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

Java®

7th Edition

by Barry Burd, PhD

for dummies®
A Wiley Brand

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

Contents at a Glance

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.

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


TIP

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


WARNING

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


REMEMBER

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

Answer: A Remember icon.


CROSS REFERENCE

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


TRY IT OUT

Writing computer code is an activity, and the best way to learn an activity is to practice it. That's why I've created things for you to try in order to reinforce your knowledge. Many of these are confidence-builders, but some are a bit more challenging. When you first start putting things into practice, you'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!


TECHNICAL STUFF

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

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.

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`.

1 Getting Started with Java

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.

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.

> 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.

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?"

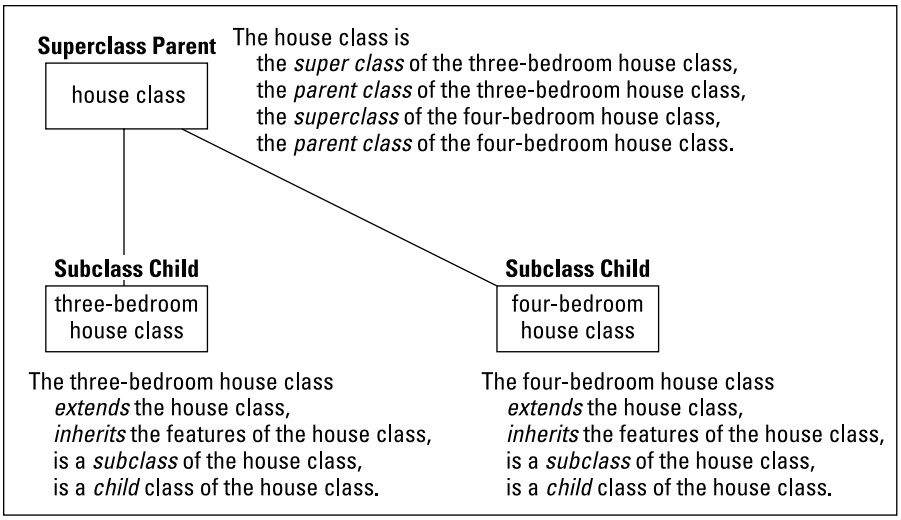

FIGURE 1-1: Terminology in object-oriented programming.

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.

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.


TECHNICAL STUFF

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.


TECHNICAL STUFF

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)

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

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 2-4 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.

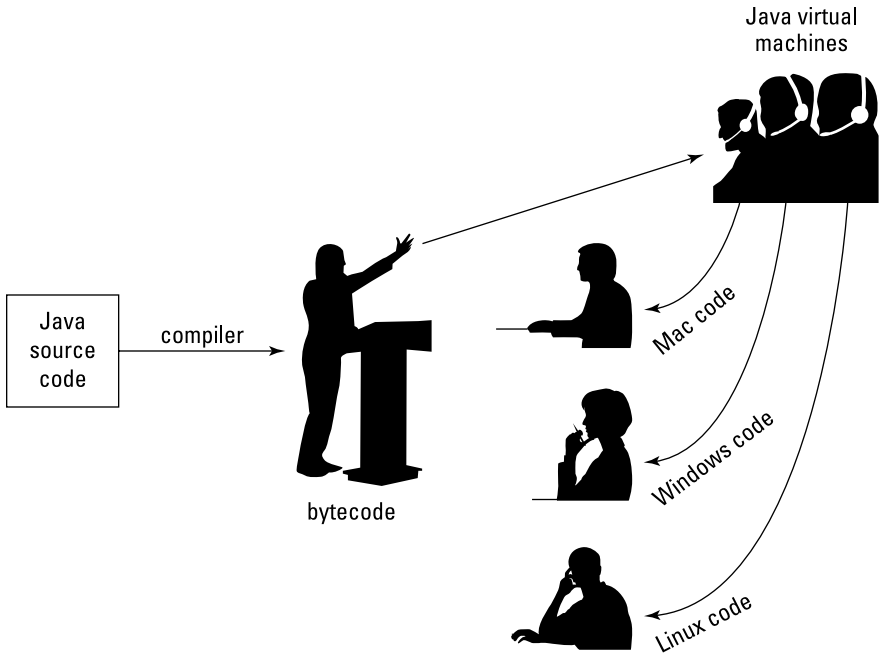

FIGURE 2-1: An imaginary meeting of the UN Security Council.

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)