Chapter 1

Introduction to Java Programming

Introduction

This chapter begins with a review of some basic terminology about computers and computer programming. Many of these concepts will come up in later chapters, so it will be useful to review them before we start delving into the details of how to program in Java.

We will begin our exploration of Java by looking at simple programs that produce output. This discussion will allow us to explore many elements that are common to all Java programs, while working with programs that are fairly simple in structure.

After we have reviewed the basic elements of Java programs, we will explore the technique of procedural decomposition by learning how to break up a Java program into several methods. Using this technique, we can break up complex tasks into smaller subtasks that are easier to manage and we can avoid redundancy in our program solutions.

1.1 Basic Computing Concepts

Computers are pervasive in our daily lives, and, thanks to the Internet, they give us access to nearly limitless information. Some of this information is essential news, like the headlines at cnn.com. Computers let us share photos with our families and map directions to the nearest pizza place for dinner.

Lots of real-world problems are being solved by computers, some of which don't much resemble the one on your desk or lap. Computers allow us to sequence the human genome and search for DNA patterns within it. Computers in recently manufactured cars monitor each vehicle's status and motion. Digital music players such as Apple's iPod actually have computers inside their small casings. Even the Roomba vacuum-cleaning robot houses a computer with complex instructions about how to dodge furniture while cleaning your floors.

But what makes a computer a computer? Is a calculator a computer? Is a human being with a paper and pencil a computer? The next several sections attempt to address this question while introducing some basic terminology that will help prepare you to study programming.

Why Programming?

At most universities, the first course in computer science is a programming course. Many computer scientists are bothered by this because it leaves people with the impression that computer science is programming. While it is true that many trained computer scientists spend time programming, there is a lot more to the discipline. So why do we study programming first?

A Stanford computer scientist named Don Knuth answers this question by saying that the common thread for most computer scientists is that we all in some way work with *algorithms*.

Algorithm

A step-by-step description of how to accomplish a task.

Knuth is an expert in algorithms, so he is naturally biased toward thinking of them as the center of computer science. Still, he claims that what is most important is not the algorithms themselves, but rather the thought process that computer scientists employ to develop them. According to Knuth,

> It has often been said that a person does not really understand something until after teaching it to someone else. Actually a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm.[1]

[1]Knuth, Don. *Selected Papers on Computer Science*. Stanford, CA: Center for the Study of Language and Information, 1996.

Knuth is describing a thought process that is common to most of computer science, which he refers to as *algorithmic thinking*. We study programming not because it is the most important aspect of computer science, but because it is the best way to explain the approach that computer scientists take to solving problems.

The concept of algorithms is helpful in understanding what a computer is and what computer science is all about. The Merriam-Webster dictionary defines the word "computer" as "one that computes." Using that definition, all sorts of devices qualify as computers, including calculators, GPS navigation systems, and children's toys like the Furby. Prior to the invention of electronic computers, it was common to refer to humans as computers. The nineteenth-century mathematician Charles Peirce, for example, was originally hired to work for the U.S. government as an "Assistant Computer" because his job involved performing mathematical computations.

In a broad sense, then, the word "computer" can be applied to many devices. But when computer scientists refer to a computer, we are usually thinking of a universal computation device that can be programmed to execute any algorithm. Computer science, then, is the study of computational devices and the study of computation itself, including algorithms.

Algorithms are expressed as computer programs, and that is what this book is all about. But before we look at how to program, it will be useful to review some basic concepts about computers.

Hardware and Software

A computer is a machine that manipulates data and executes lists of instructions known as *programs*.

Program

A list of instructions to be carried out by a computer.

One key feature that differentiates a computer from a simpler machine like a calculator is its versatility. The same computer can perform many different tasks (playing games, computing income taxes, connecting to other computers around the world), depending on what program it is running at a given moment. A computer can run not only the programs that exist on it currently, but also new programs that haven't even been written yet.

The physical components that make up a computer are collectively called *hardware*. One of the most important pieces of hardware is the central processing unit, or *CPU*. The CPU is the "brain" of the computer: It is what executes the instructions. Also important is the computer's *memory* (often called random access memory, or *RAM*, because the computer can access any part of that memory at any time). The computer uses its memory to store programs that are being executed, along with their data. RAM is limited in size and does not retain its contents when the computer is turned off. Therefore, computers generally also use a *hard disk* as a larger permanent storage area.

Computer programs are collectively called *software*. The primary piece of software running on a computer is its operating system. An *operating system* provides an environment in which many programs may be run at the same time; it also provides a bridge between those programs, the hardware, and the *user* (the person using the computer). The programs that run inside the operating system are often called *applications*.

When the user selects a program for the operating system to run (e.g., by double-clicking the program's icon on the desktop), several things happen: The instructions for that program are loaded into the computer's memory from the hard disk, the operating system allocates memory for that program to use, and the instructions to run the program are fed from memory to the CPU and executed sequentially.

The Digital Realm

In the last section, we saw that a computer is a general-purpose device that can be programmed. You will often hear people refer to modern computers as *digital* computers because of the way they operate.

> **Digital**
>
> Based on numbers that increase in discrete increments, such as the integers 0, 1, 2, 3, etc.

Because computers are digital, everything that is stored on a computer is stored as a sequence of integers. This includes every program and every piece of data. An MP3 file, for example, is simply a long sequence of integers that stores audio information. Today we're used to digital music, digital pictures, and digital movies, but in the 1940s, when the first computers were built, the idea of storing complex data in integer form was fairly unusual.

Not only are computers digital, storing all information as integers, but they are also *binary*, which means they store integers as *binary numbers*.

> **Binary Number**
>
> A number composed of just 0s and 1s, also known as a base-2 number.

Humans generally work with *decimal* or base-10 numbers, which match our physiology (10 fingers and 10 toes). However, when we were designing the first computers, we wanted systems that would be easy to create and very reliable. It turned out to be simpler to build these systems on top of binary phenomena (e.g., a circuit being open or closed) rather than having 10 different states that would have to be distinguished from one another (e.g., 10 different voltage levels).

From a mathematical point of view, you can store things just as easily using binary numbers as you can using base-10 numbers. But since it is easier to construct a physical device that uses binary numbers, that's what computers use.

This does mean, however, that people who aren't used to computers find their conventions unfamiliar. As a result, it is worth spending a little time reviewing how binary

numbers work. To count with binary numbers, as with base-10 numbers, you start with 0 and count up, but you run out of digits much faster. So, counting in binary, you say

```
0
1
```

And already you've run out of digits. This is like reaching 9 when you count in base-10. After you run out of digits, you carry over to the next digit. So, the next two binary numbers are

```
10
11
```

And again, you've run out of digits. This is like reaching 99 in base-10. Again, you carry over to the next digit to form the three-digit number 100. In binary, whenever you see a series of ones, such as 111111, you know you're just one away from the digits all flipping to 0s with a 1 added in front, the same way that, in base-10, when you see a number like 999999, you know that you are one away from all those digits turning to 0s with a 1 added in front.

Table 1.1 shows how to count up to the base-10 number 8 using binary.

Table 1.1 Decimal vs. Binary

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

We can make several useful observations about binary numbers. Notice in the table that the binary numbers 1, 10, 100, and 1000 are all perfect powers of 2 (2^0, 2^1, 2^2, 2^3). In the same way that in base-10 we talk about a ones digit, tens digit, hundreds digit, and so on, we can think in binary of a ones digit, twos digit, fours digit, eights digit, sixteens digit, and so on.

Computer scientists quickly found themselves needing to refer to the sizes of different binary quantities, so they invented the term *bit* to refer to a single binary digit and the term *byte* to refer to 8 bits. To talk about large amounts of memory, they invented the terms "kilobytes" (KB), "megabytes" (MB), "gigabytes" (GB), and so on. Many people think that these correspond to the metric system, where "kilo" means 1000, but that is only approximately true. We use the fact that 2^{10} is approximately equal to 1000 (it actually equals 1024). Table 1.2 shows some common units of memory storage:

Table 1.2 Units of Memory Storage

Measurement	Power of 2	Actual Value	Example
kilobyte (KB)	2^{10}	1,024	500-word paper (3 KB)
megabyte (MB)	2^{20}	1,048,576	typical book (1 MB) or song (5 MB)
gigabyte (GB)	2^{30}	1,073,741,824	typical movie (4.7 GB)
terabyte (TB)	2^{40}	1,099,511,627,776	20 million books in the Library of Congress (20 TB)
petabyte (PB)	2^{50}	1,125,899,906,842,624	10 billion photos on Facebook (1.5 PB)

The Process of Programming

The word *code* describes program fragments ("these four lines of code") or the act of programming ("Let's code this into Java"). Once a program has been written, you can *execute* it.

> **Program Execution**
>
> The act of carrying out the instructions contained in a program.

The process of execution is often called *running*. This term can also be used as a verb ("When my program runs it does something strange") or as a noun ("The last run of my program produced these results").

A computer program is stored internally as a series of binary numbers known as the *machine language* of the computer. In the early days, programmers entered numbers like these directly into the computer. Obviously, this is a tedious and confusing way to program a computer, and we have invented all sorts of mechanisms to simplify this process.

Modern programmers write in what are known as high-level programming languages, such as Java. Such programs cannot be run directly on a computer: They first have to be translated into a different form by a special program known as a *compiler*.

> **Compiler**
>
> A program that translates a computer program written in one language into an equivalent program in another language (often, but not always, translating from a high-level language into machine language).

A compiler that translates directly into machine language creates a program that can be executed directly on the computer, known as an *executable*. We refer to such compilers as *native compilers* because they compile code to the lowest possible level (the native machine language of the computer).

This approach works well when you know exactly what computer you want to use to run your program. But what if you want to execute a program on many different

computers? You'd need a compiler that generates different machine language output for each of them. The designers of Java decided to use a different approach. They cared a lot about their programs being able to run on many different computers, because they wanted to create a language that worked well for the Web.

Instead of compiling into machine language, Java programs compile into what are known as *Java bytecodes*. One set of bytecodes can execute on many different machines. These bytecodes represent an intermediate level: They aren't quite as high-level as Java or as low-level as machine language. In fact, they are the machine language of a theoretical computer known as the *Java Virtual Machine (JVM)*.

Java Virtual Machine (JVM)

A theoretical computer whose machine language is the set of Java bytecodes.

A JVM isn't an actual machine, but it's similar to one. When we compile programs to this level, there isn't much work remaining to turn the Java bytecodes into actual machine instructions.

To actually execute a Java program, you need another program that will execute the Java bytecodes. Such programs are known generically as *Java runtimes,* and the standard environment distributed by Oracle Corporation is known as the *Java Runtime Environment (JRE).*

Java Runtime

A program that executes compiled Java bytecodes.

Most people have Java runtimes on their computers, even if they don't know about them. For example, Apple's Mac OS X includes a Java runtime, and many Windows applications install a Java runtime.

Why Java?

When Sun Microsystems released Java in 1995, it published a document called a "white paper" describing its new programming language. Perhaps the key sentence from that paper is the following:

> Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.[2]

This sentence covers many of the reasons why Java is a good introductory programming language. For starters, Java is reasonably simple for beginners to learn, and it embraces object-oriented programming, a style of writing programs that has been shown to be very successful for creating large and complex software systems.

[2] http://www.oracle.com/technetwork/java/langenv-140151.html

Java also includes a large amount of prewritten software that programmers can utilize to enhance their programs. Such off-the-shelf software components are often called *libraries*. For example, if you wish to write a program that connects to a site on the Internet, Java contains a library to simplify the connection for you. Java contains libraries to draw graphical user interfaces (GUIs), retrieve data from databases, and perform complex mathematical computations, among many other things. These libraries collectively are called the *Java class libraries*.

Java Class Libraries

The collection of preexisting Java code that provides solutions to common programming problems.

The richness of the Java class libraries has been an extremely important factor in the rise of Java as a popular language. The Java class libraries in version 1.7 include over 4000 entries.

Another reason to use Java is that it has a vibrant programmer community. Extensive online documentation and tutorials are available to help programmers learn new skills. Many of these documents are written by Oracle, including an extensive reference to the Java class libraries called the *API Specification* (API stands for Application Programming Interface).

Java is extremely platform independent; unlike programs written in many other languages, the same Java program can be executed on many different operating systems, such as Windows, Linux, and Mac OS X.

Java is used extensively for both research and business applications, which means that a large number of programming jobs exist in the marketplace today for skilled Java programmers. A sample Google search for the phrase "Java jobs" returned around 180,000,000 hits at the time of this writing.

The Java Programming Environment

You must become familiar with your computer setup before you start programming. Each computer provides a different environment for program development, but there are some common elements that deserve comment. No matter what environment you use, you will follow the same basic three steps:

1. Type in a program as a Java class.
2. Compile the program file.
3. Run the compiled version of the program.

The basic unit of storage on most computers is a *file*. Every file has a name. A file name ends with an *extension*, which is the part of a file's name that follows the period. A file's extension indicates the type of data contained in the file. For example, files with the extension `.doc` are Microsoft Word documents, and files with the extension `.mp3` are MP3 audio files.

The Java program files that you create must use the extension `.java`. When you compile a Java program, the resulting Java bytecodes are stored in a file with the same name and the extension `.class`.

Most Java programmers use what are known as Integrated Development Environments, or IDEs, which provide an all-in-one environment for creating, editing, compiling, and executing program files. Some of the more popular choices for introductory computer science classes are Eclipse, jGRASP, DrJava, BlueJ, and TextPad. Your instructor will tell you what environment you should use.

Try typing the following simple program in your IDE (the line numbers are not part of the program but are used as an aid):

```
1  public class Hello {
2      public static void main(String[] args) {
3          System.out.println("Hello, world!");
4      }
5  }
```

Don't worry about the details of this program right now. We will explore those in the next section.

Once you have created your program file, move to step 2 and compile it. The command to compile will be different in each development environment, but the process is the same (typical commands are "compile" or "build"). If any errors are reported, go back to the editor, fix them, and try to compile the program again. (We'll discuss errors in more detail later in this chapter.)

Once you have successfully compiled your program, you are ready to move to step 3, running the program. Again, the command to do this will differ from one environment to the next, but the process is similar (the typical command is "run"). The diagram in Figure 1.1 summarizes the steps you would follow in creating a program called `Hello.java`.

In some IDEs (most notably Eclipse), the first two steps are combined. In these environments the process of compiling is more incremental; the compiler will warn you about errors as you type in code. It is generally not necessary to formally ask such an environment to compile your program because it is compiling as you type.

When your program is executed, it will typically interact with the user in some way. The `Hello.java` program involves an onscreen window known as the *console*.

Console Window

A special text-only window in which Java programs interact with the user.

The console window is a classic interaction mechanism wherein the computer displays text on the screen and sometimes waits for the user to type responses. This is known as *console* or *terminal interaction*. The text the computer prints to the console window is known as the *output* of the program. Anything typed by the user is known as the console *input*.

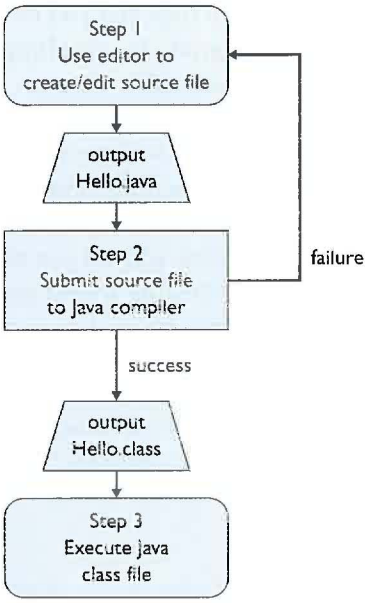


Figure 1.1 Creation and execution of a Java program

To keep things simple, most of the sample programs in this book involve console interaction. Keeping the interaction simple will allow you to focus your attention and effort on other aspects of programming.

1.2 And Now—Java

It's time to look at a complete Java program. In the Java programming language, nothing can exist outside of a *class*.

Class

A unit of code that is the basic building block of Java programs.

The notion of a class is much richer than this, as you'll see when we get to Chapter 8, but for now all you need to know is that each of your Java programs will be stored in a class.

It is a tradition in computer science that when you describe a new programming language, you should start with a program that produces a single line of output with the words, "Hello, world!" The "hello world" tradition has been broken by many authors of Java books because the program turns out not to be as short and simple when it is written in Java as when it is written in other languages, but we'll use it here anyway.

Here is our "hello world" program:

```
1  public class Hello {
2      public static void main(String[] args) {
3          System.out.println("Hello, world!");
4      }
5  }
```

This program defines a class called `Hello`. Oracle has established the convention that class names always begin with a capital letter, which makes it easy to recognize them. Java requires that the class name and the file name match, so this program must be stored in a file called `Hello.java`. You don't have to understand all the details of this program just yet, but you do need to understand the basic structure.

The basic form of a Java class is as follows:

```
public class <name> {
    <method>
    <method>
    ...
    <method>
}
```

This type of description is known as a *syntax template* because it describes the basic form of a Java construct. Java has rules that determine its legal *syntax* or grammar. Each time we introduce a new element of Java, we'll begin by looking at its syntax template. By convention, we use the less-than (<) and greater-than (>) characters in a syntax template to indicate items that need to be filled in (in this case, the name of the class and the methods). When we write "..." in a list of elements, we're indicating that any number of those elements may be included.

The first line of the class is known as the *class header.* The word `public` in the header indicates that this class is available to anyone to use. Notice that the program code in a class is enclosed in curly brace characters (`{ }`). These characters are used in Java to group together related bits of code. In this case, the curly braces are indicating that everything defined within them is part of this public class.

So what exactly can appear inside the curly braces? What can be contained in a class? All sorts of things, but for now, we'll limit ourselves to *methods.* Methods are the next-smallest unit of code in Java, after classes. A method represents a single action or calculation to be performed.

Method

A program unit that represents a particular action or computation.

Simple methods are like verbs: They command the computer to perform some action. Inside the curly braces for a class, you can define several different methods.

At a minimum, a complete program requires a special method that is known as the `main` method. It has the following syntax:

```
public static void main(String[] args) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

Just as the first line of a class is known as a class header, the first line of a method is known as a *method header.* The header for `main` is rather complicated. Most people memorize this as a kind of magical incantation. You want to open the door to Ali Baba's cave? You say, "Open Sesame." You want to create an executable Java program? You say, `public static void main(String[] args)`. A group of Java teachers make fun of this with a website called publicstaticvoidmain.com.

Just memorizing magical incantations is never satisfying, especially for computer scientists who like to know everything that is going on in their programs. But this is a place where Java shows its ugly side, and you'll just have to live with it. New programmers, like new drivers, must learn to use something complex without fully understanding how it works. Fortunately, by the time you finish this book, you'll understand every part of the incantation.

Notice that the `main` method has a set of curly braces of its own. They are again used for grouping, indicating that everything that appears between them is part of the `main` method. The lines in between the curly braces specify the series of actions the computer should perform when it executes the method. We refer to these as the *statements* of the method. Just as you put together an essay by stringing together complete sentences, you put together a method by stringing together statements.

Statement

An executable snippet of code that represents a complete command.

Each statement is terminated by a semicolon. The sample "hello world" program has just a single statement that is known as a `println` statement:

```
System.out.println("Hello, world!");
```

Notice that this statement ends with a semicolon. The semicolon has a special status in Java; it is used to terminate statements in the same way that periods terminate sentences in English.

In the basic "hello world" program there is just a single command to produce a line of output, but consider the following variation (called `Hello2`), which has four lines of code to be executed in the `main` method:

```
1 public class Hello2 {
2     public static void main(String[] args) {
3         System.out.println("Hello, world!");
4         System.out.println();
5         System.out.println("This program produces four");
6         System.out.println("lines of output.");
7     }
8 }
```

Notice that there are four semicolons in the `main` method, one at the end of each of the four `println` statements. The statements are executed in the order in which they appear, from first to last, so the `Hello2` program produces the following output:

```
Hello, world!

This program produces four
lines of output.
```

Let's summarize the different levels we just looked at:

- A Java program is stored in a class.
- Within the class, there are methods. At a minimum, a complete program requires a special method called `main`.
- Inside a method like `main`, there is a series of statements, each of which represents a single command for the computer to execute.

It may seem odd to put the opening curly brace at the end of a line rather than on a line by itself. Some people would use this style of indentation for the program instead:

```
1 public class Hello3
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Hello, world!");
6     }
7 }
```

Different people will make different choices about the placement of curly braces. The style we use follows Oracle's official Java coding conventions, but the other style has its advocates too. Often people will passionately argue that one way is much better than the other, but it's really a matter of personal taste because each choice has some advantages and some disadvantages. Your instructor may require a particular style; if not, you should choose a style that you are comfortable with and then use it consistently.

Now that you've seen an overview of the structure, let's examine some of the details of Java programs.

Did You Know?

Hello, World!

The "hello world" tradition was started by Brian Kernighan and Dennis Ritchie. Ritchie invented a programming language known as C in the 1970s and, together with Kernighan, coauthored the first book describing C, published in 1978. The first complete program in their book was a "hello world" program. Kernighan and Ritchie, as well as their book *The C Programming Language*, have been affectionately referred to as "K & R" ever since.

Many major programming languages have borrowed the basic C syntax as a way to leverage the popularity of C and to encourage programmers to switch to it. The languages C++ and Java both borrow a great deal of their core syntax from C.

Kernighan and Ritchie also had a distinctive style for the placement of curly braces and the indentation of programs that has become known as "K & R style." This is the style that Oracle recommends and that we use in this book.

String Literals (Strings)

When you are writing Java programs (such as the preceding "hello world" program), you'll often want to include some literal text to send to the console window as output. Programmers have traditionally referred to such text as a *string* because it is composed of a sequence of characters that we string together. The Java language specification uses the term *string literals*.

In Java you specify a string literal by surrounding the literal text in quotation marks, as in

```
"This is a bunch of text surrounded by quotation marks."
```

You must use double quotation marks, not single quotation marks. The following is not a valid string literal:

```
'Bad stuff here.'
```

The following is a valid string literal:

```
"This is a string even with 'these' quotes inside."
```

String literals must not span more than one line of a program. The following is not a valid string literal:

```
"This is really
bad stuff
right here."
```

System.out.println

As you have seen, the `main` method of a Java program contains a series of statements for the computer to carry out. They are executed sequentially, starting with the first statement, then the second, then the third, and so on until the final statement has been executed. One of the simplest and most common statements is `System.out.println`, which is used to produce a line of output. This is another "magical incantation" that you should commit to memory. As of this writing, Google lists around 8,000,000 web pages that mention `System.out.println`. The key thing to remember about this statement is that it's used to produce a line of output that is sent to the console window.

The simplest form of the `println` statement has nothing inside its parentheses and produces a blank line of output:

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

You need to include the parentheses even if you don't have anything to put inside them. Notice the semicolon at the end of the line. All statements in Java must be terminated with a semicolon.

More often, however, you use `println` to output a line of text:

```
System.out.println("This line uses the println method.");
```

The above statement commands the computer to produce the following line of output:

```
This line uses the println method.
```

Each `println` statement produces a different line of output. For example, consider the following three statements:

```
System.out.println("This is the first line of output.");
System.out.println();
System.out.println("This is the third, below a blank line.");
```

Executing these statements produces the following three lines of output (the second line is blank):

```
This is the first line of output.

This is the third, below a blank line.
```

Escape Sequences

Any system that involves quoting text will lead you to certain difficult situations. For example, string literals are contained inside quotation marks, so how can you include a quotation mark inside a string literal? String literals also aren't allowed to break across lines, so how can you include a line break inside a string literal?

The solution is to embed what are known as *escape sequences* in the string literals. Escape sequences are two-character sequences that are used to represent special characters. They all begin with the backslash character (\). Table 1.3 lists some of the more common escape sequences.

Table 1.3 Common Escape Sequences

Sequence	Represents
\t	tab character
\n	new line character
\"	quotation mark
\\	backslash character

Keep in mind that each of these two-character sequences actually stands for just a single character. For example, consider the following statement:

```
System.out.println("What \"characters\" does this \\ print?");
```

If you executed this statement, you would get the following output:

```
What "characters" does this \ print?
```

The string literal in the `println` has three escape sequences, each of which is two characters long and produces a single character of output.

While string literals themselves cannot span multiple lines (that is, you cannot use a carriage return within a string literal to force a line break), you can use the `\n` escape sequence to embed new line characters in a string. This leads to the odd situation where a single `println` statement can produce more than one line of output.

For example, consider this statement:

```
System.out.println("This\nproduces 3 lines\nof output.");
```

If you execute it, you will get the following output:

```
This
produces 3 lines
of output.
```

The `println` itself produces one line of output, but the string literal contains two new line characters that cause it to be broken up into a total of three lines of output. To produce the same output without new line characters, you would have to issue three separate `println` statements.

This is another programming habit that tends to vary according to taste. Some people (including the authors) find it hard to read string literals that contain `\n` escape sequences, but other people prefer to write fewer lines of code. Once again, you should make up your own mind about when to use the new line escape sequence.

print versus println

Java has a variation of the `println` command called `print` that allows you to produce output on the current line without going to a new line of output. The `println` command really does two different things: It sends output to the current line, and then it moves to the beginning of a new line. The `print` command does only the first of these. Thus, a series of `print` commands will generate output all on the same line. Only a `println` command will cause the current line to be completed and a new line to be started. For example, consider these six statements:

```
System.out.print("To be ");
System.out.print("or not to be.");
System.out.print("That is ");
System.out.println("the question.");
System.out.print("This is");
System.out.println(" for the whole family!");
```

These statements produce two lines of output. Remember that every `println` statement produces exactly one line of output; because there are two `println` statements here, there are two lines of output. After the first statement executes, the current line looks like this:

```
To be
      ^
```

The arrow below the output line indicates the position where output will be sent next. We can simplify our discussion if we refer to the arrow as the *output cursor*. Notice that the output cursor is at the end of this line and that it appears after a space. The reason is that the command was a `print` (doesn't go to a new line) and the string literal in the `print` ended with a space. Java will not insert a space for you unless you specifically request it. After the next `print`, the line looks like this:

```
To be or not to be.
                   ^
```

There's no space at the end now because the string literal in the second `print` command ends in a period, not a space. After the next `print`, the line looks like this:

```
To be or not to be.That is
                          ^
```

There is no space between the period and the word "That" because there was no space in the `print` commands, but there is a space at the end of the string literal in the third statement. After the next statement executes, the output looks like this:

```
To be or not to be.That is the question.

^
```

Because this fourth statement is a `println` command, it finishes the output line and positions the cursor at the beginning of the second line. The next statement is another `print` that produces this:

```
To be or not to be.That is the question.
This is
      ^
```

The final `println` completes the second line and positions the output cursor at the beginning of a new line:

```
To be or not to be.That is the question.
This is for the whole family!

^
```

These six statements are equivalent to the following two single statements:

```
System.out.println("To be or not to be.That is the question.");
System.out.println("This is for the whole family!");
```

Using the `print` and `println` commands together to produce lines like these may seem a bit silly, but you will see that there are more interesting applications of `print` in the next chapter.

Remember that it is possible to have an empty `println` command:

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

Because there is nothing inside the parentheses to be written to the output line, this command positions the output cursor at the beginning of the next line. If there are `print` commands before this empty `println`, it finishes out the line made by those `print` commands. If there are no previous `print` commands, it produces a blank line. An empty `print` command is meaningless and illegal.

Identifiers and Keywords

The words used to name parts of a Java program are called *identifiers*.

Identifier

A name given to an entity in a program, such as a class or method.

Identifiers must start with a letter, which can be followed by any number of letters or digits. The following are all legal identifiers:

```
first          hiThere        numStudents    TwoBy4
```

The Java language specification defines the set of letters to include the underscore and dollar-sign characters (_ and $), which means that the following are legal identifiers as well:

```
two_plus_two      _count      $2donuts      MAX_COUNT
```

The following are illegal identifiers:

```
two+two           hi there      hi-There      2by4
```

Java has conventions for capitalization that are followed fairly consistently by programmers. All class names should begin with a capital letter, as with the `Hello`, `Hello2`, and `Hello3` classes introduced earlier. The names of methods should begin with lowercase letters, as in the `main` method. When you are putting several words together to form a class or method name, capitalize the first letter of each word after the first. In the next chapter we'll discuss constants, which have yet another capitalization scheme, with all letters in uppercase and words separated by underscores. These different schemes might seem like tedious constraints, but using consistent capitalization in your code allows the reader to quickly identify the various code elements.

For example, suppose that you were going to put together the words "all my children" into an identifier. The result would be

- `AllMyChildren` for a class name (each word starts with a capital)
- `allMyChildren` for a method name (starts with a lowercase letter, subsequent words capitalized)
- `ALL_MY_CHILDREN` for a constant name (all uppercase, with words separated by underscores; described in Chapter 2)

Java is case sensitive, so the identifiers `class`, `Class`, `CLASS`, and `cLASs` are all considered different. Keep this in mind as you read error messages from the compiler. People are good at understanding what you write, even if you misspell words or make little mistakes like changing the capitalization of a word. However, mistakes like these cause the Java compiler to become hopelessly confused.

Don't hesitate to use long identifiers. The more descriptive your names are, the easier it will be for people (including you) to read your programs. Descriptive identifiers are worth the time they take to type. Java's `String` class, for example, has a method called `compareToIgnoreCase`.

Be aware, however, that Java has a set of predefined identifiers called *keywords* that are reserved for particular uses. As you read this book, you will learn many of these keywords and their uses. You can only use keywords for their intended purposes. You must be careful to avoid using these words in the names of identifiers. For example, if you name a method `short` or `try`, this will cause a problem, because `short` and `try` are reserved keywords. Table 1.4 shows the complete list of reserved keywords.

Table 1.4 List of Java Keywords

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

A Complex Example: `DrawFigures1`

The `println` statement can be used to draw text figures as output. Consider the following more complicated program example (notice that it uses two empty `println` statements to produce blank lines):

```
1  public class DrawFigures1 {
2      public static void main(String[] args) {
3          System.out.println("   /\\");
4          System.out.println("  /  \\");
5          System.out.println(" /    \\");
6          System.out.println(" \\    /");
7          System.out.println("  \\  /");
8          System.out.println("   \\/");
9          System.out.println();
10         System.out.println(" \\    /");
11         System.out.println("  \\  /");
12         System.out.println("   \\/");
13         System.out.println("   /\\");
14         System.out.println("  /  \\");
15         System.out.println(" /    \\");
16         System.out.println();
17         System.out.println("   /\\");
18         System.out.println("  /  \\");
19         System.out.println(" /    \\");
20         System.out.println("+------+");
21         System.out.println("|      |");
22         System.out.println("|      |");
23         System.out.println("+------+");
24         System.out.println("|United|");
25         System.out.println("|States|");
26         System.out.println("+------+");
27         System.out.println("|      |");
```

```
28          System.out.println("|      |");
29          System.out.println("+------+");
30          System.out.println("   /\\");
31          System.out.println("  /  \\");
32          System.out.println(" /    \\");
33      }
34  }
```

The following is the output the program generates. Notice that the program includes double backslash characters (\\), but the output has single backslash characters. This is an example of an escape sequence, as described previously.

```
   /\
  /  \
 /    \
 \    /
  \  /
   \/

 \    /
  \  /
   \/
   /\
  /  \
 /    \

   /\
  /  \
 /    \
+------+
|      |
|      |
+------+
|United|
|States|
+------+
|      |
|      |
+------+
   /\
  /  \
 /    \
```

Comments and Readability

Java is a free-format language. This means you can put in as many or as few spaces and blank lines as you like, as long as you put at least one space or other punctuation mark between words. However, you should bear in mind that the layout of a program can enhance (or detract from) its readability. The following program is legal but hard to read:

```
1  public class Ugly{public static void main(String[] args)
2  {System.out.println("How short I am!");}}
```

Here are some simple rules to follow that will make your programs more readable:

- Put class and method headers on lines by themselves.
- Put no more than one statement on each line.
- Indent your program properly. When an opening brace appears, increase the indentation of the lines that follow it. When a closing brace appears, reduce the indentation. Indent statements inside curly braces by a consistent number of spaces (a common choice is four spaces per level of indentation).
- Use blank lines to separate parts of the program (e.g., methods).

Using these rules to rewrite the `Ugly` program yields the following code:

```
1  public class Ugly {
2      public static void main(String[] args) {
3          System.out.println("How short I am!");
4      }
5  }
```

Well-written Java programs can be quite readable, but often you will want to include some explanations that are not part of the program itself. You can annotate programs by putting notes called *comments* in them.

Comment

Text that programmers include in a program to explain their code. The compiler ignores comments.

There are two comment forms in Java. In the first form, you open the comment with a slash followed by an asterisk and you close it with an asterisk followed by a slash:

```
/* like this */
```

You must not put spaces between the slashes and the asterisks:

```
/ * this is bad * /
```

You can put almost any text you like, including multiple lines, inside the comment:

```
/* Thaddeus Martin
   Assignment #1
   Instructor: Professor Walingford
   Grader:     Bianca Montgomery      */
```

The only things you aren't allowed to put inside a comment are the comment end characters. The following code is not legal:

```
/* This comment has an asterisk/slash /*/ in it,
    which prematurely closes the comment. This is bad. */
```

Java also provides a second comment form for shorter, single-line comments. You can use two slashes in a row to indicate that the rest of the current line (everything to the right of the two slashes) is a comment. For example, you can put a comment after a statement:

```
System.out.println("You win!"); // Good job!
```

Or you can create a comment on its own line:

```
// give an introduction to the user
System.out.println("Welcome to the game of blackjack.");
System.out.println();
System.out.println("Let me explain the rules.");
```

You can even create blocks of single-line comments:

```
// Thaddeus Martin
// Assignment #1
// Instructor:  Professor Walingford
// Grader:      Bianca Montgomery
```

Some people prefer to use the first comment form for comments that span multiple lines but it is safer to use the second form because you don't have to remember to close the comment. It also makes the comment stand out more. This is another case in which, if your instructor does not tell you to use a particular comment style, you should decide for yourself which style you prefer and use it consistently.

Don't confuse comments with the text of `println` statements. The text of your comments will not be displayed as output when the program executes. The comments are there only to help readers examine and understand the program.

It is a good idea to include comments at the beginning of each class file to indicate what the class does. You might also want to include information about who you are, what course you are taking, your instructor and/or grader's name, the date, and so on. You should also comment each method to indicate what it does.

Commenting becomes more useful in larger and more complicated programs, as well as in programs that will be viewed or modified by more than one programmer. Clear comments are extremely helpful to explain to another person, or to yourself at a later time, what your program is doing and why it is doing it.

In addition to the two comment forms already discussed, Java supports a particular style of comments known as *Javadoc comments*. Their format is more complex, but they have the advantage that you can use a program to extract the comments to make HTML files suitable for reading with a web browser. Javadoc comments are useful in more advanced programming and are discussed in more detail in Appendix B.

1.3 Program Errors

In 1949, Maurice Wilkes, an early pioneer of computing, expressed a sentiment that still rings true today:

> As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

You also will have to face this reality as you learn to program. You're going to make mistakes, just like every other programmer in history, and you're going to need strategies for eliminating those mistakes. Fortunately, the computer itself can help you with some of the work.

There are three kinds of errors that you'll encounter as you write programs:

- *Syntax errors* occur when you misuse Java. They are the programming equivalent of bad grammar and are caught by the Java compiler.
- *Logic errors* occur when you write code that doesn't perform the task it is intended to perform.
- *Runtime errors* are logic errors that are so severe that Java stops your program from executing.

Syntax Errors

Human beings tend to be fairly forgiving about minor mistakes in speech. For example, we might find it to be odd phrasing, but we generally understand Master Yoda when he says, "Unfortunate that you rushed to face him . . . that incomplete was your training. Not ready for the burden were you."

The Java compiler will be far less forgiving. The compiler reports syntax errors as it attempts to translate your program from Java into bytecodes if your program breaks any of Java's grammar rules. For example, if you misplace a single semicolon in your program, you can send the compiler into a tailspin of confusion. The compiler may report several error messages, depending on what it thinks is wrong with your program.

A program that generates compilation errors cannot be executed. If you submit your program to the compiler and the compiler reports errors, you must fix the errors and resubmit the program. You will not be able to proceed until your program is free of compilation errors.

Some development environments, such as Eclipse, help you along the way by underlining syntax errors as you write your program. This makes it easy to spot exactly where errors occur.

It's possible for you to introduce an error before you even start writing your program, if you choose the wrong name for its file.

Common Programming Error

File Name Does Not Match Class Name

As mentioned earlier, Java requires that a program's class name and file name match. For example, a program that begins with `public class Hello` must be stored in a file called `Hello.java`.

If you use the wrong file name (for example, saving it as `WrongFileName.java`), you'll get an error message like this:

```
WrongFileName.java:1: error: class Hello is public,
    should be declared in a file named Hello.java
public class Hello {
       ^
1 error
```

The file name is just the first hurdle. A number of other errors may exist in your Java program. One of the most common syntax errors is to misspell a word. You may have punctuation errors, such as missing semicolons. It's also easy to forget an entire word, such as a required keyword.

The error messages the compiler gives may or may not be helpful. If you don't understand the content of the error message, look for the caret marker (^) below the line, which points at the position in the line where the compiler became confused. This can help you pinpoint the place where a required keyword might be missing.

Common Programming Error

Misspelled Words

Java (like most programming languages) is very picky about spelling. You need to spell each word correctly, including proper capitalization. Suppose, for example, that you were to replace the `println` statement in the "hello world" program with the following:

```
System.out.pruntln("Hello, world!");
```

When you try to compile this program, it will generate an error message similar to the following:

```
Hello.java:3: error: cannot find symbol
symbol  : method pruntln(java.lang.String)
```

Continued on next page

Continued from previous page

```
location: variable out of type PrintStream
        System.out.pruntln("Hello, world!");
                   ^
1 error
```

The first line of this output indicates that the error occurs in the file `Hello.java` on line 3 and that the error is that the compiler cannot find a symbol. The second line indicates that the symbol it can't find is a method called `pruntln`. That's because there is no such method; the method is called `println`. The error message can take slightly different forms depending on what you have misspelled. For example, you might forget to capitalize the word `System`:

```
system.out.println("Hello, world!");
```

You will get the following error message:

```
Hello.java:3: error: package system does not exist
        system.out.println("Hello, world!");
               ^
1 error
```

Again, the first line indicates that the error occurs in line 3 of the file `Hello.java`. The error message is slightly different here, though, indicating that it can't find a package called `system`. The second and third lines of this error message include the original line of code with an arrow (caret) pointing to where the compiler got confused. The compiler errors are not always very clear, but if you pay attention to where the arrow is pointing, you'll have a pretty good sense of where the error occurs.

If you still can't figure out the error, try looking at the error's line number and comparing the contents of that line with similar lines in other programs. You can also ask someone else, such as an instructor or lab assistant, to examine your program.

Common Programming Error

Forgetting a Semicolon

All Java statements must end with semicolons, but it's easy to forget to put a semicolon at the end of a statement, as in the following program:

```
1  public class MissingSemicolon {
2      public static void main(String[] args) {
3          System.out.println("A rose by any other name")
```

Continued on next page

Continued from previous page

```
4           System.out.println("would smell as sweet");
5       }
6   }
```

In this case, the compiler produces output similar to the following:

```
MissingSemicolon.java:3: error: ';' expected
        System.out.println("would smell as sweet");
                          ^
1 error
```

Some versions of the Java compiler list line 4 as the cause of the problem, not line 3, where the semicolon was actually forgotten. This is because the compiler is looking forward for a semicolon and isn't upset until it finds something that isn't a semicolon, which it does when it reaches line 4. Unfortunately, as this case demonstrates, compiler error messages don't always direct you to the correct line to be fixed.

Common Programming Error

Forgetting a Required Keyword

Another common syntax error is to forget a required keyword when you are typing your program, such as `static` or `class`. Double-check your programs against the examples in the textbook to make sure you haven't omitted an important keyword.

The compiler will give different error messages depending on which keyword is missing, but the messages can be hard to understand. For example, you might write a program called `Bug4` and forget the keyword `class` when writing its class header. In this case, the compiler will provide the following error message:

```
Bug4.java:1: error: class, interface, or enum expected
public Bug4 {
       ^
1 error
```

However, if you forget the keyword `void` when declaring the main method, the compiler generates a different error message:

```
Bug5.java:2: error: invalid method declaration; return type required
    public static main(String[] args) {
                  ^
1 error
```

Yet another common syntax error is to forget to close a string literal.

A good rule of thumb to follow is that the first error reported by the compiler is the most important one. The rest might be the result of that first error. Many programmers don't even bother to look at errors beyond the first, because fixing that error and recompiling may cause the other errors to disappear.

Logic Errors (Bugs)

Logic errors are also called *bugs*. Computer programmers use words like "bug-ridden" and "buggy" to describe poorly written programs, and the process of finding and eliminating bugs from programs is called *debugging*.

The word "bug" is an old engineering term that predates computers; early computing bugs sometimes occurred in hardware as well as software. Admiral Grace Hopper, an early pioneer of computing, is largely credited with popularizing the use of the term in the context of computer programming. She often told the true story of a group of programmers at Harvard University in the mid-1940s who couldn't figure out what was wrong with their programs until they opened up the computer and found an actual moth trapped inside.

The form that a bug takes may vary. Sometimes your program will simply behave improperly. For example, it might produce the wrong output. Other times it will ask the computer to perform some task that is clearly a mistake, in which case your program will have a runtime error that stops it from executing. In this chapter, since your knowledge of Java is limited, generally the only type of logic error you will see is a mistake in program output from an incorrect `println` statement or method call.

We'll look at an example of a runtime error in the next section.

1.4 Procedural Decomposition

Brian Kernighan, coauthor of *The C Programming Language*, has said, "Controlling complexity is the essence of computer programming." People have only a modest capacity for detail. We can't solve complex problems all at once. Instead, we structure our problem solving by dividing the problem into manageable pieces and conquering each piece individually. We often use the term *decomposition* to describe this principle as applied to programming.

Decomposition

A separation into discernible parts, each of which is simpler than the whole.

With procedural programming languages like C, decomposition involves dividing a complex task into a set of subtasks. This is a very verb- or action-oriented approach, involving dividing up the overall action into a series of smaller actions. This technique is called *procedural decomposition*.

Common Programming Error

Not Closing a String Literal or Comment

Every string literal has to have an opening quote and a closing quote, but it's easy to forget the closing quotation mark. For example, you might say:

```
System.out.println("Hello, world!);
```

This produces three different error messages, even though there is only one underlying syntax error:

```
Hello.java:3: error: unclosed string literal
        System.out.println("hello world);
                           ^
Hello.java:3: error: ';' expected
        System.out.println("hello world);
                                       ^
Hello.java:5: error: reached end of file while parsing

    }
    ^
3 errors
```

In this case, the first error message is quite clear, including an arrow pointing at the beginning of the string literal that wasn't closed. The second error message was caused by the first. Because the string literal was not closed, the compiler didn't notice the right parenthesis and semicolon that appear at the end of the line.

A similar problem occurs when you forget to close a multiline comment by writing `*/`, as in the first line of the following program:

```
/* This is a bad program.

public class Bad {
   public static void main(String[] args){
       System.out.println("Hi there.");
  }
} /* end of program */
```

The preceding file is not a program; it is one long comment. Because the comment on the first line is not closed, the entire program is swallowed up.

Luckily, many Java editor programs color the parts of a program to help you identify them visually. Usually, if you forget to close a string literal or comment, the rest of your program will turn the wrong color, which can help you spot the mistake.

Java was designed for a different kind of decomposition that is more noun- or object-oriented. Instead of thinking of the problem as a series of actions to be performed, we think of it as a collection of objects that have to interact.

As a computer scientist, you should be familiar with both types of problem solving. This book begins with procedural decomposition and devotes many chapters to mastering various aspects of the procedural approach. Only after you have thoroughly practiced procedural programming will we turn our attention back to object decomposition and object-oriented programming.

As an example of procedural decomposition, consider the problem of baking a cake. You can divide this problem into the following subproblems:

- Make the batter.
- Bake the cake.
- Make the frosting.
- Frost the cake.

Each of these four tasks has details associated with it. To make the batter, for example, you follow these steps:

- Mix the dry ingredients.
- Cream the butter and sugar.
- Beat in the eggs.
- Stir in the dry ingredients.

Thus, you divide the overall task into subtasks, which you further divide into even smaller subtasks. Eventually, you reach descriptions that are so simple they require no further explanation (i.e., primitives).

A partial diagram of this decomposition is shown in Figure 1.2. "Make cake" is the highest-level operation. It is defined in terms of four lower-level operations called "Make batter," "Bake," "Make frosting," and "Frost cake." The "Make batter" operation is defined in terms of even lower-level operations, and the same could be done for the other three operations. This diagram is called a structure diagram and is intended to show how a problem is broken down into subproblems. In this diagram, you can also tell in what order operations are performed by reading from left to right. That is not true of most structure diagrams. To determine the actual order in which subprograms are performed, you usually have to refer to the program itself.

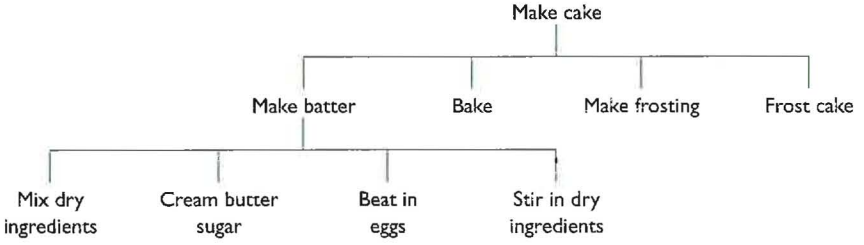


Figure 1.2 Decomposition of "Make cake" task

One final problem-solving term has to do with the process of programming. Professional programmers develop programs in stages. Instead of trying to produce a complete working program all at once, they choose some piece of the problem to implement first. Then they add another piece, and another, and another. The overall program is built up slowly, piece by piece. This process is known as *iterative enhancement* or *stepwise refinement.*

> **Iterative Enhancement**
>
> The process of producing a program in stages, adding new functionality at each stage. A key feature of each iterative step is that you can test it to make sure that piece works before moving on.

Now, let's look at a construct that will allow you to iteratively enhance your Java programs to improve their structure and reduce their redundancy: static methods.

Static Methods

Java is designed for objects, and programming in Java usually involves decomposing a problem into various objects, each with methods that perform particular tasks. You will see how this works in later chapters, but for now, we are going to explore procedural decomposition. We will postpone examining some of Java's details while we discuss programming in general.

Consider the following program, which draws two text boxes on the console:

```
1  public class DrawBoxes {
2      public static void main(String[] args) {
3          System.out.println("+------+");
4          System.out.println("|      |");
5          System.out.println("|      |");
6          System.out.println("+------+");
7          System.out.println();
8          System.out.println("+------+");
9          System.out.println("|      |");
10         System.out.println("|      |");
11         System.out.println("+------+");
12     }
13 }
```

The program works correctly, but the four lines used to draw the box appear twice. This redundancy is undesirable for several reasons. For example, you might wish to change the appearance of the boxes, in which case you'll have to make all of the edits twice. Also, you might wish to draw additional boxes, which would require you to type additional copies of (or copy and paste) the redundant lines.

A preferable program would include a Java command that specifies how to draw the box and then executes that command twice. Java doesn't have a "draw a box" command, but you can create one. Such a named command is called a *static method.*

> **Static Method**
>
> A block of Java statements that is given a name.

Static methods are units of procedural decomposition. We typically break a class into several static methods, each of which solves some piece of the overall problem. For example, here is a static method to draw a box:

```
public static void drawBox() {
    System.out.println("+------+");
    System.out.println("|      |");
    System.out.println("|      |");
    System.out.println("+------+");
}
```

You have already seen a static method called `main` in earlier programs. Recall that the `main` method has the following form:

```
public static void main(String[] args) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

The static methods you'll write have a similar structure:

```
public static void <name>() {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

The first line is known as the method header. You don't yet need to fully understand what each part of this header means in Java; for now, just remember that you'll need to write `public static void`, followed by the name you wish to give the method, followed by a set of parentheses. Briefly, here is what the words in the header mean:

- The keyword `public` indicates that this method is available to be used by all parts of your program. All methods you write will be public.
- The keyword `static` indicates that this is a static (procedural-style, not object-oriented) method. For now, all methods you write will be static, until you learn about defining objects in Chapter 8.
- The keyword `void` indicates that this method executes statements but does not produce any value. (Other methods you'll see later compute and return values.)
- `<name>` (e.g., `drawBox`) is the name of the method.
- The empty parentheses specify a list (in this case, an empty list) of values that are sent to your method as input; such values are called *parameters* and will not be included in your methods until Chapter 3.

Including the keyword `static` for each method you define may seem cumbersome. Other Java textbooks often do not discuss static methods as early as we do here; instead, they show other techniques for decomposing problems. But even though static methods require a bit of work to create, they are powerful and useful tools for improving basic Java programs.

After the header in our sample method, a series of `println` statements makes up the body of this static method. As in the `main` method, the statements of this method are executed in order from first to last.

By defining the method `drawBox`, you have given a simple name to this sequence of `println` statements. It's like saying to the Java compiler, "Whenever I tell you to 'drawBox,' I really mean that you should execute the `println` statements in the `drawBox` method." But the command won't actually be executed unless our `main` method explicitly says that it wants to do so. The act of executing a static method is called a *method call.*

Method Call

A command to execute another method, which causes all of the statements inside that method to be executed.

To execute the `drawBox` command, include this line in your program's `main` method:

```
drawBox();
```

Since we want to execute the `drawBox` command twice (to draw two boxes), the `main` method should contain two calls to the `drawBox` method. The following

program uses the `drawBox` method to produce the same output as the original `DrawBoxes` program:

```
1  public class DrawBoxes2 {
2      public static void main(String[] args) {
3          drawBox();
4          System.out.println();
5          drawBox();
6      }
7
8      public static void drawBox() {
9          System.out.println("+------+");
10         System.out.println("|      |");
11         System.out.println("|      |");
12         System.out.println("+------+");
13     }
14 }
```

Flow of Control

The most confusing thing about static methods is that programs with static methods do not execute sequentially from top to bottom. Rather, each time the program encounters a static method call, the execution of the program "jumps" to that static method, executes each statement in that method in order, and then "jumps" back to the point where the call began and resumes executing. The order in which the statements of a program are executed is called the program's *flow of control.*

> **Flow of Control**
>
> The order in which the statements of a Java program are executed.

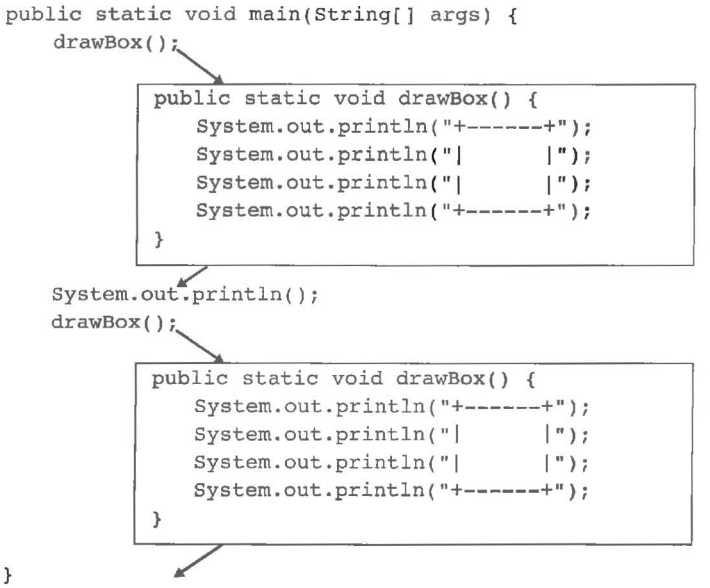
Let's look at the control flow of the `DrawBoxes2` program shown previously. It has two methods. The first method is the familiar `main` method, and the second is `drawBox`. As in any Java program, execution starts with the `main` method:

```
public static void main(String[] args) {
    drawBox();
    System.out.println();
    drawBox();
}
```

In a sense, the execution of this program is sequential: Each statement listed in the `main` method is executed in turn, from first to last.

But this `main` method includes two different calls on the `drawBox` method. This program will do three different things: execute `drawBox`, execute a `println`, then execute `drawBox` again.

The diagram below indicates the flow of control produced by this program.



Following the diagram, you can see that nine `println` statements are executed. First you transfer control to the `drawBox` method and execute its four statements. Then you return to `main` and execute its `println` statement. Then you transfer control a second time to `drawBox` and once again execute its four statements. Making these method calls is almost like copying and pasting the code of the method into the `main` method. As a result, this program has the exact same behavior as the nine-line `main` method of the `DrawBoxes` program:

```
public static void main(String[] args) {
    System.out.println("+------+");
    System.out.println("|      |");
    System.out.println("|      |");
    System.out.println("+------+");
    System.out.println();
    System.out.println("+------+");
    System.out.println("|      |");
    System.out.println("|      |");
    System.out.println("+------+");
}
```

This version is simpler in terms of its flow of control, but the first version avoids the redundancy of having the same `println` statements appear multiple times. It also gives a better sense of the structure of the solution. In the original version it is clear that there is a subtask called `drawBox` that is being performed twice. Also, while the last version of

the `main` method contains fewer lines of code than the `DrawBoxes2` program, consider what would happen if you wanted to add a third box to the output. You would have to add the five requisite `println` statements again, whereas in the programs that use the `drawBox` method you can simply add one more `println` and a third method call.

Java allows you to define methods in any order you like. It is a common convention to put the `main` method as either the first or last method in the class. In this textbook we will generally put `main` first, but the programs would behave the same if we switched the order. For example, the following modified program behaves identically to the previous `DrawBoxes2` program:

```
1  public class DrawBoxes3 {
2      public static void drawBox() {
3          System.out.println("+------+");
4          System.out.println("|      |");
5          System.out.println("|      |");
6          System.out.println("+------+");
7      }
8
9      public static void main(String[] args) {
10         drawBox();
11         System.out.println();
12         drawBox();
13     }
14 }
```

The `main` method is always the starting point for program execution, and from that starting point you can determine the order in which other methods are called.

Methods That Call Other Methods

The `main` method is not the only place where you can call another method. In fact, any method may call any other method. As a result, the flow of control can get quite complicated. Consider, for example, the following rather strange program. We use nonsense words ("foo," "bar," "baz," and "mumble") on purpose because the program is not intended to make sense.

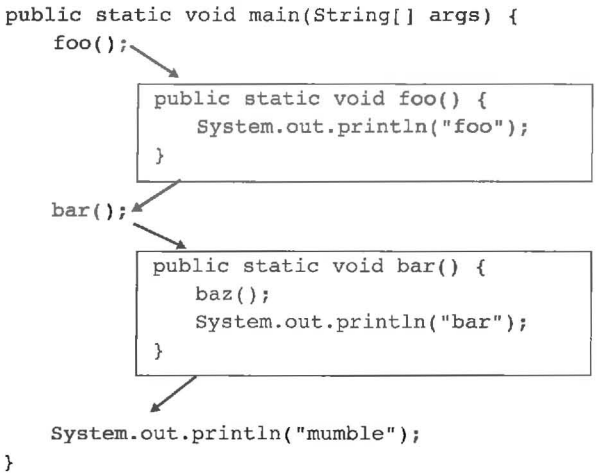
```
1  public class FooBarBazMumble {
2      public static void main(String[] args) {
3          foo();
4          bar();
5          System.out.println("mumble");
6      }
7
8      public static void foo() {
9          System.out.println("foo");
10     }
```

```
11
12      public static void bar() {
13          baz();
14          System.out.println("bar");
15      }
16
17      public static void baz() {
18          System.out.println("baz");
19      }
20  }
```

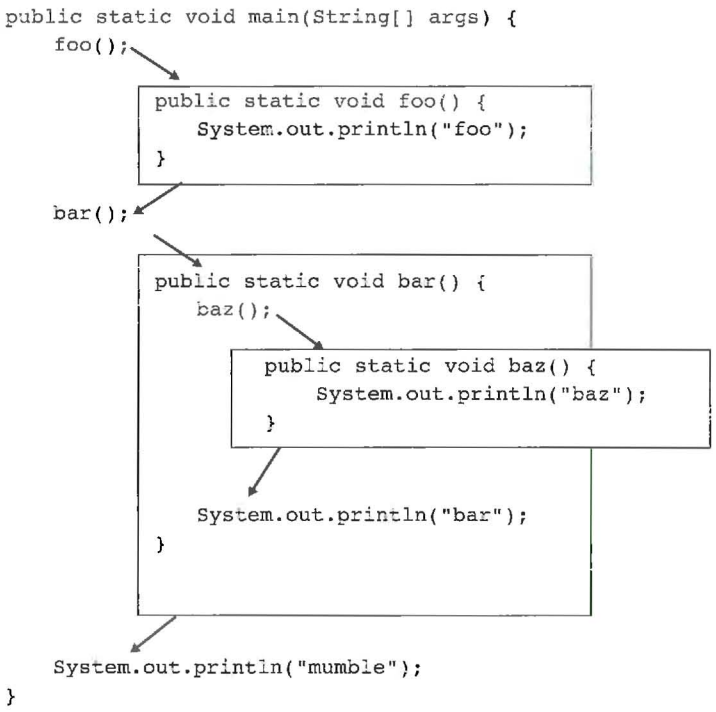
You can't tell easily what output this program produces, so let's explore in detail what the program is doing. Remember that Java always begins with the method called `main`. In this program, the `main` method calls the `foo` method and the `bar` method and then executes a `println` statement:

```
public static void main(String[] args) {
    foo();
    bar();
    System.out.println("mumble");
}
```

Each of these two method calls will expand into more statements. Let's first expand the calls on the `foo` and `bar` methods:



This helps to make our picture of the flow of control more complete, but notice that `bar` calls the `baz` method, so we have to expand that as well.



Finally, we have finished our picture of the flow of control of this program. It should make sense, then, that the program produces the following output:

```
foo
baz
bar
mumble
```

We will see a much more useful example of methods calling methods when we go through the case study at the end of the chapter.

Did You Know?

The New Hacker's Dictionary

Computer scientists and computer programmers use a lot of jargon that can be confusing to novices. A group of software professionals spearheaded by Eric Raymond have collected together many of the jargon terms in a book called *The New Hacker's Dictionary*. You can buy the book, or you can browse it online at Eric's website: http://catb.org/esr/jargon/html/frames.html.

For example, if you look up *foo*, you'll find this definition: "Used very generally as a sample name for absolutely anything, esp. programs and files." In

Continued on next page

Continued from previous page

other words, when we find ourselves looking for a nonsense word, we use "foo."

The New Hacker's Dictionary contains a great deal of historical information about the origins of jargon terms. The entry for *foo* includes a lengthy discussion of the combined term *foobar* and how it came into common usage among engineers.

If you want to get a flavor of what is there, check out the entries for *bug, hacker, bogosity,* and *bogo-sort.*

An Example Runtime Error

Runtime errors occur when a bug causes your program to be unable to continue executing. What could cause such a thing to happen? One example is if you asked the computer to calculate an invalid value, such as 1 divided by 0. Another example would be if your program tries to read data from a file that does not exist.

We haven't discussed how to compute values or read files yet, but there is a way you can "accidentally" cause a runtime error. The way to do this is to write a static method that calls itself. If you do this, your program will not stop running, because the method will keep calling itself indefinitely, until the computer runs out of memory. When this happens, the program prints a large number of lines of output, and then eventually stops executing with an error message called a `StackOverflowError`. Here's an example:

```
1  public class Infinite {
2      public static void main(String[] args) {
3          oops();
4      }
5
6      public static void oops() {
7          System.out.println("Make it stop!");
8          oops();
9      }
10 }
```

This ill-fated program produces the following output (with large groups of identical lines represented by "..."):

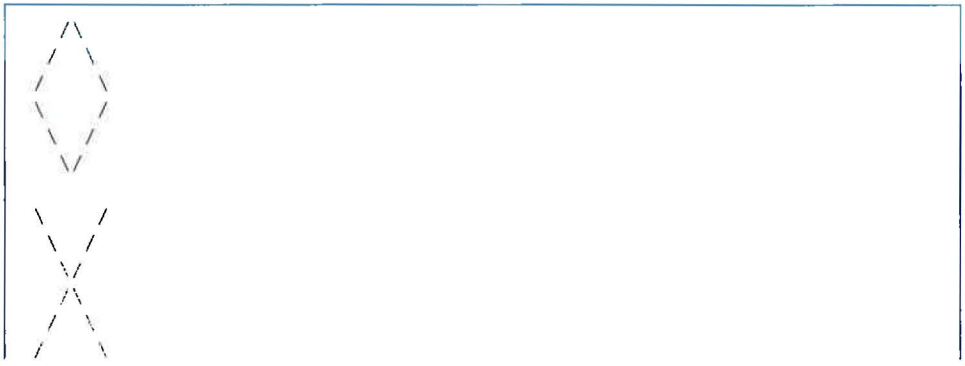
```
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
```

```
Make it stop!
Make it stop!
...
Make it stop!
Exception in thread "main" java.lang.StackOverflowError
        at sun.nio.cs.SingleByteEncoder.encodeArrayLoop(Unknown Source)
        at sun.nio.cs.SingleByteEncoder.encodeLoop(Unknown Source)
        at java.nio.charset.CharsetEncoder.encode(Unknown Source)
        at sun.nio.cs.StreamEncoder$CharsetSE.implWrite(Unknown Source)
        at sun.nio.cs.StreamEncoder.write(Unknown Source)
        at java.io.OutputStreamWriter.write(Unknown Source)
        at java.io.BufferedWriter.flushBuffer(Unknown Source)
        at java.io.PrintStream.newLine(Unknown Source)
        at java.io.PrintStream.println(Unknown Source)
        at Infinite.oops(Infinite.java:7)
        at Infinite.oops(Infinite.java:8)
        at Infinite.oops(Infinite.java:8)
        at Infinite.oops(Infinite.java:8)
        at ...
```

Runtime errors are, unfortunately, something you'll have to live with as you learn to program. You will have to carefully ensure that your programs not only compile successfully, but do not contain any bugs that will cause a runtime error. The most common way to catch and fix runtime errors is to run the program several times to test its behavior.

1.5 Case Study: `DrawFigures`

Earlier in the chapter, you saw a program called `DrawFigures1` that produced the following output:



```
   /\
  /  \
 /    \
+------+
|      |
|      |
+------+
|United|
|States|
+------+
|      |
|      |
+------+
   /\
  /  \
 /    \
```

It did so with a long sequence of `println` statements in the `main` method. In this section you'll improve the program by using static methods for procedural decomposition to capture structure and eliminate redundancy. The redundancy might be more obvious, but let's start by improving the way the program captures the structure of the overall task.

Structured Version

If you look closely at the output, you'll see that it has a structure that would be desirable to capture in the program structure. The output is divided into three subfigures: the diamond, the X, and the rocket.

You can better indicate the structure of the program by dividing it into static methods. Since there are three subfigures, you can create three methods, one for each subfigure. The following program produces the same output as `DrawFigures1`:

```
1  public class DrawFigures2 {
2      public static void main(String[] args) {
3          drawDiamond();
4          drawX();
5          drawRocket();
6      }
7
8      public static void drawDiamond() {
9          System.out.println("    /\\");
10         System.out.println("   /  \\");
11         System.out.println("  /    \\");
12         System.out.println(" \\     /");
13         System.out.println("  \\   /");
14         System.out.println("   \\ /");
15         System.out.println();
16     }
17
```

```
18      public static void drawX() {
19          System.out.println(" \\     /");
20          System.out.println("  \\   /");
21          System.out.println("   \\ /");
22          System.out.println("    /\\");
23          System.out.println("   /  \\");
24          System.out.println(" /     \\");
25          System.out.println();
26      }
27
28      public static void drawRocket() {
29          System.out.println("    /\\");
30          System.out.println("   /  \\");
31          System.out.println(" /     \\");
32          System.out.println("+------+");
33          System.out.println("|      |");
34          System.out.println("|      |");
35          System.out.println("+------+");
36          System.out.println("|United|");
37          System.out.println("|States|");
38          System.out.println("+------+");
39          System.out.println("|      |");
40          System.out.println("|      |");
41          System.out.println("+------+");
42          System.out.println("    /\\");
43          System.out.println("   /  \\");
44          System.out.println(" /     \\");
45      }
46  }
```

The program appears in a class called `DrawFigures2` and has four static methods defined within it. The first static method is the usual `main` method, which calls three methods. The three methods called by `main` appear next.

Figure 1.3 is a structure diagram for this version of the program. Notice that it has two levels of structure. The overall problem is broken down into three subtasks.

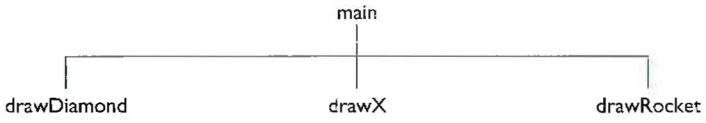


Figure 1.3 Decomposition of `DrawFigures2`

Final Version without Redundancy

The program can still be improved. Each of the three subfigures has individual elements, and some of those elements appear in more than one of the three subfigures. The program prints the following redundant group of lines several times:

```
  /\
 /  \
/    \
```

A better version of the preceding program adds an additional method for each redundant section of output. The redundant sections are the top and bottom halves of the diamond shape and the box used in the rocket. Here is the improved program:

```
1  public class DrawFigures3 {
2      public static void main(String[] args) {
3          drawDiamond();
4          drawX();
5          drawRocket();
6      }
7
8      public static void drawDiamond() {
9          drawCone();
10         drawV();
11         System.out.println();
12     }
13
14     public static void drawX() {
15         drawV();
16         drawCone();
17         System.out.println();
18     }
19
20     public static void drawRocket() {
21         drawCone();
22         drawBox();
23         System.out.println("|United|");
24         System.out.println("|States|");
25         drawBox();
26         drawCone();
27         System.out.println();
28     }
29
```

```
30      public static void drawBox() {
31          System.out.println("+------+");
32          System.out.println("|      |");
33          System.out.println("|      |");
34          System.out.println("+------+");
35      }
36
37      public static void drawCone() {
38          System.out.println("   /\\");
39          System.out.println("  /  \\");
40          System.out.println(" /    \\");
41      }
42
43      public static void drawV() {
44          System.out.println(" \\    /");
45          System.out.println("  \\  /");
46          System.out.println("   \\/");
47      }
48  }
```

This program, now called `DrawFigures3`, has seven static methods defined within it. The first static method is the usual `main` method, which calls three methods. These three methods in turn call three other methods, which appear next.

Analysis of Flow of Execution

The structure diagram in Figure 1.4 shows which static methods `main` calls and which static methods each of them calls. As you can see, this program has three levels of structure and two levels of decomposition. The overall task is split into three subtasks, each of which has two subtasks.

A program with methods has a more complex flow of control than one without them, but the rules are still fairly simple. Remember that when a method is called, the computer executes the statements in the body of that method. Then the computer proceeds to the next statement after the method call. Also remember that the computer always starts with the `main` method, executing its statements from first to last.

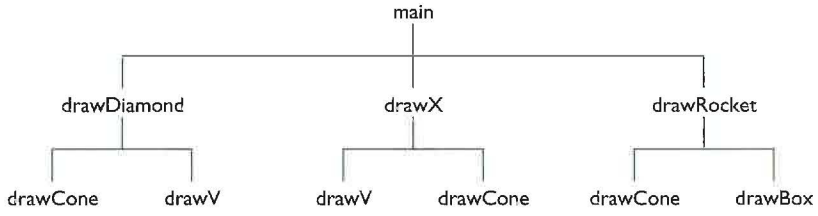


Figure 1.4 Decomposition of `DrawFigures3`

So, to execute the DrawFigures3 program, the computer first executes its main method. That, in turn, first executes the body of the method drawDiamond. drawDiamond executes the methods drawCone and drawV (in that order). When drawDiamond finishes executing, control shifts to the next statement in the body of the main method: the call to the drawX method.

A complete breakdown of the flow of control from static method to static method in DrawFigures3 follows:

```
 1st  main
 2nd      drawDiamond
 3rd          drawCone
 4th          drawV
 5th      drawX
 6th          drawV
 7th          drawCone
 8th      drawRocket
 9th          drawCone
10th          drawBox
11th          drawBox
12th          drawCone
```

Recall that the order in which you define methods does not have to parallel the order in which they are executed. The order of execution is determined by the body of the main method and by the bodies of methods called from main. A static method declaration is like a dictionary entry—it defines a word, but it does not specify how the word will be used. The body of this program's main method says to first execute drawDiamond, then drawX, then drawRocket. This is the order of execution, regardless of the order in which the methods are defined.

Java allows you to define methods in any order you like. Starting with main at the top and working down to lower and lower-level methods is a popular approach to take, but many people prefer the opposite, placing the low-level methods first and main at the end. Java doesn't care what order you use, so you can decide for yourself and do what you think is best. Consistency is important, though, so that you can easily find a method later in a large program.

It is important to note that the programs DrawFigures1, DrawFigures2, and DrawFigures3 produce exactly the same output to the console. While DrawFigures1 may be the easiest program for a novice to read, DrawFigures2 and particularly DrawFigures3 have many advantages over it. For one, a well-structured solution is easier to comprehend, and the methods themselves become a means of explaining the program. Also, programs with methods are more flexible and can more easily be adapted to similar but different tasks. You can take the seven methods defined in DrawFigures3 and write a new program to produce a larger and more complex output. Building static methods to create new commands increases your flexibility without adding unnecessary complication. For example, you could replace the main

method with a version that calls the other methods in the following new order. What output would it produce?

```
public static void main(String[] args) {
    drawCone();
    drawCone();
    drawRocket();
    drawX();
    drawRocket();
    drawDiamond();
    drawBox();
    drawDiamond();
    drawX();
    drawRocket();
}
```

Chapter Summary

Computers execute sets of instructions called programs. Computers store information internally as sequences of 0s and 1s (binary numbers).

Programming and computer science deal with algorithms, which are step-by-step descriptions for solving problems.

Java is a modern object-oriented programming language developed by Sun Microsystems, now owned by Oracle Corporation, that has a large set of libraries you can use to build complex programs.

A program is translated from text into computer instructions by another program called a compiler. Java's compiler turns Java programs into a special format called Java bytecodes, which are executed using a special program called the Java Runtime Environment.

Java programmers typically complete their work using an editor called an Integrated Development Environment (IDE). The commands may vary from environment to environment, but the same three-step process is always involved:

1. Type in a program as a Java class.
2. Compile the program file.
3. Run the compiled version of the program.

Java uses a command called `System.out.println` to display text on the console screen.

Written words in a program can take different meanings. Keywords are special reserved words that are part of the language. Identifiers are words defined by the programmer to name entities in the program. Words can also be put into strings, which are pieces of text that can be printed to the console.

Java programs that use proper spacing and layout are more readable to programmers. Readability is also improved by writing notes called comments inside the program.

The Java language has a syntax, or a legal set of commands that can be used. A Java program that does not follow the proper syntax will not compile. A program that does compile but that is written incorrectly may still contain errors called exceptions that occur when the program runs. A third kind of error is a logic or intent error. This kind of error occurs when the program runs but does not do what the programmer intended.

Commands in programs are called statements. A class can group statements into larger commands called static methods. Static methods help the programmer group code into reusable pieces. An important static method that must be part of every program is called `main`.

Iterative enhancement is the process of building a program piece by piece, testing the program at each step before advancing to the next.

Complex programming tasks should be broken down into the major tasks the computer must perform. This process is called procedural decomposition. Correct use of static methods aids procedural decomposition.

Self-Check Problems

Section 1.1: Basic Computing Concepts

1. Why do computers use binary numbers?
2. Convert each of the following decimal numbers into its equivalent binary number:
   a. 6
   b. 44
   c. 72
   d. 131
3. What is the decimal equivalent of each of the following binary numbers?
   a. 100
   b. 1011
   c. 101010
   d. 1001110
4. In your own words, describe an algorithm for baking cookies. Assume that you have a large number of hungry friends, so you'll want to produce several batches of cookies!
5. What is the difference between the file `MyProgram.java` and the file `MyProgram.class`?

Section 1.2: And Now—Java

6. Which of the following can be used in a Java program as identifiers?

```
println          first-name    AnnualSalary    "hello"    ABC
42isTheAnswer    for           sum_of_data     _average   B4
```

7. Which of the following is the correct syntax to output a message?

```
a. System.println(Hello, world!);
b. System.println.out('Hello, world!');
c. System.println("Hello, world!");
```

d. `System.out.println("Hello, world!");`

e. `Out.system.println"(Hello, world!)";`

8. What is the output produced from the following statements?

```
System.out.println("\"Quotes\"");
System.out.println("Slashes \\//");
System.out.println("How '\"confounding' \"\\\\" it is!");
```

9. What is the output produced from the following statements?

```
System.out.println("name\tage\theight");
System.out.println("Archie\t17\t5'9\"");
System.out.println("Betty\t17\t5'6\"");
System.out.println("Jughead\t16\t6'");
```

10. What is the output produced from the following statements?

```
System.out.println("Shaq is 7'1");
System.out.println("The string \"\" is an empty message.");
System.out.println("\\'\"\"");
```

11. What is the output produced from the following statements?

```
System.out.println("\ta\tb\tc");
System.out.println("\\\\");
System.out.println("'");
System.out.println("\"\"\"");
System.out.println("C:\nin\the downward spiral");
```

12. What is the output produced from the following statements?

```
System.out.println("Dear \"DoubleSlash\" magazine,");
System.out.println();
System.out.println("\tYour publication confuses me. Is it");
System.out.println("a \\\\ slash or a //// slash?");
System.out.println("\nSincerely,");
System.out.println("Susan \"Suzy\" Smith");
```

13. What series of `println` statements would produce the following output?

```
"Several slashes are sometimes seen,"
said Sally. "I've said so," See?
\ / \\ // \\\ ///
```

14. What series of `println` statements would produce the following output?

```
This is a test of your
knowledge of "quotes" used
in 'string literals.'

You're bound to "get it right"
if you read the section on
''quotes.''
```

15. Write a `println` statement that produces the following output:

```
/ \ // \\ /// \\\
```

16. Rewrite the following code as a series of equivalent `System.out.println` statements (i.e., without any `System.out.print` statements):

```
System.out.print("Twas ");
System.out.print("brillig and the");
System.out.println(" ");
System.out.print(" slithy toves did");
System.out.print(" ");
System.out.println("gyre and");
System.out.println("gimble");
System.out.println();
System.out.println( "in the wabe.");
```

17. What is the output of the following program? Note that the program contains several comments.

```
1  public class Commentary {
2      public static void main(String[] args) {
3          System.out.println("some lines of code");
4          System.out.println("have // characters on them");
5          System.out.println("which means ");  // that they are comments
6          // System.out.println("written by the programmer.");
7
8          System.out.println("lines can also");
9          System.out.println("have /* and */ characters");
10         /* System.out.println("which represents");
11         System.out.println("a multi-line style");
12         */ System.out.println("of comment.");
13     }
14 }
```

Section 1.3: Program Errors

18. Name the three errors in the following program:

```
1  public MyProgram {
2      public static void main(String[] args) {
3          System.out.println("This is a test of the")
4          System.out.Println("emergency broadcast system.");
5      }
6  }
```

19. Name the four errors in the following program:

```
1  public class SecretMessage {
2      public static main(string[] args) {
3          System.out.println("Speak friend");
```

```
4          System.out.println("and enter);
5
6  }
```

20. Name the four errors in the following program:

```
1  public class FamousSpeech
2      public static void main(String[]) {
3          System.out.println("Four score and seven years ago,");
4          System.out.println("our fathers brought forth on");
5          System.out.println("this continent a new nation");
6          System.out.println("conceived in liberty,");
7          System.out.println("and dedicated to the proposition");
8          System.out.println("that");    /* this part should
9          System.out.println("all");        really say,
10         System.out.println("men");        "all PEOPLE!" */
11         System.out.println("are";
12         System.out.println("created");
13         System.out.println("equal");
14     }
15 }
```

Section 1.4: Procedural Decomposition

21. Which of the following method headers uses the correct syntax?

a. `public static example() {`
b. `public static void example() {`
c. `public void static example() {`
d. `public static example void[] {`
e. `public void static example{} {`

22. What is the output of the following program? (You may wish to draw a structure diagram first.)

```
1  public class Tricky {
2      public static void main(String[] args) {
3          message1();
4          message2();
5          System.out.println("Done with main.");
6      }
7
8      public static void message1() {
9          System.out.println("This is message1.");
10     }
11
12     public static void message2() {
13         System.out.println("This is message2.");
14         message1();
```

```
15          System.out.println("Done with message2.");
16      }
17  }
```

23. What is the output of the following program? (You may wish to draw a structure diagram first.)

```
1  public class Strange {
2      public static void first() {
3          System.out.println("Inside first method");
4      }
5
6      public static void second() {
7          System.out.println("Inside second method");
8          first();
9      }
10
11     public static void third() {
12         System.out.println("Inside third method");
13         first();
14         second();
15     }
16
17     public static void main(String[] args) {
18         first();
19         third();
20         second();
21         third();
22     }
23 }
```

24. What would have been the output of the preceding program if the `third` method had contained the following statements?

```
public static void third() {
    first();
    second();
    System.out.println("Inside third method");
}
```

25. What would have been the output of the `Strange` program if the `main` method had contained the following statements? (Use the original version of `third`, not the modified version from the most recent exercise.)

```
public static void main(String[] args) {
    second();
    first();
    second();
    third();
}
```

26. What is the output of the following program? (You may wish to draw a structure diagram first.)

```
1  public class Confusing {
2      public static void method2() {
3          method1();
4          System.out.println("I am method 2.");
5      }
6
7      public static void method3() {
8          method2();
9          System.out.println("I am method 3.");
10         method1();
11     }
12
13     public static void method1() {
14         System.out.println("I am method 1.");
15     }
16
17     public static void main(String[] args) {
18         method1();
19         method3();
20         method2();
21         method3();
22     }
23 }
```

27. What would have been the output of the preceding program if the `method3` method had contained the following statements?

```
public static void method3() {
    method1();
    method2();
    System.out.println("I am method 3.");
}
```

28. What would have been the output of the `Confusing` program if the `main` method had contained the following statements? (Use the original version of `method3`, not the modified version from the most recent exercise.)

```
public static void main(String[] args) {
    method2();
    method1();
    method3();
    method2();
}
```

29. The following program contains at least 10 syntax errors. What are they?

```
1  public class LotsOf Errors {
2      public static main(String args) {
3          System.println(Hello, world!);
```

```
4           message()
5       }
6
7       public static void message {
8           System.out println("This program surely cannot ";
9           System.out.println("have any "errors" in it");
10      }
```

30. Consider the following program, saved into a file named `Example.java`:

```
1  public class Example {
2      public static void displayRule() {
3          System.out.println("The first rule ");
4          System.out.println("of Java Club is,");
5          System.out.println();
6          System.out.println("you do not talk about Java Club.");
7      }
8
9      public static void main(String[] args) {
10         System.out.println("The rules of Java Club.");
11         displayRule();
12         displayRule();
13     }
14 }
```

What would happen if each of the following changes were made to the `Example` program? For example, would there be no effect, a syntax error, or a different program output? Treat each change independently of the others.

a. Change line 1 to: `public class Demonstration`
b. Change line 9 to: `public static void MAIN(String[] args) {`
c. Insert a new line after line 11 that reads: `System.out.println();`
d. Change line 2 to: `public static void printMessage() {`
e. Change line 2 to: `public static void showMessage() {` and change lines 11 and 12 to: `showMessage();`
f. Replace lines 3–4 with: `System.out.println("The first rule of Java Club is,");`

31. The following program is legal under Java's syntax rules, but it is difficult to read because of its layout and lack of comments. Reformat it using the rules given in this chapter, and add a comment header at the top of the program.

```
1     public
2  class GiveAdvice{ public static
3  void main (String[ ]args){ System.out.println (
4
5  "Programs can be easy or");  System.out.println(
6    "difficult to read, depending"
7  );  System.out.println("upon their format.")
8            ;System.out.println();System.out.println(
9  "Everyone, including yourself,");
10 System.out.println
11 ("will be happier if you choose");
```

```
12          System.out.println("to format your programs."
13  );    }
14    }
```

32. The following program is legal under Java's syntax rules, but it is difficult to read because of its layout and lack of comments. Reformat it using the rules given in this chapter, and add a comment header at the top of the program.

```
1  public
2  class Messy{public
3  static void main(String[]args){message ()
4    ;System.out.println()    ; message ( );}   public static void
5  message() { System.out.println(
6      "I really wish that"
7      );System.out.println
8  ("I had formatted my source")
9  ;System.out.println("code correctly!");}}
```

Exercises

1. Write a complete Java program called `Stewie` that prints the following output:

```
//////////////////////
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
```

2. Write a complete Java program called `Spikey` that prints the following output:

```
  \/
 \\//
\\\///
///\\\
 //\\
  /\
```

3. Write a complete Java program called `WellFormed` that prints the following output:

```
A well--formed Java program has
a main method with { and }
braces.

A System.out.println statement
has ( and ) and usually a
String that starts and ends
with a " character.
(But we type \" instead!)
```

4. Write a complete Java program called `Difference` that prints the following output:

```
What is the difference between
a ' and a "? Or between a " and a \"?

One is what we see when we're typing our program.
The other is what appears on the "console."
```

5. Write a complete Java program called `MuchBetter` that prints the following output:

```
A "quoted" String is
'much' better if you learn
the rules of "escape sequences."
Also, "" represents an empty String.
Don't forget: use \" instead of " !
'' is not the same as "
```

6. Write a complete Java program called **`Meta`** whose output is the text that would be the source code of a Java program that prints "Hello, world!" as its output.

7. Write a complete Java program called `Mantra` that prints the following output. Use at least one static method besides `main`.

```
There's one thing every coder must understand:
The System.out.println command.

There's one thing every coder must understand:
The System.out.println command.
```

8. Write a complete Java program called `Stewie2` that prints the following output. Use at least one static method besides `main`.

```
//////////////////////
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
```

9. Write a program called `Egg` that displays the following output:

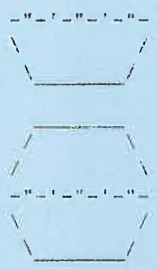
```
  _______
 /       \
/         \
-"-'-"-'-"-
\         /
 \_______/
```

10. Modify the program from the previous exercise to become a new program `Egg2` that displays the following output. Use static methods as appropriate.

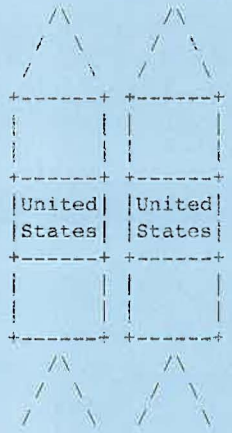
```
  _______
 /       \
/         \
\         /
 \_______/

-"-'-"-'-"-

  _______
 /       \
/         \
\         /
 \_______/
```



11. Write a Java program called `TwoRockets` that generates the following output. Use static methods to show structure and eliminate redundancy in your solution. Note that there are two rocket ships next to each other. What redundancy can you eliminate using static methods? What redundancy cannot be eliminated?



12. Write a program called `FightSong` that produces this output. Use at least two static methods to show structure and eliminate redundancy in your solution.

```
Go, team, go!
You can do it.

Go, team, go!
You can do it.
You're the best,
In the West.
Go, team, go!
You can do it.

Go, team, go!
You can do it.
You're the best,
in the West.
Go, team, go!
You can do it.

Go, team, go!
You can do it.
```

13. Write a Java program called `StarFigures` that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.

```
*****
*****
 * *
  *
 * *

*****
*****
 * *
  *
 * *
*****
*****

  *
  *
  *
*****
*****
 * *
  *
 * *
```

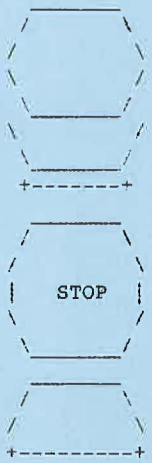
14. Write a Java program called `Lanterns` that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.

```
    *****
  *********
*************

    *****
  *********
*************
* | | | | | *
*************
    *****
  *********
*************

    *****
  *********
*************
    *****
* | | | | | *
* | | | | | *
    *****
    *****
```

15. Write a Java program called `EggStop` that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.



16. Write a program called `Shining` that prints the following line of output 1000 times:

```
All work and no play makes Jack a dull boy.
```

You should not write a program that uses 1000 lines of source code; use methods to shorten the program. What is the shortest program you can write that will produce the 1000 lines of output, using only the material from this chapter?

Programming Projects

1. Write a program to spell out MISSISSIPPI using block letters like the following (one per line):

```
M     M    IIIII     SSSSS     PPPPPP
MM   MM      I      S     S    P     P
M M M M      I      S          P     P
M  M  M      I       SSSSS     PPPPPP
M     M      I            S    P
M     M      I      S     S    P
M     M    IIIII     SSSSS     P
```

2. Sometimes we write similar letters to different people. For example, you might write to your parents to tell them about your classes and your friends and to ask for money; you might write to a friend about your love life, your classes, and your hobbies; and you might write to your brother about your hobbies and your friends and to ask for money. Write a program that prints similar letters such as these to three people of your choice. Each letter should have at least one paragraph in common with each of the other letters. Your main program should have three method calls, one for each of the people to whom you are writing. Try to isolate repeated tasks into methods.

3. Write a program that produces as output the lyrics of the song, "There Was an Old Lady Who Swallowed a Fly," by Simms Taback. Use methods for each verse and the refrain. Here are the song's complete lyrics:

```
There was an old lady who swallowed a fly
I don't know why she swallowed the fly
Perhaps she'll die
But it's only a fly
I think I'll cry
```

```
She gulped it out of the sky
Oh, my!

There was an old lady who swallowed a spider
That wiggled and jiggled and tickled inside her.
She swallowed the spider to catch the fly
I don't know why
She swallowed the fly
Perhaps she'll die.
Gone to the by and by
Sigh

There was an old lady who swallowed a bird.
How absurd! She swallowed a bird!
She swallowed the bird to catch the spider.
She swallowed the spider to catch the fly
I don't know why
She swallowed the fly
Perhaps she'll die.
She'll leave us high and dry.

There was an old lady who swallowed a cat.
Imagine that! She swallowed a cat.
She swallowed the cat to catch the bird.
She swallowed the bird to catch the spider.
She swallowed the spider to catch the fly.
I don't know why
She swallowed the fly
Perhaps she'll die.
I hope it's a lie.

There was an old lady who swallowed a dog.
She went whole hog to swallow the dog.
She swallowed the dog to catch the cat.
She swallowed the cat to catch the bird.
She swallowed the bird to catch the spider.
She swallowed the spider to catch the fly.
I don't know why
She swallowed the fly
Perhaps she'll die.
There's a tear in my eye.

There was an old lady who swallowed a cow.
I don't know how she swallowed the cow.
She swallowed the cow to catch the dog.
She swallowed the dog to catch the cat.
She swallowed the cat to catch the bird.
She swallowed the bird to catch the spider.
She swallowed the spider to catch the fly.
I don't know why
She swallowed the fly
```

```
Perhaps she'll die.
I'd rather have ham on rye.
And she had a frog on the sly.
She did it in one try.

There was an old lady who swallowed a horse.
She died, of course.
It was the last course.
I'm filled with remorse.
What's left to say...
Even the artist is crying.
We'll miss her dearly.
It is such a loss.
She had no time to floss.
She missed out on the sauce.

Moral: Never swallow a horse.
```

4. Write a program that produces as output the words of "The Twelve Days of Christmas." (Static methods simplify this task.) Here are the first two verses and the last verse of the song:

```
On the first day of Christmas,
my true love sent to me
a partridge in a pear tree.

On the second day of Christmas,
my true love sent to me
two turtle doves, and
a partridge in a pear tree.
...

On the twelfth day of Christmas,
my true love sent to me
Twelve drummers drumming,
eleven pipers piping,
ten lords a-leaping,
nine ladies dancing,
eight maids a-milking,
seven swans a-swimming,
six geese a-laying,
five golden rings,
four calling birds,
three French hens,
two turtle doves, and
a partridge in a pear tree.
```

5. Write a program that produces as output the words of "The House That Jack Built." Use methods for each verse and for repeated text. Here are lyrics to use:

```
This is the house that Jack built.

This is the malt
That lay in the house that Jack built.
```

```
This is the rat,
That ate the malt
That lay in the house that Jack built.

This is the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

This is the dog,
That worried the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

This is the cow with the crumpled horn,
That tossed the dog,
That worried the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.

This is the maiden all forlorn
That milked the cow with the crumpled horn,
That tossed the dog,
That worried the cat,
That killed the rat,
That ate the malt
That lay in the house that Jack built.
```

6. Write a program that produces as output the words of "Bought Me a Cat." Use methods for each verse and for repeated text. Here are the song's complete lyrics:

```
Bought me a cat and the cat pleased me,
I fed my cat under yonder tree.
Cat goes fiddle-i-fee.

Bought me a hen and the hen pleased me,
I fed my hen under yonder tree.
Hen goes chimmy-chuck, chimmy-chuck,
Cat goes fiddle-i-fee.

Bought me a duck and the duck pleased me,
I fed my duck under yonder tree.
Duck goes quack, quack,
Hen goes chimmy-chuck, chimmy-chuck,
Cat goes fiddle-i-fee.

Bought me a goose and the goose pleased me,
I fed my goose under yonder tree.
Goose goes hissy, hissy,
Duck goes quack, quack,
```

```
Hen goes chimmy-chuck, chimmy-chuck,
Cat goes fiddle-i-fee.

Bought me a sheep and the sheep pleased me,
I fed my sheep under yonder tree.
Sheep goes baa, baa,
Goose goes hissy, hissy,
Duck goes quack, quack,
Hen goes chimmy-chuck, chimmy-chuck,
Cat goes fiddle-i-fee.
```

7. Write a program that produces as output the words of the following silly song. Use methods for each verse and for repeated text. Here are the song's complete lyrics:

```
I once wrote a program that wouldn't compile
I don't know why it wouldn't compile,
My TA just smiled.

My program did nothing
So I started typing.
I added System.out.println("I <3 coding"),
I don't know why it wouldn't compile,
My TA just smiled.

"Parse error," cried the compiler
Luckily I'm such a code baller.
I added a backslash to escape the quotes,
I added System.out.println("I <3 coding"),
I don't know why it wouldn't compile,
My TA just smiled.

Now the compiler wanted an identifier
And I thought the situation was getting dire.
I added a main method with its String[] args,
I added a backslash to escape the quotes,
I added System.out.println("I <3 coding"),
I don't know why it wouldn't compile,
My TA just smiled.

Java complained it expected an enum
Boy, these computers really are dumb!
I added a public class and called it Scum,
I added a main method with its String[] args,
I added a backslash to escape the quotes,
I added System.out.println("I <3 coding"),
I don't know why it wouldn't compile,
My TA just smiled.
```