Chapter 1

All about Java

In This Chapter

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

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, 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,* 6th 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,* 6th 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.)

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.

*I made up this fact all by myself.

- **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. Even today, COBOL represents a large part of the computer programming industry.

- **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 VB (Visual Basic) 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`).

- **August 2013: Java runs on more than 1.1 billion desktop computers** (`http://java.com/en/about`) **and Android Java runs on 250 million mobile phones** (`www.mobiledevicemanager.com/mobile-device-statistics/250-million-android-devices-in-use`).

  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 to the front of my house's driveway. As I pick up the morning paper, I glance at the front page and see the two-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).

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.*

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 one-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.

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:

- You could 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."
- You could take 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.

So there you have it. Object-oriented programming is misnamed. It should really 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 previous 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.)

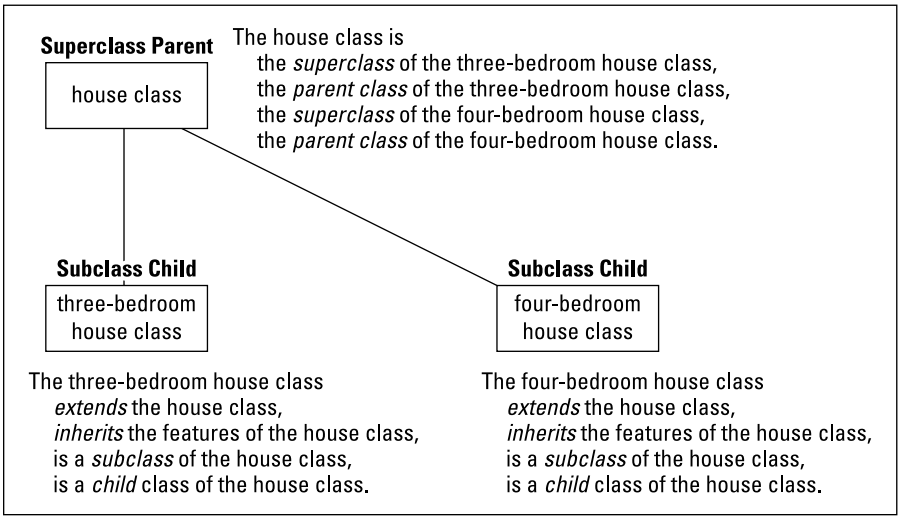

Figure 1-1: Terminology in object-oriented programming.

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. (See 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?"

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, 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.

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.