IN THIS CHAPTER

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

# Chapter 1
# The Big Picture



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

## What's It All About?

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

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

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

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

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

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

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

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

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

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

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

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

# Telling a computer what to do

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

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

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

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

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

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

## Pick your poison

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

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

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

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

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

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

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

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

# From Your Mind to the Computer's Processor

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

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

The next three sections describe each of the three tools.

## Translating your code

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

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

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


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

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

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

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

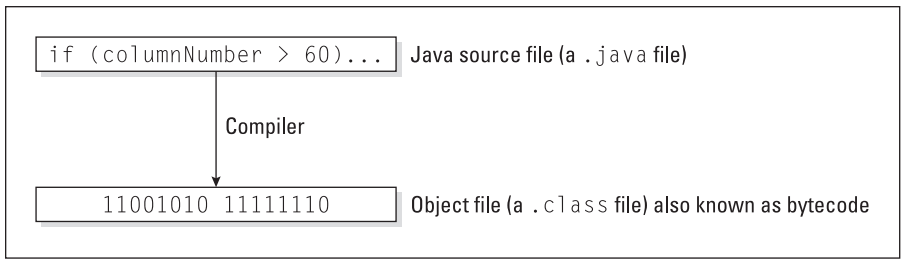
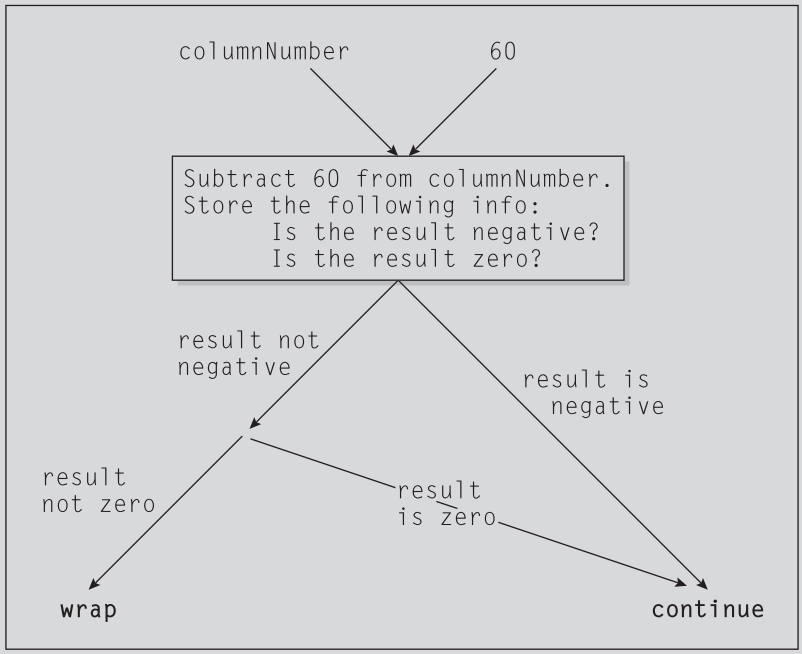



**FIGURE 1-2:** The computer compiles source code to create object code.

## WHAT IS BYTECODE, ANYWAY?

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

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




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


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

## Running code

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

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

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

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

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

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

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

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

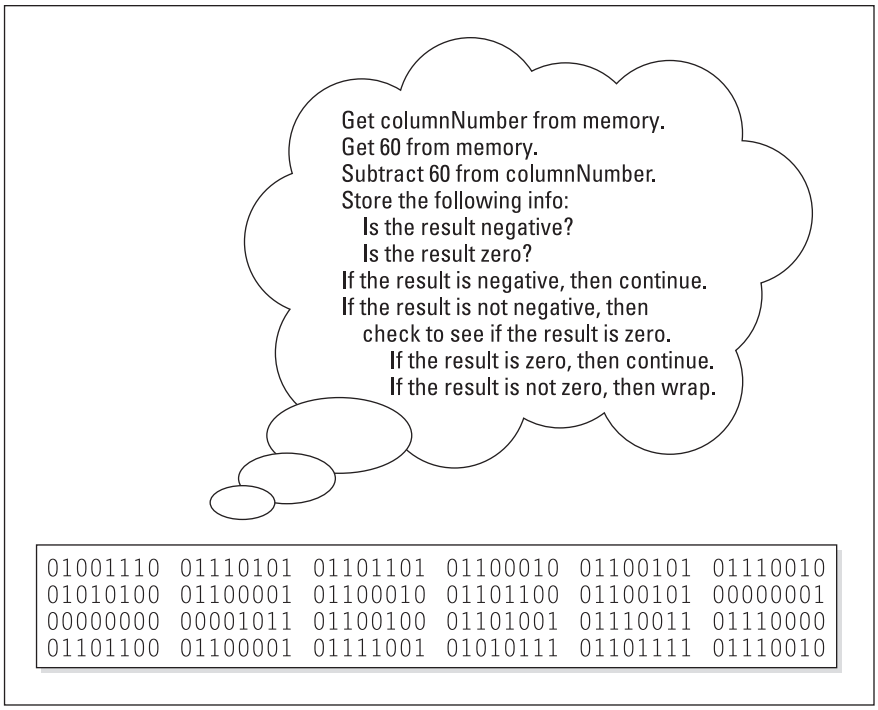



**FIGURE 1-3:** What the computer gleans from a bytecode file.

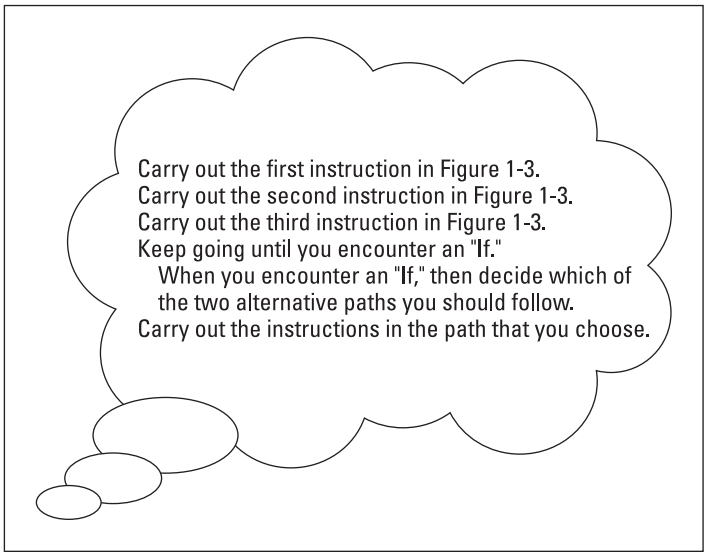



**FIGURE 1-4:** How a computer runs a Java program.

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

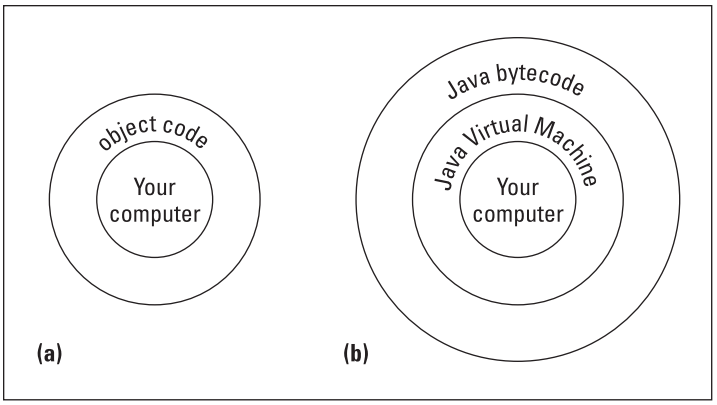



**FIGURE 1-5:** Two ways to run a computer program.

## WRITE ONCE, RUN ANYWHERE

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

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

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


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

## Code you can use

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

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

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

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

Every Java program, even the simplest one, calls on code in the Java API. This Java API is both useful and formidable. It's useful because of all the things you can do with the API's programs. It's formidable because the API is extensive. No one memorizes all the features made available by the Java API. Programmers remember the features that they use often, and they look up the features that they need

in a pinch. They look up these features in an online document called the *API Specification* (known affectionately to most Java programmers as the *API documentation*, or the *Javadocs*).

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

# Your Java Programming Toolset

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

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

You also need some less exotic tools:

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

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

  
  REMEMBER

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

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

## A tool for creating code

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

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

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

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

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

## What's already on your hard drive?

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

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