameter list; the other call has the variable `yourInterestRate` in its parameter list. Why does one call use a literal and the other call use a variable? No reason. I just want to show you that you can do it either way.

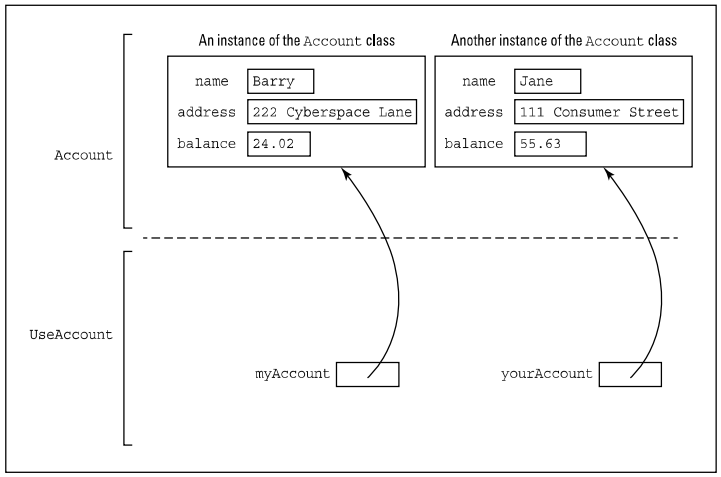

FIGURE 7-5: My account and your account.

Passing a value to a method

Take a look at the `getInterest` method's header. (As you read the explanation in the next few bullets, you can follow some of the ideas visually with the diagram in Figure 7-6.)

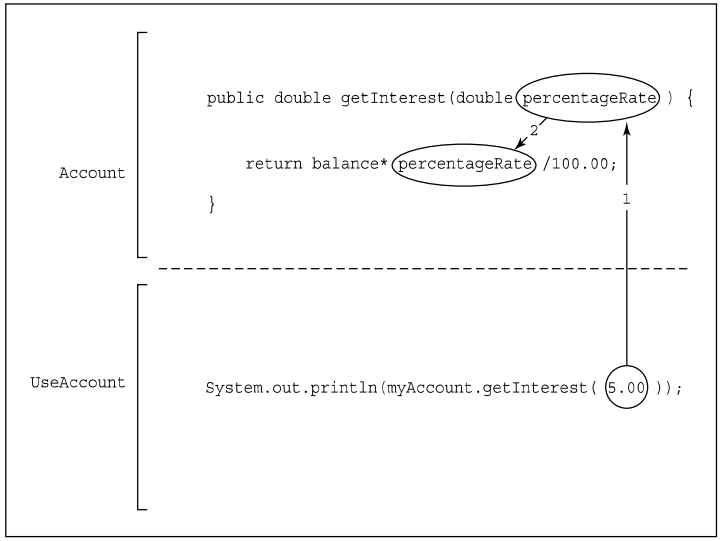

FIGURE 7-6: Passing a value to a method.

- **The word *double* tells Java that when the `getInterest` method is called, the `getInterest` method returns a double value back to the place that called it.** The statement in the `getInterest` method's body confirms this. The statement says `return balance * percentageRate / 100.00`, and the expression `balance * percentageRate / 100.00` has type `double`. (That's because all the things in the expression — `balance`, `percentageRate`, and `100.00` — have type `double`.)

  When the `getInterest` method is called, the `return` statement calculates `balance * percentageRate / 100.00` and hands the calculation's result back to the code that called the method.

- **The word *getInterest* is the method's name.** That's the name you use to call the method when you're writing the code for the `UseAccount` class.

- **The parentheses contain all the things that you 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 `getInterest` method's header says that the `getInterest` method takes one piece of information and that piece of information must be of type `double`:

```
public double getInterest(double  percentageRate)
```

  Sure enough, if you look at the first call to `getInterest` (down in the `useAccount` class's main method), that call has the number `5.00` in it. And `5.00` is a `double` literal. When I call `getInterest`, I'm giving the method a value of type `double`.

  If you don't remember what a literal is, see Chapter 4.

  The same story holds true for the second call to `getInterest`. Down near the bottom of Listing 7-6, I call `getInterest` and feed the variable `yourInterestRate` to the method in its parameter list. Luckily for me, I declared `yourInterestRate` to be of type `double` just a few lines before that.

CROSS REFERENCE

When you run the code in Listings 7-5 and 7-6, the flow of action isn't from top to bottom. The action goes from `main` to `getInterest`, and then back to `main`, and then back to `getInterest`, and, finally, back to `main` again. Figure 7-7 shows the whole business.

```
public class Account {
    Yada, yada, yada...

    double getInterest(double percentageRate) {
        return balance * percentageRate / 100.00;
    }
}


public class UseAccount {

    public static void main(String args[]) {
        Account myAccount = new  Account();
        Account yourAccount = new Account();

        myAccount.name = "Barry Burd";
        myAccount.address = "222 Cyberspace Lane";
        myAccount.balance = 24.02;

        yourAccount.name = "Jane Q. Public";
        yourAccount.address = "111 Consumer Street";
        yourAccount.balance = 55.63;

        myAccount.display();

        out.print(" plus $");

        out.print( myAccount.getInterest(5.00) );

        out.println(" interest ");

        yourAccount.display();

        double yourInterestRate = 7.00;
        out.print(" plus $");
        double yourInterestAmount =

            yourAccount.getInterest(yourInterestRate) ;

        out.print(yourInterestAmount);
        out.println(" interest ");
    }
}
```


FIGURE 7-7: The flow of control in Listings 7-5 and 7-6.

Returning a value from the getInterest method

When the `getInterest` method is called, the method executes the one statement that's in the method's body: a `return` statement. The `return` statement computes the value of `balance * percentageRate / 100.00`. If `balance` happens to be `24.02`, and `percentageRate` is `5.00`, the value of the expression is `1.201` — around $1.20. (Because the computer works exclusively with 0s and 1s, Java gets this number wrong by an ever-so-tiny amount. Java gets 1.2009999999999998. That's just something that humans have to live with.)

Anyway, after this value is calculated, Java executes the `return`, which sends the value back to the place in `main` where `getInterest` was called. At that point in the process, the entire method call — `myAccount.getInterest(5.00)` — takes on the value 1.2009999999999998. The call itself is inside a `println`:

```
out.println(myAccount.getInterest(5.00));
```

So the `println` ends up with the following meaning:

```
out.println(1.2009999999999998);
```

The whole process, in which a value is passed back to the method call, is illustrated in Figure 7-8.

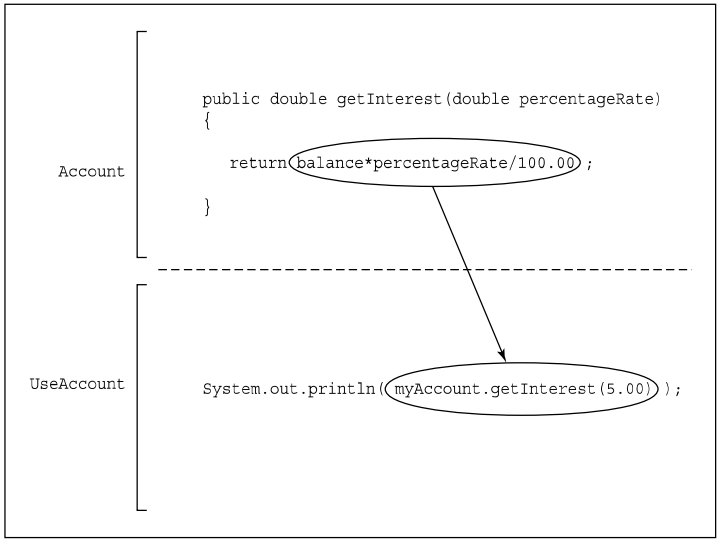

FIGURE 7-8: A method call is an expression with a value.

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.

You might use the `Account` class in Listing 7-5 to solve a real problem. You'd call the `Account` class's `display` and `getInterest` methods in the course of an actual banking application. But the `UseAccount` class in Listing 7-6 is artificial. The `UseAccount` code creates some fake account data and then calls some `Account` class methods to convince you that the `Account` class's code works correctly. (You don't

seriously think that a bank has depositors named "Jane Q. Public" and "Barry Burd," do you?) The `UseAccount` class in Listing 7-6 is a *test case* — a short-lived class whose sole purpose is to test another class's code. Like the code in Listing 7-6, each test case in this book is an ordinary class — a free-form class containing its own `main` method. Free-form classes are okay, but they're not optimal. Java developers have something better — a more disciplined way of writing test cases. The "better way" is called *JUnit*, and it's described on this book's website (`www.allmycode.com/JavaForDummies`).


TRY IT OUT

» In previous sections, you create your own `Organization` class. Add a method to the class that computes the amount of tax the organization pays. A profit-making organization pays 10 percent of its revenue in tax, but a nonprofit organization pays only 2 percent of its revenue in tax.

Make a separate class that creates two or three organizations and displays information about each organization, including the amount of tax the organization pays.

» In previous sections, you create your own `FoodProduct` class. Add methods to the class to compute the cost per 100 grams, the cost per serving, and the total number of calories in the product.

Make a separate class that creates two or three products and displays information about each product.

Making Numbers Look Good

Looking at Figure 7-4 again, you may be concerned that the interest on my account is only $1.2009999999999998. Seemingly, the bank is cheating me out of 2 hundred-trillionths of a cent. I should go straight to the bank and demand my fair interest. Maybe you and I should go together. We'll kick up some fur at that old bank and bust this scam right open. If my guess is correct, this is part of a big salami scam. In a *salami scam*, someone shaves tiny amounts off millions of accounts. People don't notice their tiny little losses, but the person doing the shaving collects enough for a quick escape to Barbados (or for a whole truckload of salami).

Wait a minute! What about you? In Listing 7-6, you have `yourAccount`. And in Figure 7-4, your name is Jane Q. Public. Nothing is motivating you to come with me to the bank. Checking Figure 7-4 again, I see that you're way ahead of the game. According to my calculations, the program overpays you by 3 hundred-trillionths of a cent. Between the two of us, we're ahead by a hundred-trillionth of a cent. What gives?

Well, because computers use 0s (zeros) and 1s and don't have an infinite amount of space to do calculations, such inaccuracies as the ones shown in Figure 7-4 are normal. The quickest solution is to display the inaccurate numbers in a more sensible fashion. You can round the numbers and display only two digits beyond the decimal point, and some handy tools from Java's API (application programming interface) can help. Listing 7-7 shows the code, and Figure 7-9 displays the pleasant result.

LISTING 7-7: Making Your Numbers Look Right

```
import static java.lang.System.out;

public class UseAccount {

    public static void main(String args[]) {
        Account myAccount = new Account();
        Account yourAccount = new Account();

        myAccount.balance = 24.02;
        yourAccount.balance = 55.63;

        double myInterest = myAccount.getInterest(5.00);
        double yourInterest = yourAccount.getInterest(7.00);

        out.printf("$%4.2f\n", myInterest);
        out.printf("$%5.2f\n", myInterest);
        out.printf("$%.2f\n",  myInterest);
        out.printf("$%3.2f\n", myInterest);
        out.printf("$%.2f $%.2f", myInterest, yourInterest);
    }
}
```

FIGURE 7-9: Numbers that look like dollar amounts.

```
$1.20
$ 1.20
$1.20
$1.20
$1.20 $3.89
```

The inaccurate numbers in Figure 7-4 come from the computer's use of 0s and 1s. A mythical computer whose circuits were wired to use digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 wouldn't suffer from the same inaccuracies. So, to make things better, Java provides its own special way around the computer's inaccurate calculations. Java's API has a class named `BigDecimal` — a class that bypasses the computer's strange 0s and 1s and uses ordinary decimal digits to perform arithmetic calculations. For more information, visit this book's website (`www.allmycode.com/JavaForDummies`).

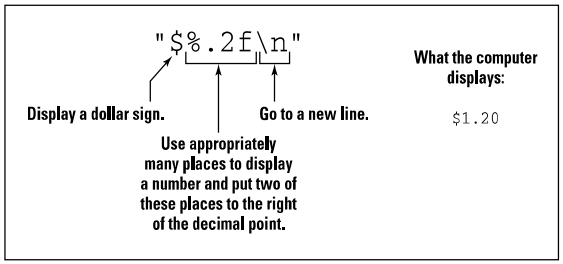

FIGURE 7-12: Displaying a value without specifying the exact number of places.

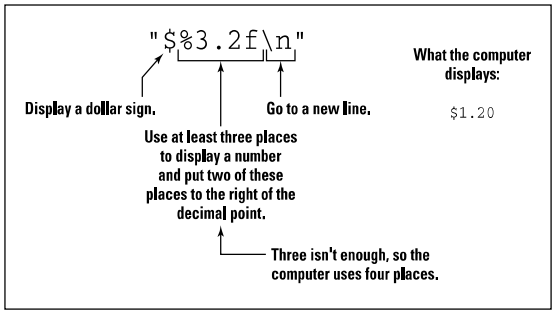

FIGURE 7-13: Specifying too few places to display a value.

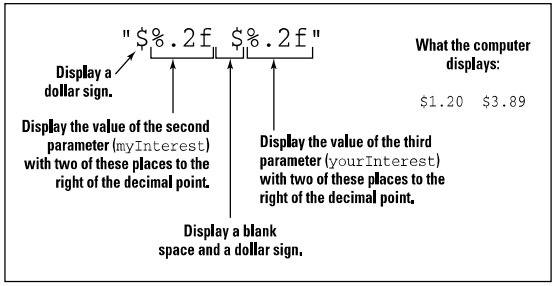

FIGURE 7-14: Displaying more than one value with a format string.

For more examples using the `printf` method and its format strings, see Chapters 8 and 9. For a complete list of options associated with the `printf` method's format string, see the `java.util.Formatter` page of Java's API documentation at `https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html`.

REMEMBER

The format string in a `printf` call doesn't change the way a number is stored internally for calculations. All the format string does is create a nice-looking bunch of digit characters that can be displayed on your screen.

Listing 7-7 uses a handy method named `printf`. When you call `printf`, you always put at least two parameters inside the call's parentheses:

- **The first parameter is a *format string*.**

  The format string uses funny-looking codes to describe exactly how the other parameters are displayed.

- **All the other parameters (after the first) are values to be displayed.**

Look at the last `printf` call of Listing 7-7. The first parameter's format string has two placeholders for numbers. The first placeholder (`%.2f`) describes the display of `myInterest`. The second placeholder (another `%.2f`) describes the display of `yourInterest`. To find out exactly how these format strings work, see Figures 7-10 through 7-14.

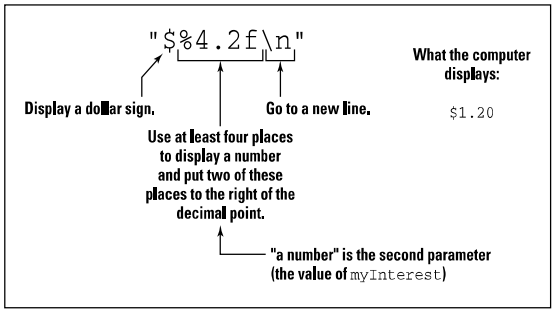

FIGURE 7-10: Using a format string.

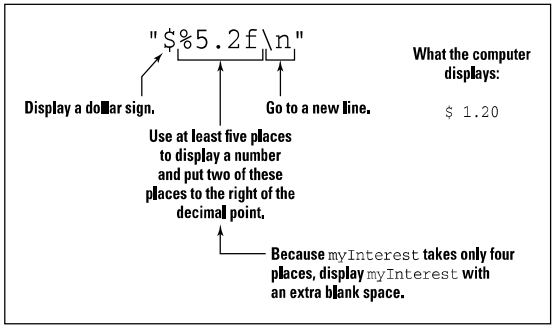

FIGURE 7-11: Adding extra places to display a value.

To do this, look for clues in the `java.util.Formatter` page of Java's API documentation at `https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html`.

Hiding Details with Accessor Methods

Put down this book and put on your hat. You've been such a loyal reader that I'm taking you out to lunch!

I have just one problem. I'm a bit short on cash. Would you mind if, on the way to lunch, we stopped at an automatic teller machine and picked up a few bucks? Also, we have to use your account. My account is a little low.

Fortunately, the teller machine is easy to use. Just step right up and enter your PIN. After you enter your PIN, the machine asks which of several variable names you want to use for your current balance. You have a choice of `balance324`, `myBal`, `currentBalance`, `b$`, `BALANCE`, `asj999`, or `conStanTinople`. Having selected a variable name, you're ready to select a memory location for the variable's value. You can select any number between 022FFF and 0555AA. (Those numbers are in hexadecimal format.) After you configure the teller machine's software, you can easily get your cash. You did bring a screwdriver, didn't you?

Good programming

When it comes to good computer programming practice, one word stands out above all others: *simplicity.* When you're writing complicated code, the last thing you want is to deal with somebody else's misnamed variables, convoluted solutions to problems, or clever, last-minute kludges. You want a clean interface that makes you solve your own problems and no one else's.

In the automatic teller machine scenario that I describe earlier, the big problem is that the machine's design forces you to worry about other people's concerns. When you should be thinking about getting money for lunch, you're thinking instead about variables and storage locations. Sure, someone has to work out the teller machine's engineering problems, but the banking customer isn't the person.


REMEMBER

This section is about safety, not security. Safe code keeps you from making accidental programming errors. Secure code (a completely different story) keeps malicious hackers from doing intentional damage.


CROSS REFERENCE

The `printf` method is good for formatting values of any kind — ordinary numbers, hexadecimal numbers, dates, strings of characters, and some other strange values. That's why I show it to you in this section. But when you work with currency amounts, this section's `printf` tricks are fairly primitive. For some better ways to deal with currency amounts (such as the interest amounts in this section's example), see Chapter 11.


TRY IT OUT

Here's a Java "un-program." It's not a real Java program, because I've masked some of the characters in the code. I replaced these characters with underscores (_):

```
import static java.lang.System.out;

public class Main {

    public static void main(String[] args) {
        out.printf("%s%_%s", ">>", 7, "<<\n");
        out.printf("%s%___%s", ">>", 7, "<<\n");
        out.printf("%s%____%s", ">>", 7, "<<\n");
        out.printf("%s%____%s", ">>", 7, "<<\n");
        out.printf("%s%__%s", ">>", 7, "<<\n");
        out.printf("%s%__%s", ">>", -7, "<<\n");
        out.printf("%s%__%s", ">>", -7, "<<\n");
        out.printf("%s%_____%s", ">>", 7.0, "<<\n");
        out.printf("%s%_%s", ">>", "Hello", "<<\n");
        out.printf("%s%_%s", ">>", 'x', "<<\n");
        out.printf("%s%_%s", ">>", 'x', "<<\n");
    }
}
```

Replace the underscores so that this program produces the following output:

```
>>7<<
>>        7<<
>>7        <<
>>0000000007<<
>>+7<<
>>-7<<
>>(7)<<
>>   7.00000<<
>>HELLO<<
>>x<<
>>X<<
```

So, everything connected with every aspect of a computer program has to be simple, right? Well, no. That's not right. Sometimes, to make things simple in the long run, you have to do lots of preparatory work up front. The people who built the automated teller machine worked hard to make sure that the machine is consumer-proof. The machine's interface, with its screen messages and buttons, makes the machine a very complicated, but carefully designed, device.

The point is that making things look simple takes some planning. In the case of object-oriented programming, one of the ways to make things look simple is to prevent code outside a class from directly using fields defined inside the class. Take a peek at the code in Listing 7-1. You're working at a company that has just spent $10 million for the code in the `Account` class. (That's more than a million-and-a-half per line!) Now your job is to write the `UseAccount` class. You would like to write

```
myAccount.name = "Barry Burd";
```

but doing so would be getting you too far inside the guts of the `Account` class. After all, people who use an automatic teller machine aren't allowed to program the machine's variables. They can't use the machine's keypad to type the statement

```
balanceOnAccount29872865457 = balanceOnAccount29872865457 + 1000000.00;
```

Instead, they push buttons that do the job in an orderly manner. That's how a programmer achieves safety and simplicity.

To keep things nice and orderly, you need to change the `Account` class from Listing 7-1 by outlawing such statements as the following:

```
myAccount.name = "Barry Burd";
```

and

```
out.print(yourAccount.balance);
```

Of course, this poses a problem. You're the person who's writing the code for the `UseAccount` class. If you can't write `myAccount.name` or `yourAccount.balance`, how will you accomplish anything at all? The answer lies in things called *accessor methods*. Listings 7-8 and 7-9 demonstrate these methods.

LISTING 7-8: Hide Those Fields

```
public class Account {
    private String name;
    private String address;
    private double balance;

    public void setName(String n) {
        name = n;
    }

    public String getName() {
        return name;
    }

    public void setAddress(String a) {
        address = a;
    }

    public String getAddress() {
        return address;
    }

    public void setBalance(double b) {
        balance = b;
    }

    public double getBalance() {
        return balance;
    }
}
```

LISTING 7-9: Calling Accessor Methods

```
import static java.lang.System.out;

public class UseAccount {

    public static void main(String args[]) {
        Account myAccount = new Account();
        Account yourAccount = new Account();

        myAccount.setName("Barry Burd");
        myAccount.setAddress("222 Cyberspace Lane");
        myAccount.setBalance(24.02);
```

(continued)

LISTING 7-9: *(continued)*

```
        yourAccount.setName("Jane Q. Public");
        yourAccount.setAddress("111 Consumer Street");
        yourAccount.setBalance(55.63);

        out.print(myAccount.getName());
        out.print(" (");
        out.print(myAccount.getAddress());
        out.print(") has $");
        out.print(myAccount.getBalance());
        out.println();

        out.print(yourAccount.getName());
        out.print(" (");
        out.print(yourAccount.getAddress());
        out.print(") has $");
        out.print(yourAccount.getBalance());
    }
}
```

A run of the code in Listings 7-8 and 7-9 looks no different from a run of Listings 7-1 and 7-2. Either program's run is shown earlier, in Figure 7-3. The big difference is that in Listing 7-8, the `Account` class enforces the carefully controlled use of its `name`, `address`, and `balance` fields.

Public lives and private dreams: Making a field inaccessible

Notice the addition of the word *private* in front of each of the `Account` class's field declarations. The word *private* is a Java keyword. When a field is declared private, no code outside of the class can make direct reference to that field. So if you put `myAccount.name = "Barry Burd"` in the `UseAccount` class of Listing 7-9, you get an error message such as `name has private access in Account`.

Instead of referencing `myAccount.name`, the `UseAccount` programmer must call method `myAccount.setName` or method `myAccount.getName`. These methods, `setName` and `getName`, are called *accessor* methods because they provide access to the `Account` class's `name` field. (Actually, the term *accessor method* isn't formally a part of the Java programming language. It's just the term that people use for methods that do this sort of thing.) To zoom in even more, `setName` is called a *setter* method, and `getName` is called a *getter* method. (I bet you won't forget that terminology!)


TIP

With many IDEs, you don't have to type your own accessor methods. First, you type a field declaration like `private String name`. Then, in your IDE's menu bar, you choose Source➪Generate Getters and Setters, or choose Code➪Insert Code➪Setter or some mix of those commands. After you make all your choices, the IDE creates accessor methods and adds them to your code.

Notice that all the setter and getter methods in Listing 7-8 are declared to be public. This ensures that anyone from anywhere can call these two methods. The idea here is that manipulating the actual fields from outside the `Account` code is impossible, but you can easily reach the approved setter and getter methods for using those fields.

Think again about the automatic teller machine. Someone using the ATM can't type a command that directly changes the value in his or her account's `balance` field, but the procedure for depositing a million-dollar check is easy to follow. The people who build the teller machines know that if the check-depositing procedure is complicated, plenty of customers will mess it up royally. So that's the story — make impossible anything that people shouldn't do and make sure that the tasks people should be doing are easy.


TIP

Nothing about having setter and getter methods is sacred. You don't have to write any setter and getter methods that you're not going to use. For instance, in Listing 7-8, I can omit the declaration of method `getAddress`, and everything still works. The only problem if I do this is that anyone else who wants to use my `Account` class and retrieve the address of an existing account is up a creek.


TIP

When you create a method to set the value in a `balance` field, you don't have to name your method `setBalance`. You can name it `tunaFish` or whatever you like. The trouble is that the `set`*`Fieldname`* convention (with lowercase letters in `set` and an uppercase letter to start the *`Fieldname`* part) is an established stylistic convention in the world of Java programming. If you don't follow the convention, you confuse the kumquats out of other Java programmers. If your integrated development environment has drag-and-drop GUI design capability, you may temporarily lose that capability. (For a word about drag-and-drop GUI design, see Chapters 2 and 16.)


REMEMBER

When you call a setter method, you feed it a value of the type that's being set. That's why, in Listing 7-9, you call `yourAccount.setBalance(55.63)` with a parameter of type `double`. In contrast, when you call a getter method, you usually don't feed any values to the method. That's why, in Listing 7-9, you call `yourAccount.getBalance()` with an empty parameter list. Occasionally, you may want to get and set a value with a single statement. To add a dollar to your account's existing balance, you write `yourAccount.setBalance(yourAccount.getBalance() + 1.00)`.

LISTING 7-10: Your First DummiesFrame Example

```
import com.allmycode.dummiesframe.DummiesFrame;

public class GuessingGame {

    public static void main(String[] args) {
        DummiesFrame frame = new DummiesFrame("Greet Me!");
        frame.addRow("Your first name");
        frame.go();
    }

    public static String calculate(String firstName) {
        return "Hello, " + firstName + "!";
    }
}
```

FIGURE 7-15: The code in Listing 7-10 starts running.

FIGURE 7-16: The user fills in the fields.

FIGURE 7-17: The user clicks the button.

Enforcing rules with accessor methods

Go back to Listing 7-8 and take a quick look at the `setName` method. Imagine putting the method's assignment statement inside an `if` statement.

```
public void setName(String n) {
    if (!n.equals("")) {
        name = n;
    }
}
```

Now, if the programmer in charge of the `UseAccount` class writes `myAccount.setName("")`, the call to `setName` doesn't have any effect. Furthermore, because the name field is private, the following statement is illegal in the `UseAccount` class:

```
myAccount.name = "";
```

Of course, a call such as `myAccount.setName("Joe Schmoe")` still works because `"Joe Schmoe"` doesn't equal the empty string `""`.

That's cool. With a private field and an accessor method, you can prevent someone from assigning the empty string to an account's `name` field. With more elaborate `if` statements, you can enforce any rules you want.

TRY IT OUT

In previous sections, you create your own `Organization` and `FoodProduct` classes. In those classes, replace the default access fields with private fields. Create getter and setter methods for those fields. In the setter methods, add code to ensure that the `String` values aren't empty and that numeric values aren't negative.

Barry's Own GUI Class

You may be getting tired of the bland, text-based programs that litter this book's pages. You may want something a bit flashier — something with text fields and buttons. Well, I've got some examples for you!

I've created a class that I call `DummiesFrame`. When you import my `DummiesFrame` class, you can create a simple graphical user interface (GUI) application with very little effort.

Listing 7-10 uses my `DummiesFrame` class, and Figures 7-15 to 7-17 show you the results.

Here's a blow-by-blow description of the lines in Listing 7-10:

- The first line

```
import com.allmycode.dummiesframe.DummiesFrame;
```

  makes the name `DummiesFrame` available to the rest of the code in the listing.

- Inside the `main` method, the statement

```
DummiesFrame frame = new DummiesFrame("Greet Me!");
```

  creates an instance of my `DummiesFrame` class and makes the variable name `frame` refer to that instance. A `DummiesFrame` object appears as a window on the user's screen. In this example, the text on the window's title bar is *Greet Me!*

- The next statement is a call to the `frame` object's `addRow` method:

```
frame.addRow("Your first name");
```

  This call puts a row on the face of the application's window. The row consists of a label (whose text is *Your first name*), an empty text field, and a red X mark indicating that the user hasn't yet typed anything useful into the field. (Refer to Figure 7-15.)

- A call to the `frame` object's `go` method

```
frame.go();
```

  makes the app's window appear on the screen.

- The header of the `calculate` method

```
public static String calculate(String firstName) {
```

  tells Java two important things:

  - The `calculate` method returns a value of type `String`.
  - Java should expect the user to type a `String` value in the text field, and whatever the user types will become the `firstName` parameter's value.

  To use my `DummiesFrame` class, your code must have a method named `calculate`, and the `calculate` method must obey certain rules:

  - The `calculate` method's header must start with the words `public static`.
  - The method may return any Java type: `String`, `int`, `double`, or whatever. (That's actually not a rule; it's an opportunity!)
  - The `calculate` method must have the same number of parameters as there are rows in the application's window.

Listing 7-10 has only one `addRow` method call, so the window in Figure 7-10 has only one row (not including the *Submit* button), and so the `calculate` method has only one parameter.

When the user starts typing text into the window's text field, the red X mark turns into a green check mark. (Refer to Figure 7-16.) The green check mark indicates that the user has typed a value of the expected type (in this example, a `String` value) into the text field.

- When the user clicks the button, Java executes the `calculate` method. The expression in the method's `return` statement

```
return "Hello, " + firstName + "!";
```

  tells Java what to display at the bottom of the window. (Refer to Figure 7-17.) In this example, the user types *Barry* in the one and only text field, so the value of `firstName` is `"Barry"`, and the calculate method returns the string `"Hello, Barry!"` (Ah! The perks of being a *For Dummies* author!)

Using my `DummiesFrame` class, you can build a simple GUI application with only ten lines of code.

REMEMBER

The `DummiesFrame` class isn't built into the Java API so, in order to run the code in Listing 7-10, my `DummiesFrame.java` file must be part of your project. When you download the code from this book's website (`www.allmycode.com/JavaForDummies`), you get a folder named `07-10` containing both the Listing 7-10 code and my `DummiesFrame.java` code. But if you create your own project containing the Listing 7-10, you have to add my `DummiesFrame.java` file manually. The way you do this depends on which IDE you use. One way or another, my `DummiesFrame` class is in a package named `com.allmycode.dummiesframe`, so the `DummiesFrame.java` file must be in a directory named `dummiesframe`, which is inside another directory named `allmycode`, which is inside yet another directory named `com`. For more about packages, see Chapters 9 and 14.

TECHNICAL STUFF

To keep things simple, I include the `DummiesFrame.java` file in the `07-10` folder that you download from this book's website. But, really, is that the best way to add my own code to your project? In Chapter 1, I describe files with the `.class` extension, and the role that those files play in the running of a Java program. Instead of handing you my `DummiesFrame.java` file, I should be putting only a `DummiesFrame.class` file in the download. And, on some other occasion, if I have to give you hundreds of `.class` files, I should bundle them all into one big archive file. Java has a name for a big file that encodes many smaller `.class` files. It's called a JAR file and it has the `.jar` extension. In a real-life application, if you're preparing your code for other people to use as part of their own applications, a JAR file is definitely the way to go.

My `DummiesFrame` class isn't exclusively for greetings and salutations. Listing 7-11 uses `DummiesFrame` to do arithmetic.

```
  public static void main(String[] args) {
    DummiesFrame frame = new DummiesFrame("Guessing Game");
    frame.addRow("Enter an int from 1 to 10");
    frame.setButtonText("Submit your guess");
    frame.go();
  }

  public static String calculate(int inputNumber) {
    Random random = new Random();
    int randomNumber = random.nextInt(10) + 1;

    if (inputNumber == randomNumber) {
      return "You win.";
    } else {
      return "You lose. The random number was " + randomNumber + ".";
    }
  }
}
```

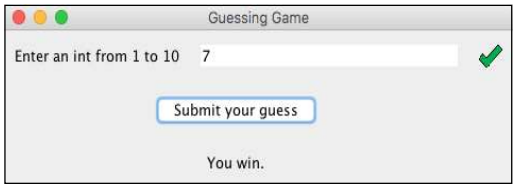

FIGURE 7-19: I win!

In Listing 7-13, I use this chapter's `Account` class alongside the `DummiesFrame` class. I could get the same results without creating an `Account` instance, but I want to show you how classes can cooperate to form a complete program. A run of the code is in Figure 7-20.

LISTING 7-13: Using the Account Class

```
import com.allmycode.dummiesframe.DummiesFrame;

public class UseAccount {

    public static void main(String args[]) {
        DummiesFrame frame = new DummiesFrame("Display an Account");
        frame.addRow("Full name");
        frame.addRow("Address");
        frame.addRow("Balance");
        frame.setButtonText("Display");
```

(continued)

LISTING 7-11: A Really Simple Calculator

```
import com.allmycode.dummiesframe.DummiesFrame;

public class Addition {

    public static void main(String[] args) {
        DummiesFrame frame = new DummiesFrame("Adding Machine");
        frame.addRow("First number");
        frame.addRow("Second number");
        frame.setButtonText("Sum");
        frame.go();
    }

    public static int calculate(int firstNumber, int secondNumber) {
        return firstNumber + secondNumber;
    }
}
```

The window in Figure 7-18 has two rows because Listing 7-11 has two `addRow` calls and the listing's `calculate` method has two parameters. In addition, Listing 7-11 calls the `frame` object's `setButtonText` method. So, in Figure 7-18, the text on the face of the button isn't the default word *Submit*.

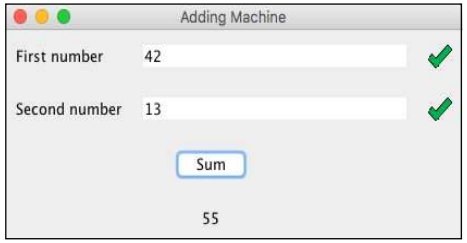

FIGURE 7-18: Look! The code in Listing 7-11 actually works!

Listing 7-12 contains a GUI version of the Guessing Game application from Chapter 5, and Figure 7-19 shows the game in action.

LISTING 7-12: I'm Thinking of a Number

```
import java.util.Random;
import com.allmycode.dummiesframe.DummiesFrame;

public class GuessingGame {
```

IN THIS CHAPTER

- Adding new life to old code
- Tweaking your code
- Making changes without spending a fortune

Chapter 8

Saving Time and Money: Reusing Existing Code

Once upon a time, there was a beautiful princess. When the princess turned 25 (the optimal age for strength, good looks, and fine moral character), her kind, old father brought her a gift in a lovely golden box. Anxious to know what was in the box, the princess ripped off the golden wrapping paper.

When the box was finally opened, the princess was thrilled. To her surprise, her father had given her what she had always wanted: a computer program that always ran correctly. The program did everything the princess wanted, and did it all exactly the way she wanted it to be done. The princess was happy, and so was her father.

Even as time marched on, the computer program never failed. For years on end, the princess changed her needs, expected more out of life, made increasing demands, expanded her career, reached for more and more fulfillment, juggled the desires of her husband and her kids, stretched the budget, and sought peace within her soul. Through all of this, the program remained her steady, faithful companion.

As the princess grew old, the program became old along with her. One evening, as she sat by the fireside, she posed a daunting question to the program: "How do you do it?" she asked. "How do you manage to keep giving the right answers, time after time, year after year?"

LISTING 7-13: *(continued)*

```
        frame.go();
    }

    public static String calculate(String name, String address,
                                                    double balance) {
        Account myAccount = new Account();

        myAccount.setName(name);
        myAccount.setAddress(address);
        myAccount.setBalance(balance);
        return myAccount.getName() + " (" + myAccount.getAddress() +
                ") has $" + myAccount.getBalance();
    }
}
```

TRY IT OUT

Use the `DummiesFrame` class to create two GUI programs.

- A window has text fields for an organization's name, annual revenue, and status (profit-making or not profit-making). When the user clicks a button, the window displays the amount of tax the organization pays.

  A profit-making organization pays 10 percent of its revenue in tax; a nonprofit organization pays 2 percent of its revenue in tax.

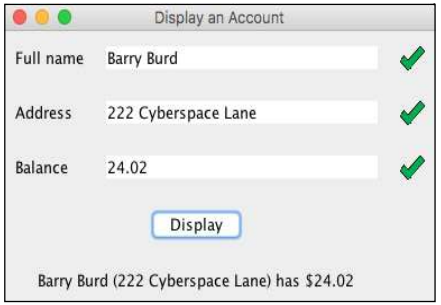

FIGURE 7-20: I'm rich.

- A window has text fields for a product's type of food, weight, cost, number of servings, and number of calories per serving. When the user clicks a button, the window displays the cost per 100 grams, the cost per serving, and the total number of calories in the product.

"Clean living," replied the program. "I swim 20 apps each day, I take C++ to Word off viruses, I avoid hogarithmic algorithms, I link Java in moderation, I say GNU to bugs, I don't smoke to back up, and I never byte off more than I can queue."

Needless to say, the princess was stunned.

Defining a Class (What It Means to Be an Employee)

Wouldn't it be nice if every piece of software did just what you wanted it to do? In an ideal world, you could buy a program, make it work right away, plug it seamlessly into new situations, and update it easily whenever your needs change. Unfortunately, software of this kind doesn't exist. (*Nothing* of this kind exists.) The truth is that no matter what you want to do, you can find software that does some of it, but not all of it.

This is one of the reasons why object-oriented programming has been successful. For years, companies were buying prewritten code, only to discover that the code didn't do what they wanted it to do. So, what did the companies do about it? They started messing with the code. Their programmers dug deep into the program files, changed variable names, moved subprograms around, reworked formulas, and generally made the code worse. The reality was that if a program didn't already do what you wanted it to do (even if it did something ever so close to what you wanted), you could never improve the situation by mucking around inside the code. The best option was always to chuck the whole program (expensive as that was) and start all over again. What a sad state of affairs!

With object-oriented programming, a big change has come about. At its heart, an object-oriented program is made to be modified. With correctly written software, you can take advantage of features that are already built-in, add new features of your own, and override features that don't suit your needs. And the best part is that the changes you make are clean. No clawing and digging into other people's brittle program code. Instead, you make nice, orderly additions and modifications without touching the existing code's internal logic. It's the ideal solution.

The last word on employees

When you write an object-oriented program, you start by thinking about the data. You're writing about accounts. So what's an account? You're writing code to handle button clicks. So what's a button? You're writing a program to send payroll checks to employees. What's an employee?

In this chapter's first example, an employee is someone with a name and a job title. Sure, employees have other characteristics, but for now I stick to the basics. The code in Listing 8-1 defines what it means to be an employee.

LISTING 8-1: What Is an Employee?

```
import static java.lang.System.out;

public class Employee {
    private String name;
    private String jobTitle;

    public void setName(String nameIn) {
        name = nameIn;
    }

    public String getName() {
        return name;
    }

    public void setJobTitle(String jobTitleIn) {
        jobTitle = jobTitleIn;
    }

    public String getJobTitle() {
        return jobTitle;
    }

    public void cutCheck(double amountPaid) {
        out.printf("Pay to the order of %s ", name);
        out.printf("(%s) ***$", jobTitle);
        out.printf("%,.2f\n", amountPaid);
    }
}
```

According to Listing 8-1, each employee has seven features. Two of these features are fairly simple: Each employee has a name and a job title. (In Listing 8-1, the `Employee` class has a `name` field and a `jobTitle` field.)

And what else does an employee have? Each employee has four methods to handle the values of the employee's name and job title. These methods are `setName`, `getName`, `setJobTitle`, and `getJobTitle`. I explain methods like these (*accessor* methods) in Chapter 7.

On top of all of that, each employee has a `cutCheck` method. The idea is that the method that writes payroll checks has to belong to one class or another. Because most of the information in the payroll check is customized for a particular employee, you may as well put the `cutCheck` method inside the `Employee` class.

For details about the `printf` calls in the `cutCheck` method, see the section "Cutting a check," later in this chapter.

Putting your class to good use

The `Employee` class in Listing 8-1 has no `main` method, so there's no starting point for executing code. To fix this deficiency, the programmer writes a separate program with a `main` method and uses that program to create `Employee` instances. Listing 8-2 shows a class with a `main` method — one that puts the code in Listing 8-1 to the test.

LISTING 8-2: Writing Payroll Checks

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

public class DoPayroll {

    public static void main(String args[]) throws IOException {
        Scanner diskScanner = new Scanner(new File("EmployeeInfo.txt"));

        for (int empNum = 1; empNum <= 3; empNum++) {
            payOneEmployee(diskScanner);
        }
        diskScanner.close();
    }

    static void payOneEmployee(Scanner aScanner) {
        Employee anEmployee = new Employee();

        anEmployee.setName(aScanner.nextLine());
        anEmployee.setJobTitle(aScanner.nextLine());
        anEmployee.cutCheck(aScanner.nextDouble());
        aScanner.nextLine();
    }
}
```


WHERE ON EARTH DO YOU LIVE?

Grouping separators vary from one country to another. This makes a big difference when you try to read `double` values using Java's `Scanner` class. To see what I mean, have a serious look at the following JShell session.

```
jshell> import java.util.Scanner

jshell> import java.util.Locale

jshell> Scanner keyboard = new Scanner(System.in)
keyboard ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ... \E]
   [infinity string=\Q8\E]

jshell> keyboard.nextDouble()
1000.00
$4 ==> 1000.0

jshell> Locale.setDefault(Locale.FRANCE)

jshell> keyboard = new Scanner(System.in)
keyboard ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ... \E]
   [infinity string=\Q8\E]

jshell> keyboard.nextDouble()
1000,00
$7 ==> 1000.0

jshell> keyboard.nextDouble()
1000.00
|  java.util.InputMismatchException thrown:
|        at Scanner.throwFor (Scanner.java:860)
|        at Scanner.next (Scanner.java:1497)
|        at Scanner.nextDouble (Scanner.java:2467)
|        at (#8:1)

jshell>
```

I conducted this session on a computer in the United States. The country of origin is relevant because, in response to the first `keyboard.nextDouble()` call, I type `1000.00` (with a period before the last two zeros) and Java accepts this as meaning "one thousand."

But then, in the JShell session, I call `Locale.setDefault(Locale.FRANCE)`, which tells Java to behave as if my computer is in France. When I create another `Scanner` instance

(continued)

(continued)

and call `keyboard.nextDouble()` again, Java accepts `1000,00` (with a comma before the last two zeros) as an expression meaning *mille* (French for "one thousand"). What's more, Java no longer accepts the period in `1000.00`. When I type `1000.00` (with a period) I get an `InputMismatchException`.

By default, your computer's `Scanner` instance wants you to input `double` numbers the way you normally type them in your country. If you type numbers according to another country's convention, you get an `InputMismatchException`. So, when you run the code in Listing 8-2, the numbers in your `EmployeeInfo.txt` file must use your country's format.

This brings me to the running of the code in Listing 8-2. The `EmployeeInfo.txt` file that you download from this book's website starts with the following three lines:

```
Barry Burd
CEO
5000.00
```

That last number `5000.00` has a period in it, so if your country prefers a comma in place of my United States period, you get an `InputMismatchException`. In response to this, you have two choices:

- In the downloaded `EmployeeInfo.txt` file, change the periods to commas.
- In the code of Listing 8-2, add the statement `Locale.setDefault(Locale.US)` before the `diskScanner` declaration.

And finally, if you want your output to look like your own country's numbers, you can do it with Java's `Formatter` class. Add something like this to your code:

```
out.print(
    new java.util.Formatter().format(java.util.Locale.FRANCE, "%,.2f", 1000.00));
```

For all the details, see the API (Application Programming Interface) documentation for Java's `Formatter` class (`https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html` and `Locale` class (`https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html`).

To run the code in Listing 8-2, your hard drive must contain a file named `EmployeeInfo.txt`. Fortunately, the stuff that you download from this book's website (`www.allmycode.com/JavaForDummies`) comes with an `EmployeeInfo.txt` file. You can import the downloaded material into any of the three most

popular Java IDEs (Eclipse, NetBeans, or IntelliJ IDEA). If you import into Eclipse, you get a project named `08-01`. That project typically lives on your hard drive in a folder named `/Users/`*`your-user-name`*`/workspace/08-01`. Directly inside that folder, you have a file named `EmployeeInfo.txt`.

CROSS REFERENCE

For more words of wisdom about files on your hard drive, see the "Working with Disk Files (a Brief Detour)" section in this chapter.

The `DoPayroll` class in Listing 8-2 has two methods. One of the methods, `main`, calls the other method, `payOneEmployee`, three times. Each time around, the `payOneEmployee` method gets stuff from the `EmployeeInfo.txt` file and feeds this stuff to the `Employee` class's methods.

Here's how the variable name *anEmployee* is reused and recycled:

- The first time that `payOneEmployee` is called, the statement `anEmployee = new Employee()` makes `anEmployee` refer to a new object.
- The second time that `payOneEmployee` is called, the computer executes the same statement again. This second execution creates a new incarnation of the `anEmployee` variable that refers to a brand-new object.
- The third time around, all the same stuff happens again. A new `anEmployee` variable ends up referring to a third object.

The whole story is pictured in Figure 8-1.

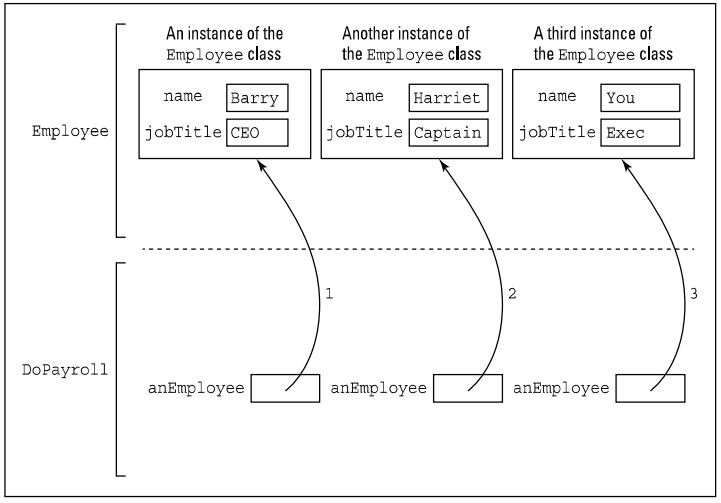

FIGURE 8-1: Three calls to the `payOneEmployee` method.

Working with Disk Files (a Brief Detour)

In previous chapters, programs read characters from the computer's keyboard. But the code in Listing 8-2 reads characters from a specific file. The file (named `EmployeeInfo.txt`) lives on your computer's hard drive.

This `EmployeeInfo.txt` file is like a word processing document. The file can contain letters, digits, and other characters. But unlike a word processing document, the `EmployeeInfo.txt` file contains no formatting — no italics, no bold, no font sizes, nothing of that kind.

The `EmployeeInfo.txt` file contains only ordinary characters — the kinds of keystrokes that you type while you play a guessing game from Chapters 5 and 6. Of course, getting guesses from a user's keyboard and reading employee data from a disk file aren't exactly the same. In a guessing game, the program displays prompts, such as `Enter an int from 1 to 10`. The game program conducts a back-and-forth dialogue with the person sitting at the keyboard. In contrast, Listing 8-2 has no dialogue. This `DoPayroll` program reads characters from a hard drive and doesn't prompt or interact with anyone.

Most of this chapter is about code reuse. But Listing 8-2 stumbles upon an important idea — an idea that's not directly related to code reuse. Unlike the examples in previous chapters, Listing 8-2 reads data from a stored disk file. So, in the following sections, I take a short side trip to explore disk files.

Storing data in a file

The code in Listing 8-2 doesn't run unless you have some employee data sitting in a file. Listing 8-2 says that this file is `EmployeeInfo.txt`. So, before running the code of Listing 8-2, I created a small `EmployeeInfo.txt` file. The file is shown in Figure 8-3; refer to Figure 8-2 for the resulting output.

```
Barry Burd
CEO
5000.00
Harriet Ritter
Captain
7000.00
Your Name Here
Honorary Exec of the Day
10000.00
```

FIGURE 8-3: An `EmployeeInfo.txt` file.

When you visit this book's website (`www.allmycode.com/JavaForDummies`) and you download the book's code listings, you get a copy of the `EmployeeInfo.txt` file.

TRY IT OUT

There are always interesting things for you to try:

- A `PlaceToLive` has an address, a number of bedrooms, and an area (in square feet or square meters). Write the `PlaceToLive` class's code. Write code for a separate class named `DisplayThePlaces`. Your `DisplayThePlaces` class creates a few `PlaceToLive` instances by assigning values to their `address`, `numberOfBedrooms`, and `area` fields. The `DisplayThePlaces` class also reads (from the keyboard) the cost of living in each place. For each place, your code displays the cost per square foot (or square meter) and the cost per bedroom.
- Use your new `PlaceToLive` class and my `DummiesFrame` class (from Chapter 7) to create a GUI application. The GUI application takes information about a place to live and displays the place's cost per square foot (or meter) and the cost per bedroom.

Cutting a check

Listing 8-1 has three `printf` calls. Each `printf` call has a format string (like `"(%s) ***$"`) and a variable (like `jobTitle`). Each format string has a placeholder (like `%s`) that determines where and how the variable's value is displayed.

For example, in the second `printf` call, the format string has a `%s` placeholder. This `%s` holds a place for the `jobTitle` variable's value. According to Java's rules, the notation `%s` always holds a place for a string and, sure enough, the variable `jobTitle` is declared to be of type `String` in Listing 8-1. Parentheses and some other characters surround the `%s` placeholder, so parentheses surround each job title in the program's output. (See Figure 8-2.)

```
Pay to the order of Barry Burd (CEO) ***$5,000.00
Pay to the order of Harriet Ritter (Captain) ***$7,000.00
Pay to the order of Your Name Here (Honorary Exec of the Day) ***$10,000.00
```

FIGURE 8-2: Everybody gets paid.

Back in Listing 8-1, notice the comma inside the `%,.2f` placeholder. The comma tells the program to use *grouping separators.* That's why, in Figure 8-2, you see `$5,000.00`, `$7,000.00`, and `$10,000.00` instead of `$5000.00`, `$7000.00`, and `$10000.00`.

```
        //Some code goes here

        scannerName.close();
    }
}
```

You want to read data from a file. You start by imagining that you're reading from the keyboard. Put the usual `Scanner` and `next` codes into your program. Then add some extra items from the Listing 8-2 pattern:

- Add two new `import` declarations — one for `java.io.File` and another for `java.io.IOException`.
- Type **throws IOException** in your method's header.
- Type **new File(" ")** in your call to `new Scanner`.
- Take a file that's already on your hard drive. Type that filename inside the quotation marks.
- Take the word that you use for the name of your scanner. Reuse that word in calls to `next`, `nextInt`, `nextDouble`, and so on.
- Take the word that you use for the name of your scanner. Reuse that word in a call to `close`.

Occasionally, copying and pasting code can get you into trouble. Maybe you're writing a program that doesn't fit the simple Listing 8-2 pattern. You need to tweak the pattern a bit. But to tweak the pattern, you need to understand some of the ideas behind the pattern.

That's how the next section comes to your rescue. It covers some of these ideas.


TECHNICAL STUFF

This paragraph is actually a confession. In almost every computer programming language, input from a disk file is a nasty business. There's no such thing as a simple `INPUT` command. You normally have to set up a connection between the code and the disk device, prepare for possible trouble reading from the device, do your reading, convert the characters you read into the type of value that you want and, finally, break your connection with the disk device. It's a big mess. That's why, in this book, I rely on Java's `Scanner` class. The `Scanner` class makes input relatively painless. But, I admit, professional Java programmers hardly ever use the `Scanner` class to do input. Instead, they use something called a `BufferedReader` or classes in the `java.nio` package. If you're not content with my use of the `Scanner` class and you want to see Listing 8-2 translated into a `BufferedReader` program, visit this book's website (`www.allmycode.com/JavaForDummies`).


WARNING

To keep Listing 8-2 simple, I insist that, when you type the characters in Figure 8-3, you finish up by typing `10000.00` and then pressing Enter. (Look again at Figure 8-3 and notice how the cursor is at the start of a brand-new line.) If you forget to finish by pressing Enter, the code in Listing 8-2 will crash when you try to run it.


WARNING

Grouping separators vary from one country to another. The file shown in Figure 8-3 works on a computer configured in the United States where *5000.00* means "five thousand." But the file doesn't work on a computer that's configured in what I call a "comma country" — a country where *5000,00* means "five thousand." If you live in a comma country, be sure to read this chapter's "Where on Earth do you live?" sidebar.

This book's website (`www.allmycode.com/JavaForDummies`) has tips for readers who need to create data files. This includes instructions for Windows, Linux, and Macintosh environments.

Copying and pasting code

In almost any computer programming language, reading data from a file can be tricky. You add extra lines of code to tell the computer what to do. Sometimes you can copy and paste these lines from other peoples' code. For example, you can follow the pattern in Listing 8-2:

```
/*
 * The pattern in Listing 8-2
 */
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

class SomeClassName {

    public static void main(String args[]) throws IOException {

        Scanner scannerName = new Scanner(new File("SomeFileName"));

        //Some code goes here

        scannerName.nextInt();
        scannerName.nextDouble();
        scannerName.next();
        scannerName.nextLine();
```

Reading from a file

In previous chapters, programs read characters from the computer's keyboard. These programs use things like `Scanner`, `System.in`, and `nextDouble` — things defined in Java's API. The `DoPayroll` program in Listing 8-2 puts a new spin on this story. Rather than read characters from the keyboard, the program reads characters from the `EmployeeInfo.txt` file. The file lives on your computer's hard drive.

To read characters from a file, you use some of the same things that help you read characters from the keyboard. You use `Scanner`, `nextDouble`, and other goodies. But in addition to these goodies, you have a few extra hurdles to jump. Here's a list:

- **You need a new File object.** To be more precise, you need a new instance of the API's `File` class. You get this new instance with code like

```
new File("EmployeeInfo.txt")
```

  The stuff in quotation marks is the name of a file — a file on your computer's hard drive. The file contains characters like those shown previously in Figure 8-3.

  At this point, the terminology makes mountains out of molehills. Sure, I use the phrases *new File object* and *new File instance,* but all you're doing is making `new File("EmployeeInfo.txt")` stand for a file on your hard drive. After you shove `new File("EmployeeInfo.txt")` into `new Scanner`,

```
Scanner diskScanner = new Scanner(new File("EmployeeInfo.txt"));
```

  you can forget all about the `new File` business. From that point on in the code, `diskScanner` stands for the `EmployeeInfo.txt` filename on your computer's hard drive. (The name `diskScanner` stands for a file on your hard drive just as, in previous examples, the name `keyboard` stands for those buttons you press day in and day out.)


REMEMBER

  Creating a `new File` object in Listing 8-2 is like creating a `new Employee` object later in the same listing. It's also like creating a `new Account` object in the examples of Chapter 7. The only difference is that the `Employee` and `Account` classes are defined in this book's examples. The `File` class is defined in Java's API.


WARNING

  When you connect to a disk file with `new Scanner`, don't forget the `new File` part. If you write `new Scanner("EmployeeInfo.txt")` without `new File`, the compiler won't mind. (You don't get any warnings or error messages before you run the code.) But when you run the code, you don't get anything like the results that you expect to get.

- **You must refer to the File class by its full name: `java.io.File`.** You can do this with an import declaration like the one in Listing 8-2. Alternatively, you can clutter your code with a statement like

```
Scanner diskScanner = new Scanner(new java.io.File("EmployeeInfo.txt"));
```

- **You need a throws IOException clause.** Lots of things can go wrong when your program connects to `EmployeeInfo.txt`. For one thing, your hard drive may not have a file named `EmployeeInfo.txt`. For another, the file `EmployeeInfo.txt` may be in the wrong directory. To brace for this kind of calamity, the Java programming language takes certain precautions. The language insists that when a disk file is involved, you acknowledge the possible dangers of calling `new Scanner`.

  You can acknowledge the hazards in several possible ways, but the simplest way is to use a `throws` clause. In Listing 8-2, the `main` method's header ends with the words *throws IOException.* By adding these two words, you appease the Java compiler. It's as if you're saying "I know that calling `new Scanner` can lead to problems. You don't have to remind me." And, sure enough, adding `throws IOException` to your `main` method keeps the compiler from complaining. (Without this `throws` clause, you get an `unreported exception` error message.)


CROSS REFERENCE

  For the full story on Java exceptions, read Chapter 13. In the meantime, add `throws IOException` to the header of any method that calls `new Scanner(new File( ....`

- **You must refer to the `IOException` class by its full name: `java.io.IOException`.**

  You can do this with an `import` declaration like the one in Listing 8-2. Alternatively, you can enlarge the `main` method's `throws` clause:

```
public static void main(String args[])throws java.io.IOException {
```

- **You must pass the file scanner's name to the `payOneEmployee` method.**

  In Listing 7-5 in Chapter 7, the `getInterest` method has a parameter named *percentageRate.* Whenever you call the `getInterest` method, you hand an extra, up-to-date piece of information to the method. (You hand a number — an interest rate — to the method. Figure 7-7 illustrates the idea.)

  The same thing happens in Listing 8-2. The `payOneEmployee` method has a parameter named *aScanner.* Whenever you call the `payOneEmployee` method, you hand an extra, up-to-date piece of information to the method. (You hand a scanner — a reference to a disk file — to the method.)

file in the wrong directory.) To diagnose this problem, add the following code to Listing 8-2:

```
File employeeInfo = new File("EmployeeInfo.txt");
System.out.println("Looking for " + employeeInfo.getCanonicalPath());
```

When you run the code, Java tells you where, on your hard drive, the `EmployeeInfo.txt` file should be.

Adding directory names to your filenames

You can specify a file's exact location in your Java code. Code like `new File("C:\\Users\\bburd\\workspace\\08-01\\EmployeeInfo.txt")` looks really ugly, but it works.

In the preceding paragraph, did you notice the double backslashes in "`C:\\Users\\bburd\\workspace ...`"? If you're a Windows user, you'd be tempted to write `C:\Users\bburd\workspace ...` with single backslashes. But in Java, the single backslash has its own, special meaning. (For example, back in Listing 7-7, `\n` means to go to the next line.) So, in Java, to indicate a backslash inside a quoted string, you use a double backslash instead.


TECHNICAL STUFF

Macintosh and Linux users might find comfort in the fact that their path separator, `/`, has no special meaning in a Java string. On a Mac, the code `new File("/Users/bburd/workspace/08-01/EmployeeInfo.txt")` is as normal as breathing. (Well, it's almost that normal!) But Mac users and Linux wonks shouldn't claim superiority too quickly. Lines such as `new File("/Users/bburd/workspace ...` work in Windows as well. In Windows, you can use either a slash (`/`) or a backslash (`\`) as the path name separator. At the Windows command prompt, I can type `cd c:/users\bburd` to get to my home directory.


TIP

If you know where your Java program looks for files, you can worm your way from that place to the directory of your choice. Assume, for the moment, that the code in Listing 8-2 normally looks for the `EmployeeInfo.txt` file in a directory named `08-01`. As an experiment, go to the `08-01` directory and create a new subdirectory named *dataFiles*. Then move my `EmployeeInfo.txt` file to the new `dataFiles` directory. To read numbers and words from the file that you moved, modify Listing 8-2 with the code `new File("dataFiles\\EmployeeInfo.txt")` or `new File("dataFiles/EmployeeInfo.txt")`.

You may wonder why the `payOneEmployee` method needs a parameter. After all, in Listing 8-2, the `payOneEmployee` method always reads data from the same file. Why bother informing this method, each time you call it, that the disk file is still the `EmployeeInfo.txt` file?

Well, there are plenty of ways to shuffle the code in Listing 8-2. Some ways don't involve a parameter. But the way that this example has arranged things, you have two separate methods: a `main` method and a `payOneEmployee` method. You create a scanner once inside the `main` method and then use the scanner three times — once inside each call to the `payOneEmployee` method.

Anything you define inside a method is like a private joke that's known only to the code inside that method. So the `diskScanner` that you define inside the `main` method isn't automatically known inside the `payOneEmployee` method. To make the `payOneEmployee` method aware of the disk file, you pass `diskScanner` from the `main` method to the `payOneEmployee` method.


CROSS REFERENCE

To read more about things that you declare inside (and outside) of methods, see Chapter 10.

Who moved my file?

When you download the code from this book's website (`www.allmycode.com/JavaForDummies`), you'll find files named `Employee.java` and `DoPayroll.java` — the code in Listings 8-1 and 8-2. You'll also find a file named `EmployeeInfo.txt`. That's good because, if Java can't find the `EmployeeInfo.txt` file, the whole project doesn't run properly. Instead, you get a `FileNotFoundException`.

In general, when you get a `FileNotFoundException`, some file that your program needs isn't available to it. This is an easy mistake to make. It can be frustrating because, to you, a file such as `EmployeeInfo.txt` may look like it's available to your program. But remember: Computers are stupid. If you make a tiny mistake, the computer can't read between the lines for you. So, if your `EmployeeInfo.txt` file isn't in the right directory on your hard drive or the filename is spelled incorrectly, the computer chokes when it tries to run your code.

Sometimes you know darn well that an `EmployeeInfo.txt` (or *whatever.xyz*) file exists on your hard drive. But when you run your program, you still get a mean-looking `FileNotFoundException`. When this happens, the file is usually in the wrong directory on your hard drive. (Of course, it depends on your point of view. Maybe the file is in the right directory, but your Java program is looking for the

Reading a line at a time

In Listing 8-2, the `payOneEmployee` method illustrates some useful tricks for reading data. In particular, every scanner that you create has a `nextLine` method. (You might not use this `nextLine` method, but the method is available nonetheless.) When you call a scanner's `nextLine` method, the method grabs everything up to the end of the current line of text. In Listing 8-2, a call to `nextLine` can read a whole line from the `EmployeeInfo.txt` file. (In another program, a scanner's `nextLine` call may read everything the user types on the keyboard up to the pressing of the Enter key.)

Notice my careful choice of words: `nextLine` reads everything "up to the end of the current line." Unfortunately, what it means to read up to the end of the current line isn't always what you think it means. Intermingling `nextInt`, `nextDouble`, and `nextLine` calls can be messy. You have to watch what you're doing and check your program's output carefully.

To understand all of this, you need to be painfully aware of a data file's line breaks. Think of a line break as an extra character, stuck between one line of text and the next. Then imagine that calling `nextLine` means to read everything up to and including the next line break.

Now take a look at Figure 8-4:

- If one call to `nextLine` reads `Barry Burd[LineBreak]`, the subsequent call to `nextLine` reads `CEO[LineBreak]`.
- If one call to `nextDouble` reads the number 5000.00, the subsequent call to `nextLine` reads the `[LineBreak]` that comes immediately after the number 5000.00. (That's all the `nextLine` reads — a `[LineBreak]` and nothing more.)
- If a call to `nextLine` reads the `[LineBreak]` after the number 5000.00, the subsequent call to `nextLine` reads `Harriet Ritter[LineBreak]`.

So, after reading the number 5000.00, you need *two* calls to `nextLine` in order to scoop up the name *Harriet Ritter*. The mistake that I usually make is to forget the first of those two calls.

WARNING

Look again at the file in Figure 8-3. For this section's code to work correctly, you must have a line break after the last `10000.00`. If you don't, a final call to `nextLine` makes your program crash and burn. The error message reads `NoSuchElementException: No line found`.

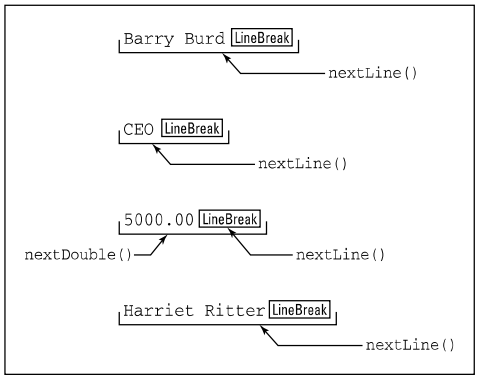

FIGURE 8-4: Calling nextDouble and nextLine.

TECHNICAL STUFF

I'm always surprised by the number of quirks that I find in each programming language's scanning methods. For example, the first `nextLine` that reads from the file in Figure 8-3 devours `Barry Burd[LineBreak]` from the file. But that `nextLine` call delivers `Barry Burd` (with no line break) to the running code. So `nextLine` looks for a line break, and then `nextLine` loses the line break. Yes, this is a subtle point. And no, this subtle point hardly ever causes problems for anyone.

TECHNICAL STUFF

If this business about `nextDouble` and `nextLine` confuses you, please don't put the blame on Java. Mixing input calls is delicate work in any computer programming language. And the really nasty thing is that each programming language approaches the problem a little differently. What you find out about `nextLine` in Java helps you understand the issues when you get to know C++ or Visual Basic, but it doesn't tell you all the details. Each language's details are unique to that language. (Yes, it's a big pain. But because all computer programmers become rich and famous, the pain eventually pays off.)

Closing the connection to a disk file

To the average computer user, a keyboard doesn't feel anything like a file stored on a computer's hard drive. But disk files and keyboard input have a lot in common. In fact, a basic principle of computer operating systems dictates that any differences between two kinds of input be, for the programmer, as blurry as possible. As a Java programmer, you should treat disk files and keyboard input almost the same way. That's why Listing 8-2 contains a `diskScanner.close()` call.

When you run a Java program, you normally execute the `main` method's statements, starting with the first statement in the method body and ending with the last statement in the method body. You take detours along the way, skipping past `else` parts and diving into method bodies, but basically you finish executing

statements at the end of the `main` method. That's why, in Listing 8-2, the call to `close` is at the end of the `main` method's body. When you run the code in Listing 8-2, the last thing you do is disconnect from the disk file. And, fortunately, that disconnection takes place after you've executed all the `nextLine` and `nextDouble` calls.


TRY IT OUT

Previously in this chapter, you create instances of your own `PlaceToLive` class and display information about those instances. Modify the text-based version of your code so that it gets each instance's characteristics (address, number of bedrooms, and area) from a disk file.

Defining Subclasses (What It Means to Be a Full-Time or Part-Time Employee)

This time last year, your company paid $10 million for a piece of software. That software came in the `Employee.class` file. People at Burd Brain Consulting (the company that created the software) don't want you to know about the innards of the software. (Otherwise, you may steal their ideas.) So you don't have the Java program file that the software came from. (In other words, you don't have `Employee.java`.) You can run the bytecode in the `Employee.class` file. You can also read the documentation in a web page named *Employee.html*. But you can't see the statements inside the `Employee.java` program, and you can't change any of the program's code.

Since this time last year, your company has grown. Unlike in the old days, your company now has two kinds of employees: full-time and part-time. Each full-time employee is on a fixed, weekly salary. (If the employee works nights and weekends, then in return for this monumental effort, the employee receives a hearty handshake.) In contrast, each part-time employee works for an hourly wage. Your company deducts an amount from each full-time employee's paycheck to pay for the company's benefits package. Part-time employees, however, don't get benefits.

The question is whether the software that your company bought last year can keep up with the company's growth. You invested in a great program to handle employees and their payroll, but the program doesn't differentiate between your full-time and part-time employees. You have several options:

- **Call your next-door neighbor, whose 12-year-old child knows more about computer programming than anyone in your company.** Get this uppity

little brat to take the employee software apart, rewrite it, and hand it back to you with all the changes and additions your company requires.

On second thought, you can't do that. No matter how smart that kid is, the complexities of the employee software will probably confuse the kid. By the time you get the software back, it'll be filled with bugs and inconsistencies. Besides, you don't even have the `Employee.java` file to hand to the kid. All you have is the `Employee.class` file, which can't be read or modified with a text editor. (See Chapter 2.) Besides, your kid just beat up the neighbor's kid. You don't want to give your neighbor the satisfaction of seeing you beg for the whiz kid's help.

- **Scrap the $10 million employee software.** Get someone in your company to rewrite the software from scratch.

  In other words, say goodbye to your time and money.

- **Write a new front end for the employee software.** That is, build a piece of code that does some preliminary processing on full-time employees and then hands the preliminary results to your $10 million software. Do the same for part-time employees.

  This idea could be decent or spell disaster. Are you sure that the existing employee software has convenient *hooks* in it? (That is, does the employee software contain entry points that allow your front-end software to easily send preliminary data to the expensive employee software?) Remember: This plan treats the existing software as one big, monolithic lump, which can become cumbersome. Dividing the labor between your front-end code and the existing employee program is difficult. And if you add layer upon layer to existing black box code, you'll probably end up with a fairly inefficient system.

- **Call Burd Brain Consulting, the company that sold you the employee software.** Tell Dr. Burd that you want the next version of his software to differentiate between full-time and part-time employees.

  "No problem," says Dr. Burd. "It'll be ready by the start of the next fiscal quarter." That evening, Dr. Burd makes a discreet phone call to his next-door neighbor. . . .

- **Create two new Java classes named *FullTimeEmployee* and *PartTimeEmployee*.** Have each new class extend the existing functionality of the expensive `Employee` class, but have each new class define its own, specialized functionality for certain kinds of employees.

  Way to go! Figure 8-5 shows the structure that you want to create.

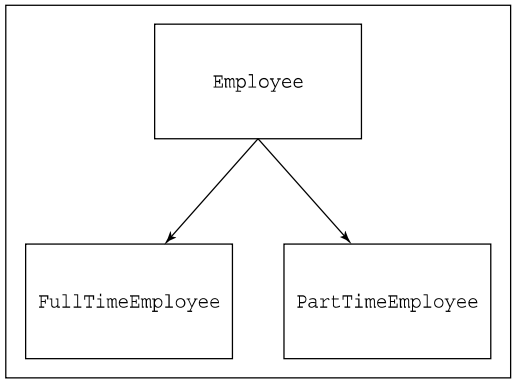

FIGURE 8-5: The Employee class family tree.

Creating a subclass

In Listing 8-1, I define an `Employee` class. I can use what I define in Listing 8-1 and extend the definition to create new, more specialized classes. So, in Listing 8-3, I define a new class: a `FullTimeEmployee` class.

LISTING 8-3: What Is a FullTimeEmployee?

```
public class FullTimeEmployee extends Employee {
    private double weeklySalary;
    private double benefitDeduction;

    public void setWeeklySalary(double weeklySalaryIn) {
        weeklySalary = weeklySalaryIn;
    }

    public double getWeeklySalary() {
        return weeklySalary;
    }

    public void setBenefitDeduction(double benefitDedIn) {
        benefitDeduction = benefitDedIn;
    }

    public double getBenefitDeduction() {
        return benefitDeduction;
    }

    public double findPaymentAmount() {
        return weeklySalary - benefitDeduction;
    }
}
```


Looking at Listing 8-3, you can see that each instance of the `FullTimeEmployee` class has two fields: `weeklySalary` and `benefitDeduction`. But are those the only fields that each `FullTimeEmployee` instance has? No, they're not. The first line of Listing 8-3 says that the `FullTimeEmployee` class extends the existing `Employee` class. This means that in addition to having a `weeklySalary` and a `benefitDeduction`, each `FullTimeEmployee` instance also has two other fields: `name` and `jobTitle`. These two fields come from the definition of the `Employee` class, which you can find in Listing 8-1.

In Listing 8-3, the magic word is *extends*. When one class extends an existing class, the extending class automatically inherits functionality that's defined in the existing class. So, the `FullTimeEmployee` class *inherits* the `name` and `jobTitle` fields. The `FullTimeEmployee` class also inherits all the methods that are declared in the `Employee` class: `setName`, `getName`, `setJobTitle`, `getJobTitle`, and `cutCheck`. The `FullTimeEmployee` class is a *subclass* of the `Employee` class. That means the `Employee` class is the *superclass* of the `FullTimeEmployee` class. You can also talk in terms of blood relatives: The `FullTimeEmployee` class is the *child* of the `Employee` class, and the `Employee` class is the *parent* of the `FullTimeEmployee` class.

It's almost (but not quite) as if the `FullTimeEmployee` class were defined by the code in Listing 8-4.

LISTING 8-4: Fake (But Informative) Code

```
import static java.lang.System.out;

public class FullTimeEmployee {
    private String name;
    private String jobTitle;
    private double weeklySalary;
    private double benefitDeduction;

    public void setName(String nameIn) {
        name = nameIn;
    }

    public String getName() {
        return name;
    }

    public void setJobTitle(String jobTitleIn) {
        jobTitle = jobTitleIn;
    }
```

(continued)