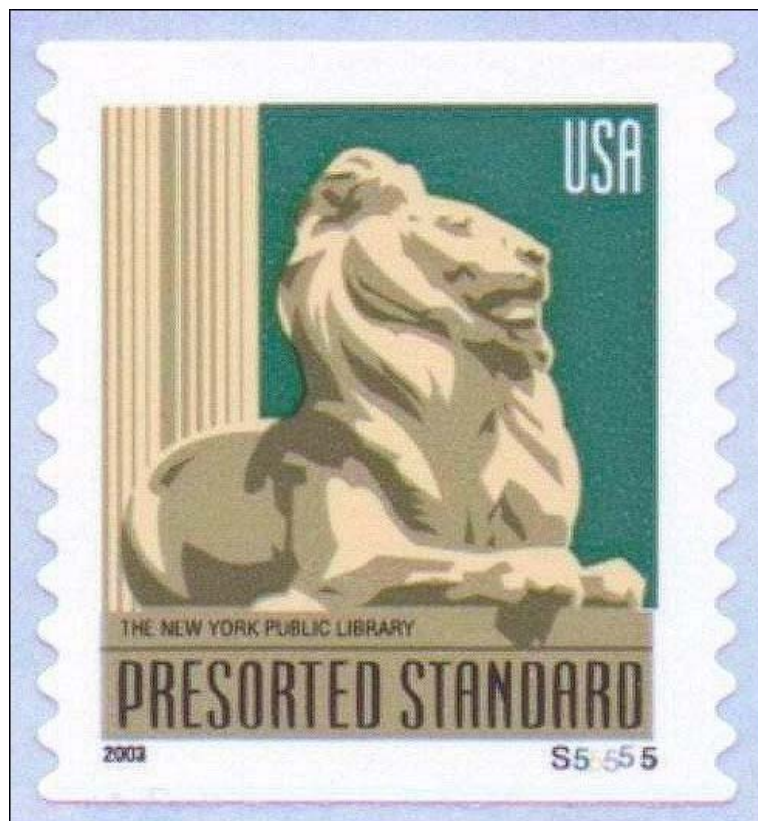# The Art and Science of *Java*

Preliminary Draft

Eric S. Roberts
Stanford University
Stanford, California
January 2006

Preface

This text is an early draft for a general introductory textbook in computer science—a Java-based version of my 1995 textbook *The Art and Science of C*. My hope is that I can use much of the existing material in writing the new book, although quite a bit of the material and overall organization have to change. At this point, the material is still in a preliminary form, and the feedback I get from those of you who are taking this course will almost certainly lead to some changes before the book is published.

One of the central features of the text is that it incorporates the work of the Association of Computing Machinery's Java Task Force, which was convened in 2004 with the following charter:

> To review the Java language, APIs, and tools from the perspective of introductory computing education and to develop a stable collection of pedagogical resources that will make it easier to teach Java to first-year computing students without having those students overwhelmed by its complexity.

I am grateful to my colleagues on the Task Force—Kim Bruce, Robb Cutler, James H. Cross II, Scott Grissom, Karl Klee, Susan Rodger, Fran Trees, Ian Utting, and Frank Yellin—for all their hard work over the past year, as well as to the National Science Foundation, the ACM Education Board, the SIGCSE Special Projects Fund for their financial support.

I also want to thank the participants in last year's CS 298 seminar—Andrew Adams, Andy Aymeloglu, Kurt Berglund, Seyed Dorminani-Tabatabaei, Erik Forslin, Alex Himel, Tom Hurlbutt, Dave Myszewski, Ann Pan, Vishakha Parvate, Cynthia Wang, Paul Wilkins, and Julie Zhuo for helping me work through these ideas. In addition, I would like to thank my CS 106A TA Brandon Burr and all the hardworking section-leaders for taking on the challenge of helping to teach a course with a just-in-time approach to the materials.

Particularly because my wife Lauren Rusk (who has edited all of my books) has not yet had her chance to work her wonderful magic on the language, you may still find some rough edges, awkward constructions, and places where real improvement is needed. Writing is, after all, at least as difficult as programming and requires just as much testing to get everything right. If you let me know when things are wrong, I think we'll end up with a textbook and a course that are exciting, thorough, and practical.

Thanks in advance for all your help.

Eric Roberts
Professor of Computer Science
Stanford University
September 2005

Table of Contents

A note on the cover image: The cover of The Art and Science of C showed a picture of Patience, one of the two stone lions that guard the entrance to the New York Public Library. Addison-Wesley and I chose that image both to emphasize the library-based approach adopted by the text and because patience is an essential skill in programming. In 2003, the United States Postal Service decided to put Patience on a stamp, which gave those of us who have a special attachment to that lion a great deal of inner pleasure.

Chapter 1
Introduction

*[The Analytical Engine offers] a new, a vast, and a powerful language . . .
for the purposes of mankind.*

> — Augusta Ada Byron, Lady Lovelace,
> *The Sketch of the Analytical Engine
> Invented by Charles Babbage,* 1843

Augusta Ada Byron, Lady Lovelace (1815–1852)

Augusta Ada Byron, the daughter of English poet Lord Byron, was encouraged b ests in science and mathematics at a time when few women were allowed to study those subjects. At the age of 17, Ada met Charles Babbage, a prominent English scientist who devoted his life to designing machines for carrying out mathematical computations—machines that he was never able to complete. Ada was firmly convinced of the potential of Babbage's Analytical Engine and wrote extensive notes on its design, along with several complex mathematical programs that have led many people to characterize her as the first programmer. In 1980, the U.S. Department of Defense named the programming language Ada in her honor.

Given our vantage point at the beginning of the 21st century, it is hard to believe that computers did not even exist in 1940. Computers are everywhere today, and it is the popular wisdom, at least among headline writers, to say that we live in the computer age.

1.1 A brief history of computing

In a certain sense, computing has been around since ancient times. Much of early mathematics was devoted to solving computational problems of practical importance, such as monitoring the number of animals in a herd, calculating the area of a plot of land, or recording a commercial transaction. These activities required people to develop new computational techniques and, in some cases, to invent calculating machines to help in the process. For example, the abacus, a simple counting device consisting of beads that slide along rods, has been used in Asia for thousands of years, possibly since 2000 BCE.

Throughout most of its history, computing has progressed relatively slowly. In 1623, a German scientist named Wilhelm Schickard invented the first known mechanical calculator, capable of performing simple arithmetical computations automatically. Although Schickard's device was lost to history through the ravages of the Thirty Years' War (1618–1648), the French philosopher Blaise Pascal used similar techniques to construct a mechanical adding machine in the 1640s, a copy of which remains on display in the Conservatoire des Arts et Métiers in Paris. In 1673, the German mathematician Gottfried Leibniz developed a considerably more sophisticated device, capable of multiplication and division as well as addition and subtraction. All these devices were purely mechanical and contained no engines or other source of power. The operator would enter numbers by setting metal wheels to a particular position; the act of turning those wheels set other parts of the machine in motion and changed the output display.

During the Industrial Revolution, the rapid growth in technology made it possible to consider new approaches to mechanical computation. The steam engine already provided the power needed to run factories and railroads. In that context, it was reasonable to ask whether one could use steam engines to drive more sophisticated computing machines, machines that would be capable of carrying out significant calculations under their own power. Before progress could be made, however, someone had to ask that question and set out to find an answer. The necessary spark of insight came from a British mathematician named Charles Babbage, who is one of the most interesting figures in the history of computing.

During his lifetime, Babbage designed two different computing machines, which he called the Difference Engine and the Analytical Engine; each represented a considerable advance over the calculating machines available at the time. The tragedy of his life is that he was unable to complete either of these projects. The Difference Engine, which he designed to produce tables of mathematical functions, was eventually built by a Swedish inventor in 1854—30 years after its original design. The Analytical Engine was Babbage's lifelong dream, but it remained incomplete when Babbage died in 1871. Even so, its design contained many of the essential features found in modern computers. Most importantly, Babbage conceived of the Analytical Engine as a general-purpose machine, capable of performing many different functions depending upon how it was *programmed*. In Babbage's design, the operation of the Analytical Engine was controlled by a pattern of holes punched on a card that the machine could read. By changing the pattern of holes, one could change the behavior of the machine so that it performed a different set of calculations.

Much of what we know of Babbage's work comes from the writings of Augusta Ada Byron, the only daughter of the poet Lord Byron and his wife Annabella. More than most of her contemporaries, Ada appreciated the potential of the Analytical Engine and

became its champion. She designed several sophisticated programs for the machine, thereby becoming the first programmer. In the 1970s, the U.S. Department of Defense named its own programming language Ada in honor of her contribution.

Some aspects of Babbage's design did influence the later history of computation, such as the use of punched cards to control computation—an idea that had first been introduced by the French inventor Joseph Marie Jacquard as part of a device to automate the process of weaving fabric on a loom. In 1890, Herman Hollerith used punched cards to automate data tabulation for the U.S. Census. To market this technology, Hollerith went on to found a company that later became the International Business Machines (IBM) corporation, which has dominated the computer industry for most of the twentieth century.

Babbage's vision of a programmable computer did not become a reality until the 1940s, when the advent of electronics made it possible to move beyond the mechanical devices that had dominated computing up to that time. A prototype of the first electronic computer was assembled in late 1939 by John Atanasoff and his student, Clifford Barry, at Iowa State College. They completed a full-scale implementation containing 300 vacuum tubes in May 1942. The computer was capable of solving small systems of linear equations. With some design modifications, the Atanasoff-Barry computer could have performed more intricate calculations, but work on the project was interrupted by World War II.

The first large-scale electronic computer was the ENIAC, an acronym for *Electronic Numerical Integrator And Computer*. Completed in 1946 under the direction of J. Presper Eckert and John Mauchly at the Moore School of the University of Pennsylvania, the ENIAC contained more than 18,000 vacuum tubes and occupied a 30-by-50 foot room. The ENIAC was programmed by plugging wires into a pegboard-like device called a **patch panel**. By connecting different sockets on the patch panel with wires, the operators could control ENIAC's behavior. This type of programming required an intimate knowledge of the internal workings of the machine and proved to be much more difficult than the inventors of the ENIAC had imagined.

Perhaps the greatest breakthrough in modern computing occurred in 1946, when John von Neumann at the Institute for Advanced Study in Princeton proposed that programs and data could be represented in a similar way and stored in the same internal memory. This concept, which simplifies the programming process enormously, is the basis of almost all modern computers. Because of this aspect of their design, modern computers are said to use **von Neumann architecture.**

Since the completion of ENIAC and the development of von Neumann's stored-programming concept, computing has evolved at a furious pace. New systems and new concepts have been introduced in such rapid succession that it would be pointless to list them all. Most historians divide the development of modern computers into the following four generations, based on the underlying technology.

- *First generation*. The first generation of electronic computers used vacuum tubes as the basis for their internal circuitry. This period of computing begins with the Atanasoff-Barry prototype in 1939.
- *Second generation*. The invention of the transistor in 1947 ushered in a new generation of computers. Transistors perform the same functions as vacuum tubes but are much smaller and require a fraction of the electrical power. The first computer to use transistors was the IBM 7090, introduced in 1958.

- *Third generation.* Even though transistors are tiny in comparison to vacuum tubes, a computer containing 100,000 or 1,000,000 individual transistors requires a large amount of space. The third generation of computing was enabled by the development in 1959 of the **integrated circuit** or **chip,** a small wafer of silicon that has been photographically imprinted to contain a large number of transistors connected together. The first computer to use integrated circuits in its construction was the IBM 360, which appeared in 1964.
- *Fourth generation.* The fourth generation of computing began in 1975, when the technology for building integrated circuits made it possible to put the entire processing unit of a computer on a single chip of silicon. The fabrication technology is called **large-scale integration.** Computer processors that consist of a single chip are called **microprocessors** and are used in most computers today.

The early machines of the first and second generations are historically important as the antecedents of modern computers, but they would hardly seem interesting or useful today. They were the dinosaurs of computer science: gigantic, lumbering beasts with small mental capacities, soon to become extinct. The late Robert Noyce, one of the inventors of the integrated circuit and founder of Intel Corporation, observed that, compared to the ENIAC, the typical modern computer chip "is twenty times faster, has a larger memory, is thousands of times more reliable, consumes the power of a light bulb rather than that of a locomotive, occupies 1/30,000 the volume, and costs 1/10,000 as much." Computers have certainly come of age.

1.2 What is computer science?

Growing up in the modern world has probably given you some idea of what a computer is. This text, however, is less concerned with computers as physical devices than with computer science. At first glance, the words *computer* and *science* seem an incongruous pair. In its classical usage, *science* refers to the study of natural phenomena; when people talk about *biological science* or *physical science*, we understand and feel comfortable with that usage. Computer science doesn't seem the same sort of thing. The fact that computers are human-made artifacts makes us reticent to classify the study of computers as a science. After all, modern technology has also produced cars, but we don't talk about "car science." Instead, we refer to "automotive engineering" or "automobile technology." Why should computers be any different?

To answer this question, it is important to recognize that the computer itself is only part of the story. The physical machine that you can buy today at your local computer store is an example of computer **hardware.** It is tangible. You can pick it up, take it home, and put it on your desk. If need be, you could use it as a doorstop, albeit a rather expensive one. But if there were nothing there besides the hardware, if a machine came to you exactly as it rolled off the assembly line, serving as a doorstop would be one of the few jobs it could do. A modern computer is a general-purpose machine, with the potential to perform a wide variety of tasks. To achieve that potential, however, the computer must be **programmed.** The act of programming a computer consists of providing it with a set of instructions—a program—that specifies all the steps necessary to solve the problem to which it is assigned. These programs are generically known as **software,** and it is the software, together with the hardware, that makes computation possible.

In contrast to hardware, software is an abstract, intangible entity. It is a sequence of simple steps and operations, stated in a precise language that the hardware can interpret. When we talk about computer science, we are concerned primarily with the domain of computer software and, more importantly, with the even more abstract domain of

problem solving. Problem solving turns out to be a highly challenging activity that requires creativity, skill, and discipline. For the most part, computer science is best thought of as the science of problem solving in which the solutions happen to involve a computer.

This is not to say that the computer itself is unimportant. Before computers, people could solve only relatively simple computational problems. Over the last 50 years, the existence of computers has made it possible to solve increasingly difficult and sophisticated problems in a timely and cost-effective way. As the problems we attempt to solve become more complex, so does the task of finding effective solution techniques. The science of problem solving has thus been forced to advance along with the technology of computing.

1.3 An overview of computer hardware

This text focuses almost exclusively on software and the activity of solving problems by computer that is the essence of computer science. Even so, it is important to spend some time in this chapter talking about the structure of computer hardware at a very general level of detail. The reason is simple: programming is a learn-by-doing discipline. You will not become a programmer just by reading this book, even if you solve all the exercises on paper. Learning to program is hands-on work and requires you to use a computer.

In order to use a computer, you need to become acquainted with its hardware. You have to know how to turn the computer on, how to use the keyboard to type in a program, and how to execute that program once you've written it. Unfortunately, the steps you must follow in order to perform these operations differ significantly from one computer system to another. As someone who is writing a general textbook, I cannot tell you how your own particular system works and must instead concentrate on general principles that are common to any computer you might be using. As you read this section, you should look at the computer you have and see how the general discussion applies to that machine.

Most computer systems today consist of the components shown in Figure 1-1. Each of the components in the diagram is connected by a communication channel called a **bus**,

FIGURE 1-1 Components of a typical computer
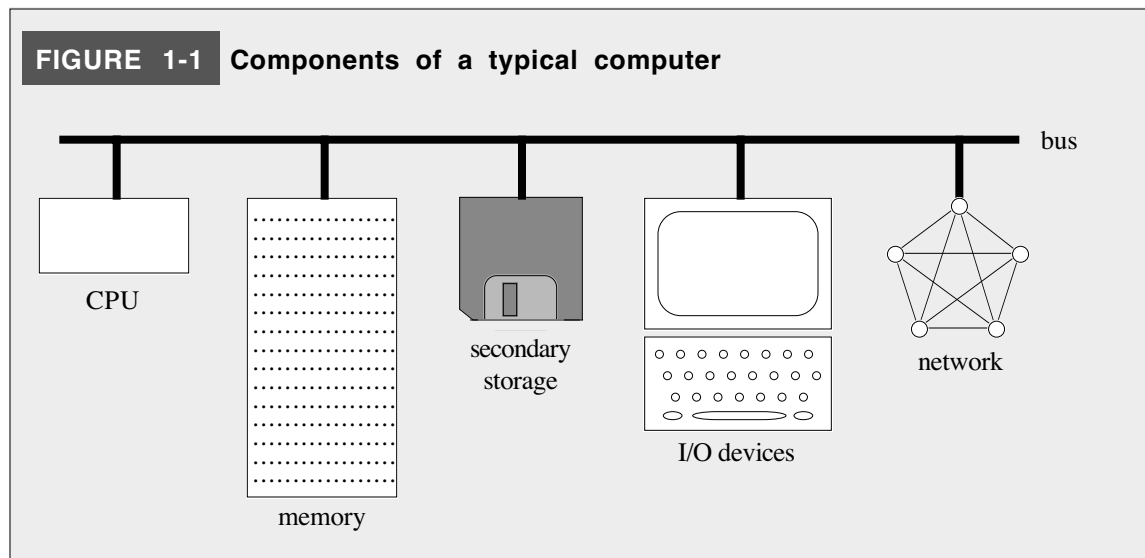
which allows data to flow between the separate units. The individual components are described in the sections that follow.

The CPU

The **central processing unit** or **CPU** is the "brain" of the computer. It performs the actual computation and controls the activity of the entire computer. The actions of the CPU are determined by a program consisting of a sequence of coded instructions stored in the memory system. One instruction, for example, might direct the computer to add a pair of numbers. Another might make a character appear on the computer screen. By executing the appropriate sequence of simple instructions, the computer can be made to perform complex tasks.

In a modern computer, the CPU consists of an **integrated circuit**—a tiny chip of silicon that has been imprinted with millions of microscopic transistors connected to form larger circuits capable of carrying out simple arithmetic and logical operations.

Memory

When a computer executes a program, it must have some way to store both the program itself and the data involved in the computation. In general, any piece of computer hardware capable of storing and retrieving information is a storage device. The storage devices that are used while a program is actively running constitute its **primary storage,** which is more often called its **memory.** Since John von Neumann first suggested the idea in 1946, computers have used the same memory to store both the individual instructions that compose the program and the data used during computation.

Memory systems are engineered to be very efficient so that they can provide the CPU with extremely fast access to their contents. In today's computers, memory is usually built out of a special integrated-circuit chip called a **RAM,** which stands for *random-access memory*. Random-access memory allows the program to use the contents of any memory cell at any time.

Secondary storage

Although computers usually keep active data in memory whenever a program is running, most primary storage devices have the disadvantage that they function only when the computer is turned on. When you turn off your computer, any information that was stored in primary memory is lost. To store permanent data, you need to use a storage device that does not require electrical power to maintain its information. Such devices constitute **secondary storage.**

The most common secondary storage devices used in computers today are **disks,** which consist of circular spinning platters coated with magnetic material used to record data. In a modern personal computer, disks come in two forms: **hard disks,** which are built into the computer system, and **floppy disks**, which are removable. When you compose and edit your program, you will usually do so on a hard disk, if one is available. When you want to move the program to another computer or make a backup copy for safekeeping, you will typically transfer the program to a floppy disk.

I/O devices

For the computer to be useful, it must have some way to communicate with users in the outside world. Computer input usually consists of characters typed on a keyboard. Output from the computer typically appears on the computer screen or on a printer. Collectively, hardware devices that perform input and output operations are called **I/O devices,** where I/O stands for *input/output*.

I/O devices vary significantly from machine to machine. Outside of the standard alphabetic keys, computer keyboards have different arrangements and even use different names for some of the important keys. For example, the key used to indicate the end of a line is labeled Return on some keyboards and Enter on others. On some computer systems, you make changes to a program by using special **function keys** on the top or side of the keyboard that provide simple editing operations. On other systems, you can accomplish the same task by using a hand-held pointing device called a **mouse** to select program text that you wish to change. In either case, the computer keeps track of the current typing position, which is usually indicated on the screen by a flashing line or rectangle called the **cursor**.

Network

The final component shown in Figure 1-1 is the **network,** which indicates a connection to the constellation of other computers that are connected together as part of the Internet. In many respects, the network is much the same as the I/O devices in terms of the overall hardware structure. As the network becomes increasingly central to our collective expectation of what computing means, it makes sense to include the network as a separate component to emphasize its importance. Adding emphasis to the role of networking is particularly important in a book that uses Java as its programming language because the success of Java was linked fairly closely to the rise of networking, as discussed later in this chapter.

1.4 Algorithms

Now that you have a sense of the structure of a computer system, let's turn to computer science. Because computer science is the discipline of solving problems with the assistance of a computer, you need to understand a concept that is fundamental to both computer science and the abstract discipline of problem solving—the concept of an **algorithm.** The word *algorithm* comes to us from the name of the ninth-century Persian mathematician Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî, who wrote a treatise on mathematics entitled *Kitab al jabr w'al-muqabala* (which itself gave rise to the English word *algebra*). Informally, you can think of an algorithm as a strategy for solving a problem. To appreciate how computer scientists use the term, however, it is necessary to formalize that intuitive understanding and tighten up the definition.

To be an algorithm, a solution technique must fulfill three basic requirements. First of all, an algorithm must be presented in a clear, unambiguous form so that it is possible to understand what steps are involved. Second, the steps within an algorithm must be effective, in the sense that it is possible to carry them out in practice. A technique, for example, that includes the operation "multiply r by the exact value of π" is not effective, since it is not possible to compute the exact value of π. Third, an algorithm must not run on forever but must deliver its answer in a finite amount of time. In summary, an algorithm must be

1. *Clearly and unambiguously defined*.
2. *Effective,* in the sense that its steps are executable.
3. *Finite,* in the sense that it terminates after a bounded number of steps.

These properties will turn out to be more important later on when you begin to work with complex algorithms. For the moment, it is sufficient to think of algorithms as abstract solution strategies—strategies that will eventually become the core of the programs you write.

As you will soon discover, algorithms—like the problems they are intended to solve—vary significantly in complexity. Some problems are so simple that an appropriate algorithm springs immediately to mind, and you can write the programs to solve such problems without too much trouble. As the problems become more complex, however, the algorithms needed to solve them begin to require more thought. In most cases, several different algorithms are available to solve a particular problem, and you need to consider a variety of potential solution techniques before writing the final program.

1.5 Stages in the programming process

Solving a problem by computer consists of two conceptually distinct steps. First, you need to develop an algorithm, or choose an existing one, that solves the problem. This part of the process is called **algorithmic design.** The second step is to express that algorithm as a computer program in a programming language. This process is called **coding.**

As you begin to learn about programming, the process of coding—translating your algorithm into a functioning program—will seem to be the more difficult phase of the process. As a new programmer, you will, after all, be starting with simple problems just as you would when learning any new skill. Simple problems tend to have simple solutions, and the algorithmic design phase will not seem particularly challenging. Because the language and its rules are entirely new and unfamiliar, however, coding may at times seem difficult and arbitrary. I hope it is reassuring to say that coding will rapidly become easier as you learn more about the programming process. At the same time, however, algorithmic design will get harder as the problems you are asked to solve increase in complexity.

When new algorithms are introduced in this text, they will usually be expressed initially in English. Although it is often less precise than one would like, English is a reasonable language in which to express solution strategies as long as the communication is entirely between people who speak English. Obviously, if you wanted to present your algorithm to someone who spoke only Russian, English would no longer be an appropriate choice. English is likewise an inappropriate choice for presenting an algorithm to a computer. Although computer scientists have been working on this problem for decades, understanding English or Russian or any other human language continues to lie beyond the boundaries of current technology. The computer would be completely unable to interpret your algorithm if it were expressed in human language. To make an algorithm accessible to the computer, you need to translate it into a programming language. There are many programming languages in the world, including Fortran, BASIC, Pascal, Lisp, C, C++, and a host of others. In this text, you will learn how to use the programming language Java—a language developed by Sun Microsystems in 1995 that has since become something of a standard both for industry and for introductory computer science courses.

Creating and editing programs

Before you can run a program on most computer systems, it is necessary to enter the text of the program and store it in a **file,** which is the generic name for any collection of information stored in the computer's secondary storage. Every file must have a name, which is usually divided into two parts separated by a period, as in `MyProgram.java`. When you create a file, you choose the **root name,** which is the part of the name preceding the period, and use it to tell yourself what the file contains. The portion of the filename following the period indicates what the file is used for and is called the **extension.** Certain extensions have preassigned meanings. For example, the extension

`.java` indicates a program file written in the Java language. A file containing program text is called a **source file.**

The general process of entering or changing the contents of a file is called **editing** that file. The editing process differs significantly between individual computer systems, so it is not possible to describe it in a way that works for every type of hardware. When you work on a particular computer system, you will need to learn how to create new files and to edit existing ones. You can find this information in the computer manual or the documentation for the compiler you are using.

The compilation process

Once you have created your source file, the next step in the process is to translate your program into a form that the computer can understand. Languages like Java, C, and C++ are examples of what computer scientists call **higher-level languages.** Such languages are designed to make it easier for human programmers to express algorithms without having to understand in detail exactly how the underlying hardware will execute those algorithms. Higher-level languages are also typically independent of the particular characteristics that differentiate individual machine architectures. Internally, however, each computer system understands a low-level language that is specific to that type of hardware, which is called its **machine language.** For example, the Apple Macintosh and a Windows-based computer use different underlying machine languages, even though both of them can execute programs written in a higher-level language.

To make it possible for a program written in a higher-level language to run on different computer systems, there are two basic strategies. The classical approach is to use a program called a **compiler** to translate the programs that you write into the low-level machine language appropriate to the computer on which the program will run. Under this strategy, different platforms require different translators. For example, if you are writing C programs for a Macintosh, you need to run a special program that translates C into the machine language for the Macintosh. If you are using a Windows platform to run the same program, you need to use a different translator because the underlying hardware uses a different machine language.

The second approach is to translate the program into an **intermediate language** that is independent of the underlying platform. On each of these platforms, programs run in a system called an **interpreter** that executes the intermediate language for that machine. In a pure interpreter, the interpreter does not actually translate the intermediate language into machine language but simply implements the intended effect for each operation.

Modern implementations of Java use a hybrid approach. A Java compiler translates your programs into a common intermediate language. That language is then interpreted by a program called the **Java Virtual Machine** (or JVM for short) that executes the intermediate language for that machine. The program that runs the Java Virtual Machine, however, typically does compile pieces of the intermediate code into the underlying machine language. As a result, Java can often achieve a level of efficiency that is unattainable with traditional interpreters.

In classical compiler-based systems, the compiler translates the source file into a second file called an **object file** that contains the actual instructions appropriate for that computer system. This object file is then combined together with other object files to produce an **executable file** that can be run on the system. These other object files typically include predefined object files, called **libraries,** that contain the machine-language instructions for various operations commonly required by programs. The

FIGURE 1-2 **Stages in the classical compilation process**
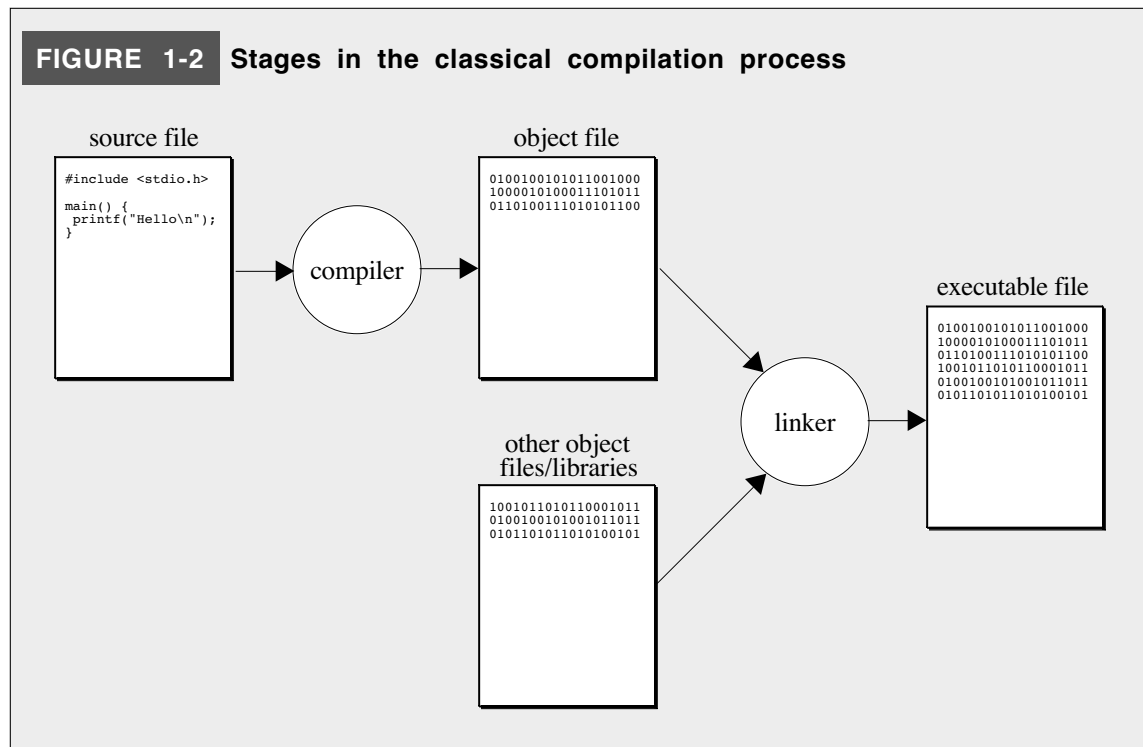
process of combining all the individual object files into an executable file is called **linking.** The entire process is illustrated by the diagram shown in Figure 1-2.

In Java, the process is slightly more elaborate. As noted earlier in this section, Java produces intermediate code that it stores in files called **class files.** Those class files are then combined with other class files and libraries to produce a complete version of the intermediate program with everything it needs linked together. The usual format for that version of the program is a compressed collection of individual files called a **JAR archive.** That archive file is then interpreted by the Java Virtual Machine in such a way that the output appears on your computer. This process is illustrated in Figure 1-3.

Programming errors and debugging

Besides translation, compilers perform another important function. Like human languages, programming languages have their own vocabulary and their own set of grammatical rules. These rules make it possible to determine that certain statements are properly constructed and that others are not. For example, in English, it is not appropriate to say "we goes" because the subject and verb do not agree in number. Rules that determine whether a statement is legally constructed are called **syntax rules.** Programming languages have their own syntax, which determines how the elements of a program can be put together. When you compile a program, the compiler first checks to see whether your program is syntactically correct. If you have violated the syntactic rules, the compiler displays an error message. Errors that result from breaking these rules are called **syntax errors.** Whenever you get a message from the compiler indicating a syntax error, you must go back and edit the program to correct it.

Syntax errors can be frustrating, particularly for new programmers. They will not, however, be your biggest source of frustration. More often than not, the programs you write will fail to operate correctly not because you wrote a program that contained syntactic errors but because your perfectly legal program somehow comes up with

FIGURE 1-3 **Stages in running a Java program**
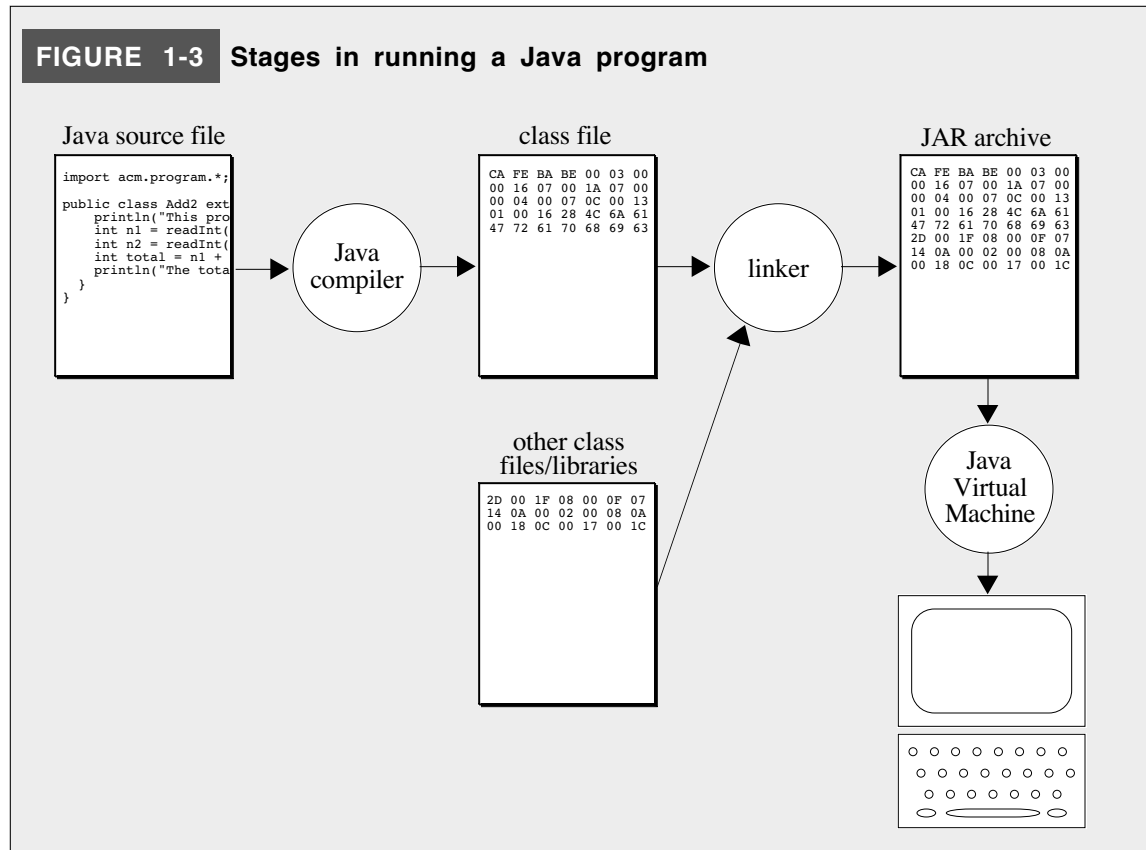
incorrect answers or fails to produce answers at all. You look at the program and discover that you have made a mistake in the logic of the program—the type of mistake programmers call a **bug.** The process of finding and correcting such mistakes is called **debugging** and is an important part of the programming process.

Bugs can be extremely insidious and frustrating. You will be absolutely certain that your algorithm is correct, and then discover that it fails to handle some case you had previously overlooked. Or perhaps you will think about a special condition at one point in your program only to forget it later on. Or you might make a mistake that seems so silly you cannot believe anyone could possibly have blundered so badly.

Relax. You're in excellent company. Even the best programmers have shared this experience. The truth is that programmers—all programmers—make logic errors. In particular, *you* will make logic errors. Algorithms are tricky things, and you will often discover that you haven't really gotten it right.

In many respects, discovering your own fallibility is an important rite of passage for you as a programmer. Describing his experiences as a programmer in the early 1960s, the pioneering computer scientist Maurice Wilkes wrote:

> Somehow, at the Moore School and afterwards, one had always assumed there
> would be no particular difficulty in getting programs right. I can remember the
> exact instant in time at which it dawned on me that a great part of my future life
> would be spent in finding mistakes in my own programs.

What differentiates good programmers from the rest of their colleagues is not that they manage to avoid bugs altogether but that they take pains to minimize the number of bugs

that persist in the finished code. When you design an algorithm and translate it into a syntactically legal program, it is critical to understand that your job is not finished. Almost certainly, your program has a bug in it somewhere. Your job as a programmer is to find that bug and fix it. Once that is done, you should find the next bug and fix that. Always be skeptical of your own programs and test them as thoroughly as you can.

Software maintenance

One of the more surprising aspects of software development is that programs require maintenance. In fact, studies of software development indicate that, for most programs, paying programmers to maintain the software after it has been released constitutes between 80 and 90 percent of the total cost. In the context of software, however, it is a little hard to imagine precisely what maintenance means. At first hearing, the idea sounds rather bizarre. If you think in terms of a car or a bridge, maintenance occurs when something has broken—some of the metal has rusted away, a piece of some mechanical linkage has worn out from overuse, or something has gotten smashed up in an accident. None of these situations apply to software. The code itself doesn't rust. Using the same program over and over again does not in any way diminish its functioning. Accidental misuse can certainly have dangerous consequences but does not usually damage the program itself; even if it does, the program can often be restored from a backup copy. What does maintenance mean in such an environment?

Software requires maintenance for two principal reasons. First, even after considerable testing and, in some cases, years of field use, bugs can still survive in the original code. Then, when some unusual situation arises or a previously unanticipated load occurs, the bug, previously dormant, causes the program to fail. Thus, debugging is an essential part of program maintenance. It is not, however, the most important part. Far more consequential, especially in terms of how much it contributes to the overall cost of program maintenance, is what might be called **feature enhancement.** Programs are written to be used; they perform, usually faster and less expensively than other methods, a task that the customer needs done. At the same time, the programs probably don't do everything the customer wants. After working with a program for a while, the customer decides it would be wonderful if the program also did something else, or did something differently, or presented its data in a more useful way, or ran a little faster, or had an expanded capacity, or just had a few more simple but attractive features (often called **bells and whistles** in the trade). Since software is extremely flexible, suppliers have the option of responding to such requests. In either case—whether one wants to repair a bug or add a feature—someone has to go in, look at the program, figure out what's going on, make the necessary changes, verify that those changes work, and then release a new version. This process is difficult, time-consuming, expensive, and prone to error.

Part of the reason program maintenance is so difficult is that most programmers do not write their programs for the long haul. To them it seems sufficient to get the program working and then move on to something else. The discipline of writing programs so that they can be understood and maintained by others is called **software engineering**. In this text, you are encouraged to write programs that demonstrate good engineering style.

As you write your programs, try to imagine how someone else might feel if called upon to look at them two years later. Would your program make sense? Would the program itself indicate to the new reader what you were trying to do? Would it be easy to change, particularly along some dimension where you could reasonably expect change? Or would it seem obscure and convoluted? If you put yourself in the place of the future maintainer (and as a new programmer in most companies, you will probably be given that role), it will help you to appreciate why good style is critical.

Many novice programmers are disturbed to learn that there is no precise set of rules you can follow to ensure good programming style. Good software engineering is not a cookbook sort of process. Instead it is a skill blended with more than a little bit of artistry. Practice is critical. One learns to write good programs by writing them, and by reading others, much as one learns to be a novelist. Good programming requires discipline—the discipline not to cut corners or to forget about that future maintainer in the rush to complete a project. And good programming style requires developing an aesthetic sense—a sense of what it means for a program to be readable and well presented.

1.6 Java and the object-oriented paradigm

As noted earlier in this chapter, this text uses the programming language Java to illustrate the more general concepts of programming and computer science. But why Java? The answer lies primarily in the way that Java encourages programmers to think about the programming process.

Over the last decade, computer science and programming have gone through something of a revolution. Like most revolutions—whether political upheavals or the conceptual restructurings that Thomas Kuhn describes in his 1962 book *The Structure of Scientific Revolutions*—this change has been driven by the emergence of an idea that challenges an existing orthodoxy. Initially, the two ideas compete. For a while, the old order maintains its dominance. Over time, however, the strength and popularity of the new idea grows, until it begins to displace the older idea in what Kuhn calls a **paradigm shift.** In programming, the old order is represented by the **procedural paradigm,** in which programs consist of a collection of procedures and functions that operate on data. The challenger is the **object-oriented paradigm,** in which programs are viewed instead as a collection of "objects" for which the data and the operations acting on that data are encapsulated into integrated units. Most traditional languages, including Fortran, Pascal, and C, embody the procedural paradigm. The best-known representatives of the object-oriented paradigm are Smalltalk, C++, and Java.

Although object-oriented languages are gaining popularity at the expense of procedural ones, it would be a mistake to regard the object-oriented and procedural paradigms as mutually exclusive. Programming paradigms are not so much competitive as they are complementary. The object-oriented and the procedural paradigm—along with other important paradigms such as the functional programming style embodied in LISP and Scheme—all have important applications in practice. Even within the context of a single application, you are likely to find a use for more than one approach. As a programmer, you must master many different paradigms, so that you can use the conceptual model that is most appropriate to the task at hand.

The history of object-oriented programming

The idea of object-oriented programming is not really all that new. The first object-oriented language was SIMULA, a language for coding simulations designed in the early 1960s by the Scandinavian computer scientists Ole-Johan Dahl, Björn Myhrhaug, and Kristen Nygaard. With a design that was far ahead of its time, SIMULA anticipated many of the concepts that later became commonplace in programming, including the concept of abstract data types and much of the modern object-oriented paradigm. In fact, most of the terminology used to describe object-oriented systems comes from the original reports on the initial version of SIMULA and its successor, SIMULA 67.

For many years, however, SIMULA mostly just sat on the shelf. Few people paid much attention to it, and the only place you were likely to hear about it would be in a course on programming language design. The first object-oriented language to gain any significant level of recognition within the computing profession was Smalltalk, which was developed at the Xerox Palo Alto Research Center (more commonly known as Xerox PARC) in the late 1970s. The purpose of Smalltalk, which is described in the book *Smalltalk-80: The Language and Its Implementation* by Adele Goldberg and David Robson, was to make programming accessible to a wider audience. As such, Smalltalk was part of a larger effort at Xerox PARC that gave rise to much of the modern user-interface technology that is now standard on personal computers.

Despite many attractive features and a highly interactive user environment that simplifies the programming process, Smalltalk never achieved much commercial success. The profession as a whole took an interest in object-oriented programming only when the central ideas were incorporated into variants of C, which had become an industry standard. Although there were several parallel efforts to design an object-oriented language based on C, the most successful was the language C++, which was designed in the early 1980s by Bjarne Stroustrup at AT&T Bell Laboratories. By making it possible to integrate object-oriented techniques with existing C code, C++ enabled large communities of programmers to adopt the object-oriented paradigm in a gradual, evolutionary way.

The Java programming language

The most recent chapter in the history of object-oriented programming is the development of Java by a team of programmers at Sun Microsystems led by James Gosling. In 1991, when Sun initiated the project that would eventually become Java, the goal was to design a language suitable for programming microprocessors embedded in consumer electronic devices. Had this goal remained the focus of the project, it is unlikely that Java would have caught on to the extent that it has. As is often the case in computing, the direction of Java changed during its development phase in response to changing conditions in the industry. The key factor leading to the change in focus was the phenomenal growth in the Internet that occurred in the early 1990s, particularly in the form of the **World Wide Web,** an ever-expanding collection of interconnected resources contributed by computer users all over the world. When interest in the Web skyrocketed in 1993, Sun redesigned Java as a tool for writing highly interactive, Web-based applications. That decision proved extremely fortuitous. Since the formal announcement of the language in May 1995, Java has generated unprecedented excitement in both the academic and commercial computing communities. In the process, object-oriented programming has become firmly established as a central paradigm in the computing industry.

To get a sense of the strengths of Java, it is useful to look at Figure 1-4, which contains excerpts from a now-classic paper on the initial Java design written in 1996 by James Gosling and Henry McGilton. In that paper, the authors describe Java with a long series of adjectives: simple, object-oriented, familiar, robust, secure, architecture-neutral, portable, high-performance, interpreted, threaded, and dynamic. The discussion in Figure 1-4 will provide you with a sense as to what these buzzwords mean, and you will come to appreciate the importance of these features even more as you learn more about Java and computer science.

FIGURE 1-4   **Excerpts from the "Java White Paper"**

DESIGN GOALS OF THE JAVA™ PROGRAMMING LANGUAGE

The design requirements of the Java™ programming language are driven by the nature of the computing environments in which software must be deployed.

The massive growth of the Internet and the World-Wide Web leads us to a completely new way of looking at development and distribution of software. To live in the world of electronic commerce and distribution, Java technology must enable the development of secure, high performance, and highly robust applications on multiple platforms in heterogeneous, distributed networks.

Operating on multiple platforms in heterogeneous networks invalidates the traditional schemes of binary distribution, release, upgrade, patch, and so on. To survive in this jungle, the Java programming language must be architecture neutral, portable, and dynamically adaptable.

The system that emerged to meet these needs is simple, so it can be easily programmed by most developers; familiar, so that current developers can easily learn the Java programming language; object oriented, to take advantage of modern software development methodologies and to fit into distributed client-server applications; multithreaded, for high performance in applications that need to perform multiple concurrent activities, such as multimedia; and interpreted, for maximum portability and dynamic capabilities.

Together, the above requirements comprise quite a collection of buzzwords, so let's examine some of them and their respective benefits before going on.

Simple, Object Oriented, and Familiar

Primary characteristics of the Java programming language include a simple language that can be programmed without extensive programmer training while being attuned to current software practices. The fundamental concepts of Java technology are grasped quickly; programmers can be productive from the very beginning.

The Java programming language is designed to be object oriented from the ground up. Object technology has finally found its way into the programming mainstream after a gestation period of thirty years. The needs of distributed, client-server based systems coincide with the encapsulated, message-passing paradigms of object-based software. To function within increasingly complex, network-based environments, programming systems must adopt object-oriented concepts. Java technology provides a clean and efficient object-based development platform.

Programmers using the Java programming language can access existing libraries of tested objects that provide functionality ranging from basic data types through I/O and network interfaces to graphical user interface toolkits. These libraries can be extended to provide new behavior.

Even though C++ was rejected as an implementation language, keeping the Java programming language looking like C++ as far as possible results in it being a familiar language, while removing the unnecessary complexities of C++. Having the Java programming language retain many of the object-oriented features and the "look and feel" of C++ means that programmers can migrate easily to the Java platform and be productive quickly.

Robust and Secure

The Java programming language is designed for creating highly reliable software. It provides extensive compile-time checking, followed by a second level of run-time checking. Language features guide programmers towards reliable programming habits.

The memory management model is extremely simple: objects are created with a new operator. There are no explicit programmer-defined pointer data types, no pointer arithmetic, and automatic garbage collection. This simple memory management model eliminates entire classes of programming errors that bedevil C and C++ programmers. You can develop Java code with confidence that the system will find many errors quickly and that major problems won't lay dormant until after your production code has shipped.

Java technology is designed to operate in distributed environments, which means that security is of paramount importance. With security features designed into the language and run-time system, Java technology lets you construct applications that can't be invaded from outside. In the network environment, applications written in the Java programming language are secure from intrusion by unauthorized code attempting to get behind the scenes and create viruses or invade file systems.

Architecture Neutral and Portable

Java technology is designed to support applications that will be deployed into heterogeneous network environments. In such environments, applications must be capable of executing on a variety of hardware architectures. Within this variety of hardware platforms, applications must execute atop a variety of operating systems and interoperate with multiple programming language interfaces. To accommodate the diversity of operating environments, the Java Compiler™ product generates bytecodes—an architecture neutral intermediate format designed to transport code efficiently to multiple hardware and software platforms. The interpreted nature of Java technology solves both the binary distribution problem and the version problem; the same Java programming language byte codes will run on any platform.

Architecture neutrality is just one part of a truly portable system. Java technology takes portability a stage further by being strict in its definition of the basic language. Java technology puts a stake in the ground and specifies the sizes of its basic data types and the behavior of its arithmetic operators. Your programs are the same on every platform—there are no data type incompatibilities across hardware and software architectures.

The architecture-neutral and portable language platform of Java technology is known as the Java virtual machine. It's the specification of an abstract machine for which Java programming language compilers can generate code. Specific implementations of the Java virtual machine for specific hardware and software platforms then provide the concrete realization of the virtual machine. The Java virtual machine is based primarily on the POSIX interface specification—an industry-standard definition of a portable system interface. Implementing the Java virtual machine on new architectures is a relatively straightforward task as long as the target platform meets basic requirements such as support for multithreading.

High Performance

Performance is always a consideration. The Java platform achieves superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the run-time environment. The automatic garbage collector runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance. Applications requiring large amounts of compute power can be designed such that compute-intensive sections can be rewritten in native machine code as required and interfaced with the Java platform. In general, users perceive that interactive applications respond quickly even though they're interpreted.

Interpreted, Threaded, and Dynamic

The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter and run-time system have been ported. In an interpreted platform such as Java technology-based system, the link phase of a program is simple, incremental, and lightweight. You benefit from much faster development cycles—prototyping, experimentation, and rapid development are the normal case, versus the traditional heavyweight compile, link, and test cycles.

Modern network-based applications, such as the HotJava™ Browser for the World Wide Web, typically need to do several things at the same time. A user working with HotJava Browser can run several animations concurrently while downloading an image and scrolling the page. Java technology's multithreading capability provides the means to build applications with many concurrent threads of activity. Multithreading thus results in a high degree of interactivity for the end user.

The Java platform supports multithreading at the language level with the addition of sophisticated synchronization primitives: the language library provides the Thread class, and the run-time system provides monitor and condition lock primitives. At the library level, moreover, Java technology's high-level system libraries have been written to be thread safe: the functionality provided by the libraries is available without conflict to multiple concurrent threads of execution.

While the Java Compiler is strict in its compile-time static checking, the language and run-time system are dynamic in their linking stages. Classes are linked only as needed. New code modules can be linked in on demand from a variety of sources, even from sources across a network. In the case of the HotJava Browser and similar applications, interactive executable code can be loaded from anywhere, which enables transparent updating of applications. The result is on-line services that constantly evolve; they can remain innovative and fresh, draw more customers, and spur the growth of electronic commerce on the Internet.

—*White Paper: The Java Language Environment*
James Gosling and Henry McGilton, May 1996

1.7 Java and the World Wide Web

In many ways, Java's initial success as a language was tied to the excitement surrounding computer networks in the early 1990s. Computer networks had at that time been around for more than 20 years, ever since the first four nodes in the ARPANET—the forerunner of today's Internet—came on line in 1969. What drove the enormous boom in Internet technology throughout the 1990s was not so much the network itself as it was the invention of the World Wide Web, which allows users to move from one document to another by clicking on interactive links.

Documents that contain interactive links are called **hypertext**—a term coined in 1965 by Ted Nelson, who proposed the creation of an integrated collection of documents that has much in common with today's World Wide Web. The fundamental concepts, however, are even older; the first Presidential Science Advisor, Vannevar Bush, proposed a similar idea in 1945. This idea of a distributed hypertext system, however, was not successfully put into practice until 1989, when Tim Berners-Lee of CERN, the European Particle Physics Laboratory in Geneva, proposed creating a repository that he called the World Wide Web. In 1991, implementers at CERN completed the first **browser,** a program that displays Web documents in a way that makes it easy for users to follow the internal links to other parts of the Web. After news of the CERN work spread to other researchers in the physics community, more groups began to create browsers. Of these, the most successful was the Mosaic project based at the National Center for Supercomputing Applications (NCSA) in Champaign, Illinois. After the appearance of the Mosaic browser in 1993, interest in the Web exploded. The number of computer systems implementing World Wide Web repositories grew from approximately 500 in 1993 to over 35,000,000 in 2003. The enthusiasm for the Web in the Internet community has also sparked considerable commercial interest, leading to the formation of several new companies and the release of commercial Web browsers like Apple's Safari, Netscape's Navigator and Microsoft's Internet Explorer.

The number of documents available on the World Wide Web has grown rapidly because Internet users can easily create new documents and add them to the Web. If you want to add a new document to the Web, all you have to do is create a file on a system equipped with a program called a **Web server** that gives external users access to the files on that system. The individual files exported by the server are called **Web pages.** Web pages are usually written in a language called **HTML**, which is short for *Hypertext Markup Language*. HTML documents consist of text along with formatting information and links to other pages elsewhere on the Web. Each page is identified by a **uniform resource locator,** or **URL,** which makes it possible for Web browsers to find this page in the sea of existing pages. URLs for the World Wide Web begin with the prefix `http://`, which is followed by a description of the Internet path needed to reach the desired page.

One of the particularly interesting aspects of Java is that the virtual machine is not always running on the same machine that houses the programs. One of Java's design goals was to make the language work well over a network. A particularly interesting consequence of this design goal is that Java supports the creation of **applets,** which are programs that run in the context of a network browser. The process of running an applet is even more intricate than the models of program execution presented earlier in the chapter and is described in Figure 1-5.

FIGURE 1-5 **Java programs running as applets**

Steps taken by the applet author   *Steps taken by the applet user*

1. The author of the Web page writes the code for a program to run as an applet.

   GraphicHello.java

   ```
   /* File: GraphicHello.java */

   import acm.graphics.*;
   import acm.program.*;

   public class GraphicHello extends GraphicsProgram {
      public void run() {
         add(new GLabel("Hello, world!"), 20, 20);
      }
   }
   ```

2. The applet author then uses a Java compiler to generate a file containing a byte-coded version of the applet.

   GraphicHello.jar

   ```
   CA FE BA BE 00 03 00 2D 00 1F 08 00 0F 07 C8 00
   00 16 07 00 1A 07 00 14 0A 00 02 00 08 0A 00 5F
   00 04 00 07 0C 00 13 00 18 0C 00 17 00 1C 72 A4
   01 00 16 28 4C 6A 61 76 61 2F 61 77 74 2F 00 FF
   47 72 61 70 68 69 63 73 3B 29 56 01 00 04 9E 00
   ```

3. The applet author publishes an HTML Web page that includes a reference to the compiled applet.

   GraphicHello.html

   ```
   <html>
   <title>Graphic Hello Applet</title>
   <applet archive="GraphicHello.jar"
           code="GraphicHello.class"
           width=300 height=150>
   </applet>
   </html>
   ```

4. The user's browser reads the HTML source for the Web page and begins to display the image on the screen.

5. The appearance of an **applet** tag in the HTML source file causes the browser to download the compiled applet over the network.

6. A verifier program in the browser checks the byte codes in the applet to ensure that they do not violate the security of the user's system.

7. The Java interpreter in the browser program runs the compiled applet, which generates the desired display on the user's console.

   Graphic Hello Program

   Hello, world!
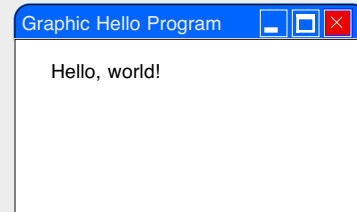
Summary

The purpose of this chapter is to set the stage for learning about computer science and programming, a process that you will begin in earnest in Chapter 2. In this chapter, you have focused on what the programming process involves and how it relates to the larger domain of computer science.

The important points introduced in this chapter include:

• The physical components of a computer system—the parts you can see and touch— constitute *hardware*. Before computer hardware is useful, however, you must specify a sequence of instructions, or *program,* that tells the hardware what to do. Such programs are called *software*.

• Computer science is not so much the science of computers as it is the science of solving problems using computers.

• Strategies for solving problems on a computer are known as *algorithms*. To be an algorithm, the strategy must be clearly and unambiguously defined, effective, and finite.

- Programs are typically written using a *higher-level language* that is then translated by a *compiler* into the *machine language* of a specific computer system or into an *intermediate language* executed by an *interpreter*.

- To run a program, you must first create a *source file* containing the text of the program. The compiler translates the source file into an *object file,* which is then linked with other object files to create the executable program.

- Programming languages have a set of *syntax rules* that determine whether a program is properly constructed. The compiler checks your program against these syntax rules and reports a *syntax error* whenever the rules are violated.

- The most serious type of programming error is one that is syntactically correct but that nonetheless causes the program to produce incorrect results or no results at all. This type of error, in which your program does not correctly solve a problem because of a mistake in your logic, is called a *bug*. The process of finding and fixing bugs is called *debugging*.

- Most programs must be updated periodically to correct bugs or to respond to changes in the demands of the application. This process is called *software maintenance*. Designing a program so that it is easier to maintain is an essential part of *software engineering*.

- This text uses the programming language Java to illustrate the programming process. The primary feature that sets Java apart from most of its predecessor languages is the fact that it is an *object-oriented language,* which means that it encapsulates data and the operations on that data into conceptually unified entities called *objects*.

- Java was designed during the "Internet boom" of the 1990s and is designed to work well in a networked environment. In particular, Java makes it possible to run programs in the context of a web browser. Programs that run in this way are called *applets*.

Review questions

1. What new concept in computing was introduced in the design of Babbage's Analytical Engine?

2. Who is generally regarded as the first programmer?

3. What concept lies at the heart of von Neumann architecture?

4. What is the difference between hardware and software?

5. Traditional science is concerned with abstract theories or the nature of the universe — not human-made artifacts. What abstract concept forms the core of computer science?

6. What are the three criteria an algorithm must satisfy?

7. What is the distinction between algorithmic design and coding? Which of these activities is usually harder?

8. What is meant by the term *higher-level language?* What higher-level language is used as the basis of this text?

9. How does an interpreter differ from a compiler?

10. What is the relationship between a source file and an object file? As a programmer, which of these files do you work with directly?

11. What is the difference between a syntax error and a bug?

12. True or false: Good programmers never introduce bugs into their programs.

13. True or false: The major expense of writing a program comes from the development of that program; once the program is put into practice, programming costs are negligible.

14. What is meant by the term *software maintenance?*

15. Why is it important to apply good software engineering principles when you write your programs?

16. What is the fundamental difference between the object-oriented and procedural paradigms?

17. What steps are involved in running an applet under the control of a web browser? In what ways does running a Java applet differ from running a Java application?

Chapter 2
Programming by Example

Example is always more efficacious than precept.
— Samuel Johnson, *Rasselas,* 1759

Grace Murray Hopper (1906–1992)

Grace Murray Hopper studied mathematics and physics at Vassar College and went on to earn her Ph.D. in mathematics at Yale. During the Second World War, Hopper joined the United States Navy and was posted to the Bureau of Ordinance Computation at Harvard University, where she worked with computing pioneer Howard Aiken. Hopper became one of the first programmers of the Mark I digital computer, which is the machine visible in the background of this photograph. Hopper made several contributions to computing in its early years and was one of the major contributors to the development of COBOL, which continues to have widespread use in business-programming applications. In 1985, Hopper became the first woman promoted to the rank of admiral. During her life, Grace Murray Hopper served as the most visible example of a successful woman in computer science. In recognition of that contribution, there is now a biennial Celebration of Women in Computing, which was named in her honor.

The purpose of this book is to teach you the fundamentals of programming. Along the way, you will become quite familiar with a particular programming language called Java, but the details of that language are not the main point. Programming is the science of solving problems by computer, and most of what you learn from this text will be independent of the specific details of Java. Even so, you will have to master many of those details eventually so that your programs can take maximum advantage of the tools that Java provides.

From your position as a new student of programming, the need to understand both the abstract concepts of programming and the concrete details of a specific programming language leads to a dilemma: there is no obvious place to start. To learn about programming, you need to write some fairly complex programs. To write those programs in Java, you must know enough about the language to use the appropriate tools. But if you spend all of your energy learning about Java, you will probably not learn as much as you should about more general programming issues. Moreover, Java was designed for experts and not for beginning programmers. There are many details that just get in the way if you try to master Java without first understanding something about programming.

Because it's important for you to get a feel for what programming is before you master its intricacies, this chapter begins by presenting a few simple programs in their entirety. When you look at these programs, try to understand what is happening in them generally without being concerned about details just yet. You can learn about those details in Chapters 3 and 4. The main purpose of this chapter and the one that follows is to help build your intuition about programming and problem solving, which is far more important in the long run.

2.1 The "hello world" program

Java is part of a collection of languages that grew out of C, one of the most successful programming languages in the history of the field. In the book that has served as C's defining document, *The C Programming Language* by Brian Kernighan and Dennis Ritchie, the authors offer the following advice on the first page of Chapter 1:

> The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:
>
> *Print the words*
>   **hello, world**
>
> This is the big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where the output went. With these mechanical details mastered, everything else is comparatively easy.

That advice was followed by the four-line text of the "hello world" program, which became part of the heritage shared by all C programmers. Java is, of course, different from C, but the underlying advice is still sound: The first program you write should be as simple as possible to ensure that you can master the mechanics of the programming process.

At the same time, it is important to remember that this is now the 21st century, and the programs that were appropriate to the early 1970s are not the same ones we would use today. The mechanical teletypes and primitive consoles that were available then have been replaced by more sophisticated displays, and the ability to print a series of words is no longer quite as exciting as it once was. Today, that output would more likely be directed to a graphical window on the screen. Fortunately, the Java program that does precisely that is still very simple and appears in Figure 2-1.

FIGURE 2-1 The "hello world" program

```
/*
 * File: HelloProgram.java
 * ----------------------
 * This program displays "hello, world" on the screen.
 * It is inspired by the first program in Brian
 * Kernighan and Dennis Ritchie's classic book,
 * The C Programming Language.
 */

import acm.graphics.*;
import acm.program.*;

public class HelloProgram extends GraphicsProgram {
   public void run() {
      add(new GLabel("hello, world"), 100, 75);
   }
}
```

program comment

imports

main class

As Figure 2-1 indicates, **HelloProgram** is divided into three separate sections: a *program comment*, a list of *imports*, and the *main class*. Although its structure is extremely simple, **HelloProgram** is typical of the programs you will see in the next few chapters, and you should use it as a model of how Java programs should be organized.

Comments

The first section of **HelloProgram** is an English-language **comment**, which is simply program text that is ignored by the compiler. In Java, comments come in two forms. The first consists of text enclosed between the markers **/*** and ***/**, even if that text continues for several lines. The second—which I won't use in this text—is introduced by the symbol **//** and continues up to the end of the line. In **HelloProgram**, the comment begins with the **/*** on the first line and ends with the ***/** eight lines later.

Comments are written for human beings, not for the computer. They are intended to convey information about the program to other programmers. When the Java compiler translates a program into a form that can be executed by the machine, it ignores the comments entirely.

In this text, every program begins with a special comment called the **program comment** that describes the operation of the program as a whole. That comment includes the name of the program file and a message that describes the operation of the program. In this case, the program comment also provides credit for the original idea of the program. Comments might also describe any particularly intricate parts of the program, indicate who might use it, offer suggestions on how to change the program behavior, or provide any additional information that other programmers might want to know about the program. For a program this simple, extensive comments are usually not necessary. As your programs become more complicated, however, you will discover that good comments are one of the best ways to make them understandable to someone else—or to figure out what you yourself intended if you return to a program after not looking at it for a while.

Imports

The second section of the program consists of the lines

```
import acm.graphics.*;
import acm.program.*;
```

These lines indicate that the program uses two additional library packages. A **library package** is a collection of tools written by other programmers that perform specific operations. The libraries used by the `HelloProgram` are a graphics library and a program library, each of which come from a collection of packages produced by the Association for Computing Machinery (ACM). The asterisk at the end of the package name is used to indicate that all components from the relevant package should be imported. Every program in this book will import at least the `acm.program` package, and any programs that use graphics will import `acm.graphics`, which means most of your programs will also need to include these lines immediately after the program comment. Some programs will use additional packages as well and must contain an `import` line for each one.

When you write your own programs, you can use the tools provided by these packages, which saves you the trouble of writing them yourself. Libraries are critical to programming, and you will quickly come to depend on several important packages as you begin to write more sophisticated programs.

The main class

The last section of the `HelloProgram.java` file is the `HelloProgram` class itself, which consists of the lines

```
public class HelloProgram extends GraphicsProgram {
   public void run() {
      add(new GLabel("hello, world"), 100, 75);
   }
}
```

These five lines represent the first example of a class definition in Java. A **class** is the primary unit into which Java programs are divided and constitute a template for the creation of individual objects. That definition is admittedly relatively vague at this point and will be refined in the section entitled "Classes and objects" later in this chapter.

As you did when you were introduced to classes in the context of Karel, it is important to understand this program by breaking it down hierarchically. The class definition itself looks like this:

```
public class HelloProgram extends GraphicsProgram {

                  body of the class definition

}
```

The first line of the class definition is called the **header line** and provides important information about the characteristics of the class. Here, the **extends** keyword indicates that `HelloProgram` is a subclass of `GraphicsProgram`, which is one of the program types defined in the `acm.program` package. The specific capabilities of the `GraphicsProgram` class will be defined in detail in Chapter 7. For the moment—while you're in the "programming by example" mode—it is sufficient to rely on your intuition.

HelloProgram is a **GraphicsProgram** and can therefore do any of the things that **GraphicsProgram** can do, the details of which you will discover later.

The body of the class definition for **HelloProgram** contains a single definition, which looks like this:

```
public void run() {
    add(new GLabel("hello, world"), 100, 75);
}
```

This definition is an example of a Java **method**, which is simply a sequence of program steps that have been collected together and given a name. The name of this method, as given on its header line, is **run**. The steps that the method performs are listed between the curly braces and are called **statements**. Collectively, the statements constitute the **body** of the method. The method **run** shown in the **HelloProgram.java** example contains only one statement, but it is common for methods to contain several statements that are performed sequentially, beginning with the first statement and continuing through the last statement in the body.

The method **run** plays a special role in programs that use the **acm.program** package. Whenever you run a Java program, the computer executes the statements enclosed in the body of the **run** method for the main class. In **HelloProgram**, the body of **run** consists of the single statement

```
add(new GLabel("hello, world"), 100, 75);
```

This statement uses two facilities from the library packages. The first is the **GLabel** class, which comes from **acm.graphics**. The part of the line that reads

```
new GLabel("hello, world")
```

creates a new **GLabel** object containing the text **"hello, world"**. The rest of the line is

add(*newly generated label* **, 100, 75);**

which takes the new **GLabel** and adds it to the graphics program at a position whose *x* and *y* coordinates are 100 and 75. The result is that the program produces the following image on the screen:
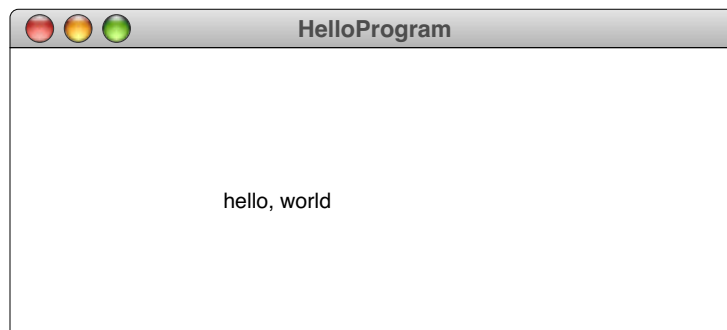
2.2 Perspectives on the programming process

The point of this chapter is not to understand the details of how Java works, but rather to get a good overall perception—what psychologists often call a **gestalt**—of a few simple programs. You could figure out a great deal about Java simply by typing the

`HelloProgram.java` file into the computer and experimenting with it, which is in fact the first exercise at the end of this chapter. It would be easy, for example, to change the message from **`"hello, world"`** to something more interesting. You could put that message at a different position on the screen by changing the numbers 100 and 75 to something else. In the process, you would presumably discover some interesting facts about the Java coordinate system, such as the fact that the origin is in the upper left corner instead of the lower left corner that you're familiar with from geometry. Thus, if you change the *y* coordinate from 75 to 100, the message moves *downward* on the screen. You might also get a feeling for what the units are: Java specifies coordinates in units corresponding to the individual dots on the display screen, which are called **pixels.** Thus the coordinate pair (100, 75) indicates a position on the screen that is 100 pixels inward from the left edge of the window and 75 pixels down from the top. And you could put more than one message on the screen simply by another statement similar in form to the single line in the existing program, but with a different message and location.

The take-home message here is that it is useful to experiment. As Brian Kernighan and Dennis Ritchie assert, "the only way to learn a new programming language is by writing programs in it." And the more programs you write and the more you play around with those programs, the more you will learn about how programming works.

At some point, of course, you will need to learn about the details of the Java statements so that you can understand how each statement works. This detailed view, however, is not the only useful way to look at a program. Sometimes it helps to stand back and look at a program as a whole. The step-by-step detailed view is an example of a *reductionistic* approach. If you look at a program from a more global perspective—as a complete entity whose operation as a whole is of primary concern—you are adopting a more *holistic* perspective that allows you to see the program in a different light.

Reductionism is the philosophical principle that the whole of an object can best be understood by understanding the parts that make it up. Its antithesis is **holism**, which recognizes that the whole is often more than the sum of its parts. As you learn how to write programs, you must learn to see the process from each of these perspectives. If you concentrate only on the big picture, you will end up not understanding the tools you need for solving problems. However, if you focus exclusively on details, you will invariably miss the forest for the trees.

When you are first learning about programming, the best approach is usually to alternate between these two perspectives. Taking the holistic view helps sharpen your intuition about the programming process and enables you to stand back from a program and say, "I understand what this program does." On the other hand, to practice writing programs, you have to adopt enough of the reductionistic perspective to know how those programs are put together.

2.3 A program to add two numbers

Learning to program by example is easier if you have more examples. Figure 2-2 shows a different kind of program that asks the user to enter two integers (which is simply the mathematical term for whole numbers without fractional parts), adds those integers together, and then displays the sum.

FIGURE 2-2   **Program to add two integers**

```java
/*
 * File: Add2Integers.java
 * ----------------------
 * This program adds two integers and prints their sum.
 */

import acm.program.*;

public class Add2Integers extends ConsoleProgram {

   public void run() {
      println("This program adds two integers.");
      int n1 = readInt("Enter n1: ");
      int n2 = readInt("Enter n2: ");
      int total = n1 + n2;
      println("The total is " + total + ".");
   }

}
```

The program in Figure 2-2 introduces several new programming concepts that were not part of **HelloProgram**. First, **Add2Integers** is a different kind of program, as indicated by its header line:

> **public class Add2Integers extends ConsoleProgram**

This program extends **ConsoleProgram** instead of **GraphicsProgram**, which means that it has access to a different set of facilities. The **ConsoleProgram** class is designed to support user interaction in a traditional text-based style. A **ConsoleProgram** can request input from the user and display information back as illustrated in the following diagram, which shows what you might see if you ran the **Add2Integers** program:
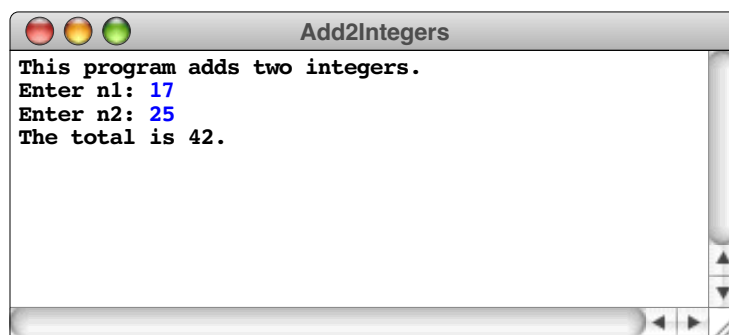
Diagrams that show the output that a program produces are called **sample runs.** Although it may be hard to see in this photocopied edition of the text, the input that the user types appears in blue, as it does on the display when you run this program. This convention makes it easy to tell which parts of a session are entered by the user and which parts are generated by the program.

If you take a holistic look of this program, I'm certain you could have told me what it did even before you knew what any of its statements really did. If nothing else, the

program comment and the first line of the **run** method are dead giveaways. But even without those signposts, most beginning programmers would have little trouble understanding the function of the code. Programs are typically easier to read than they are to write. Moreover, just as it is far easier to write a novel after having read a number of them, you will find that it is easier to write good programs if you take the time to read several well-designed programs and learn to emulate their structure and style. The meaning of a particular statement—much like unfamiliar words in a novel—often becomes clear either from the context or from simple common sense. Before going on to look at the explanations that follow, try taking a look at each line of **Add2Integers** and see how much sense you can make of it working from first principles alone.

The first line in the **run** method is

```
println("This program adds two integers.");
```

The **println** method—which is a contraction of "print line"—is used to display information on the console. The value in parentheses is called an **argument** and tells the **println** method what it should display. The double quotes surrounding the text

```
This program adds two integers.
```

do not appear in the output but are used by Java to indicate that the characters between the quotation marks are an instance of text data called a **string.** The effect of the **println** method is to display the entire argument string on the console and then to return to the beginning of the next line. Thus, if you make several **println** calls in succession, each string in the output will appear on a separate line.

Note that the purpose of the first line in this program is not to tell programmers reading the code what the program does; that function is accomplished by the program comments at the beginning of the file. The first **println** statement is there to tell the user sitting at the computer what the program does. Most people who use computers today are not programmers, and it wouldn't be reasonable to expect those users to look at the code for a program to determine what it did. The program itself has to make its purpose clear.

The next line of the **run** method looks like this:

```
int n1 = readInt("Enter n1: ");
```

At a holistic level, the intent of the line is reasonably clear, given that everything you can see suggests that it must be reading the first integer value. If you adopt a reductionistic perspective, however, this line of code introduces several new concepts. Of these, the most important is that of a **variable,** which is easiest to think of as a placeholder for some piece of data whose value is unknown when the program is written. When you write a program to add two integers, you don't yet know what integers the user will want to add. The user will enter those integers when the program runs. So that you can refer to these as-yet-unspecified values in your program, you create a variable to hold each value you need to remember, give it a name, and then use its name whenever you want to refer to the value it contains. Variable names are usually chosen so that programmers who read the program in the future can easily tell how each variable is used. In the **Add2Integers** program, the variables **n1** and **n2** represent the integers to be added, and the variable **total** represents the sum.

When you introduce a new variable in Java, you must **declare** that variable, which consists of making sure that the Java compiler knows the type of data that variable will

contain. In Java, the type used to store integer data is called **int**. A declaration of the form

> **int n1 =** `  value  ` **;**

introduces a new integer variable called **n1** whose value is given by whatever expression appears in the box labeled *value*. In this case, that expression is

> **readInt("Enter n1: ")**

Just like the **println** example from the first line, this expression is an invocation of the **readInt** method in **ConsoleProgram**. The **readInt** method begins by displaying its argument on the console so that the user knows what is expected; this type of message is generally called a **prompt**. Unlike **println**, however, the **readInt** method does not return to the beginning of the next line but waits after the prompt for the user to type in an integer. When the user has finished typing in the integer and hits the Return or Enter key, that integer is then passed back as the result of the **readInt** method. In programming terminology, we say that **readInt returns** the value the user typed.

When tracing through the operation of a program on paper, programmers often use box diagrams to indicate the values assigned to variables. If you look back at the sample run presented earlier in this section, you will see that the user entered the value 17 in response to the first input request. Thus, to illustrate that the assignment statement has stored the value 17 in the variable **n1**, you draw a box, name the box **n1**, and then indicate its value by writing a 17 inside the box, as follows:

n1

| 17 |
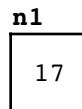
The third line in the run method is almost exactly the same as the second and reads a value for the variable **n2**. If the user enters 25 in response to this prompt, you could update your box diagram to show the new variable, as follows:

n1   **n2**

| 17 | | 25 |
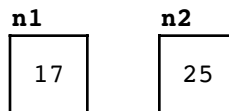
The next line in the **run** method is

> **int total = n1 + n2;**

This statement declares the variable **total** and assigns it the value to the right of the equal sign, which is

> **n1 + n2**

This piece of the code is an example of an essential programming construct called an *expression* that represents the result of computation. The structure of expressions is defined more formally in Chapter 3, but it is often easy to understand what a Java expression means given that they look very much like expressions in traditional mathematics.

In **Add2Integers**, the goal is to add the values stored in the variables **n1** and **n2**. To do so, you use the **+** operator, which you've understood since elementary-school arithmetic. To keep track of the result, you need to store it in some variable, and this statement introduces **total** for precisely this purpose.

The final statement in the **run** method is

```
println("The total is " + total + ".");
```

which accomplishes the task of displaying the computed result. For the most part, this statement looks like the first statement in the program, which is also a call to the **println** method. This time, however, there's a new twist. Instead of taking a single string argument, this statement passes to **println** the argument value

```
"The total is " + total + "."
```

Just like the **n1 + n2** expression from the previous statement, this value is given by an expression involving the **+** operator. In this statement, however, at least some of the values to which **+** is applied are string data rather than the numeric data on which addition is defined. In Java, applying the **+** operator to string data reinterprets that operator to mean adding the strings together end to end to combine their characters. This operation is called **concatenation.** If there are any parts of the expression that are not strings, Java converts them into their standard string representation before applying the concatenation operator. The effect of this last **println** statement is to display the value of **total** after adding on the surrounding text. You can see the effect of this statement in the sample run.

Although **Add2Integers** is set up to work only with integers, Java is capable of working with many other types of data. You could, for example, change this program so that it added two real numbers simply by changing the types of the variables and the names of the input methods, as shown in Figure 2-3.

FIGURE 2-3   **Program to add two double-precision numbers**

```java
/*
 * File: Add2Doubles.java
 * ----------------------
 * This program adds two double-precision floating-point numbers
 * and prints their sum.
 */

import acm.program.*;

public class Add2Doubles extends ConsoleProgram {

   public void run() {
      println("This program adds two numbers.");
      double n1 = readDouble("Enter n1: ");
      double n2 = readDouble("Enter n2: ");
      double total = n1 + n2;
      println("The total is " + total + ".");
   }

}
```

In most programming languages, numbers that include a decimal fraction are called **floating-point numbers**, which are used to approximate real numbers in mathematics. The most common type of floating-point number in Java is the type **double**, which is short for double-precision floating-point. If you need to store floating-point values in a program, you must declare variables of type **double**, just as you previously had to declare variables of type **int** to write **Add2Integers**. The only other change in the program is that the user input is obtained by calling **readDouble** instead of **readInt**. The basic pattern of the program is unchanged.

2.4 Classes and objects

Before continuing on to consider the details of expressions and statements in Chapters 3 and 4, it is important to introduce one more high-level concept illustrated by the examples in this chapter. The programs you've seen—**HelloProgram**, **Add2Integers**, and **Add2Doubles**—are each defined as Java *classes,* although the text has so far been rather vague as to the precise meaning of that term. These classes, moreover, are defined as extensions of existing classes supplied by the **acm.program** package: **HelloProgram** is an extension of **GraphicsProgram**, and the other two are extensions of **ConsoleProgram**. Whenever you define a new class as an extension of an existing one, the new class is said to be a **subclass** of the original. Thus, **HelloProgram** is a subclass of **GraphicsProgram**. Symmetrically, **GraphicsProgram** is a **superclass** of **HelloProgram**.

The concept of a class is one of the most important ideas in Java. At its essence, a **class** is as an extensible template that specifies the structure of a particular style of object. An **object,** as you already know from the discussion in the Karel book, is a conceptually integrated entity that encapsulates both state and behavior. Each object is an instance of a particular class, which can, in turn, serve as a template for many different objects. If you want to create objects in Java, you must first define the class to which those objects belong and then construct individual objects that are instances of that class.

Class hierarchies

Classes in Java form hierarchies. These hierarchies are similar in structure to many more familiar classification structures such as the organization of the biological world originally developed by the Swedish botanist Carl Linnaeus in the 18th century. Portions of this hierarchy are shown in the diagram in Figure 2-4. At the top of the chart is the universal category of all living things. That category is subdivided into several kingdoms, which are in turn broken down by phylum, class, order, family, genus, and species. At the bottom of the hierarchy shown in Figure 2-4 is the type of creature that biologists name using the genus and species together. In this case, the bottom of the hierarchy is occupied by *Iridomyrmex purpureus,* which is a type of red ant. The individual red ants in the world correspond to the objects in a programming language. Thus, each of the individuals

is an instance of the species *purpureus*. By virtue of the hierarchy, however, that individual is also an instance of the genus *Iridomyrmex,* the class *Insecta,* and the phylum *Arthropoda.* It is similarly, of course, both an animal and a living thing. Moreover, each red ant has the characteristics that pertain to each of its ancestor categories. For example, red ants have six legs, which is one of the defining characteristics of the class *Insecta*.

FIGURE 2-4 Levels in the biological classification hierarchy
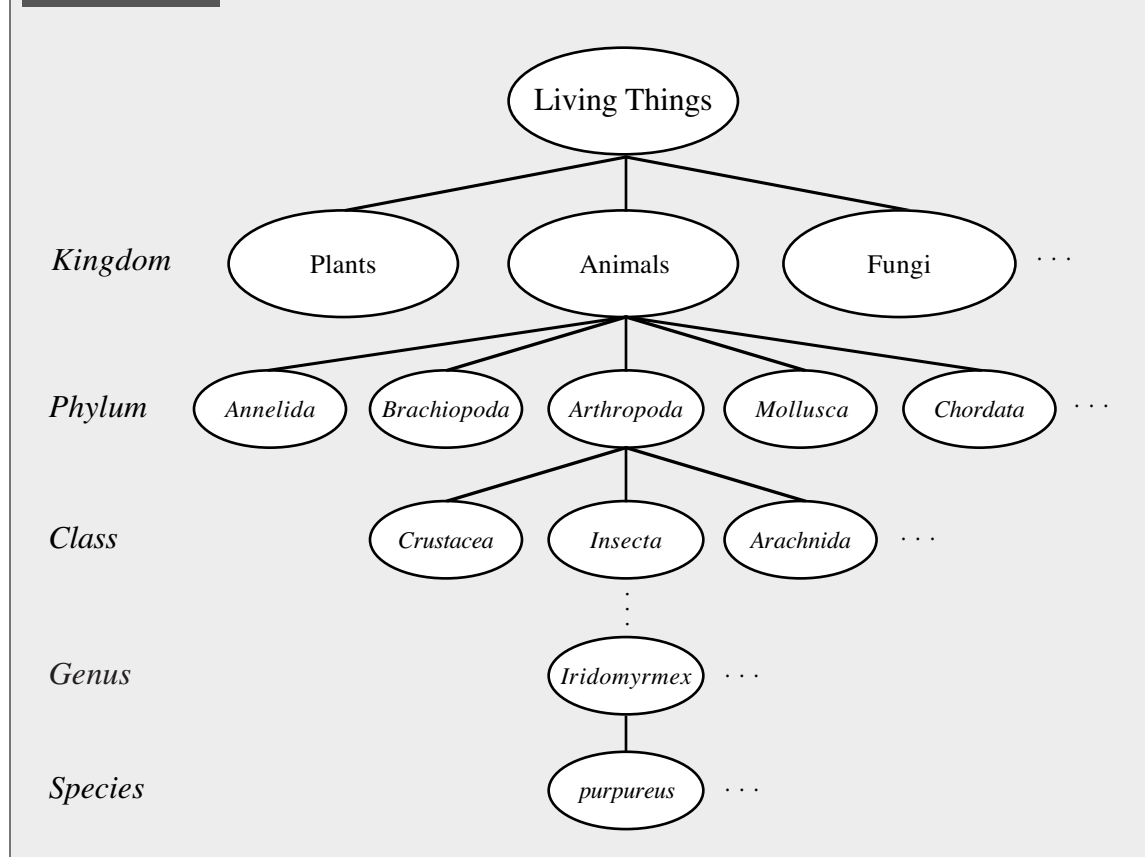
The biological metaphor illustrates one of the fundamental properties of classes in Java. Any instance of a particular class is also an instance of every one of its superclasses. Thus, any instance of **HelloProgram** is, by definition, also an instance of **GraphicsProgram**. Moreover, each instance of **HelloProgram** automatically acquires the public behavior of **GraphicsProgram**. This property of taking on the behavior of your superclasses is called **inheritance**.

The **Program** class hierarchy

The classes defined by the **acm.program** form a hierarchy with a little more structure and complexity than you have seen up to this point. That hierarchy appears in Figure 2-5. Each of the classes you have seen—**GraphicsProgram** and **ConsoleProgram**—is a subclass of a higher-level class called **Program**, which is in turn a subclass of the standard Java class called **Applet**. The diagram tells you that every instance of a program you design, such as the instance of **HelloProgram** shown in Figure 2-1, is simultaneously a **GraphicsProgram**, a **Program**, and an **Applet**. The fact that it is an applet means that you can run it in a web browser, which is a property that all **Applet**s share, along with all **Program**s and **GraphicsProgram**s, by inheritance.

Figure 2-5 also shows that there is another **Program** subclass besides the two you have already seen. The **DialogProgram** subclass turns out to be quite similar in its overall organization to **ConsoleProgram**. In particular, it shares exactly the same set of methods, which are in fact all specified by the **Program** class. What's different is that these

FIGURE 2-5 The Program class hierarchy
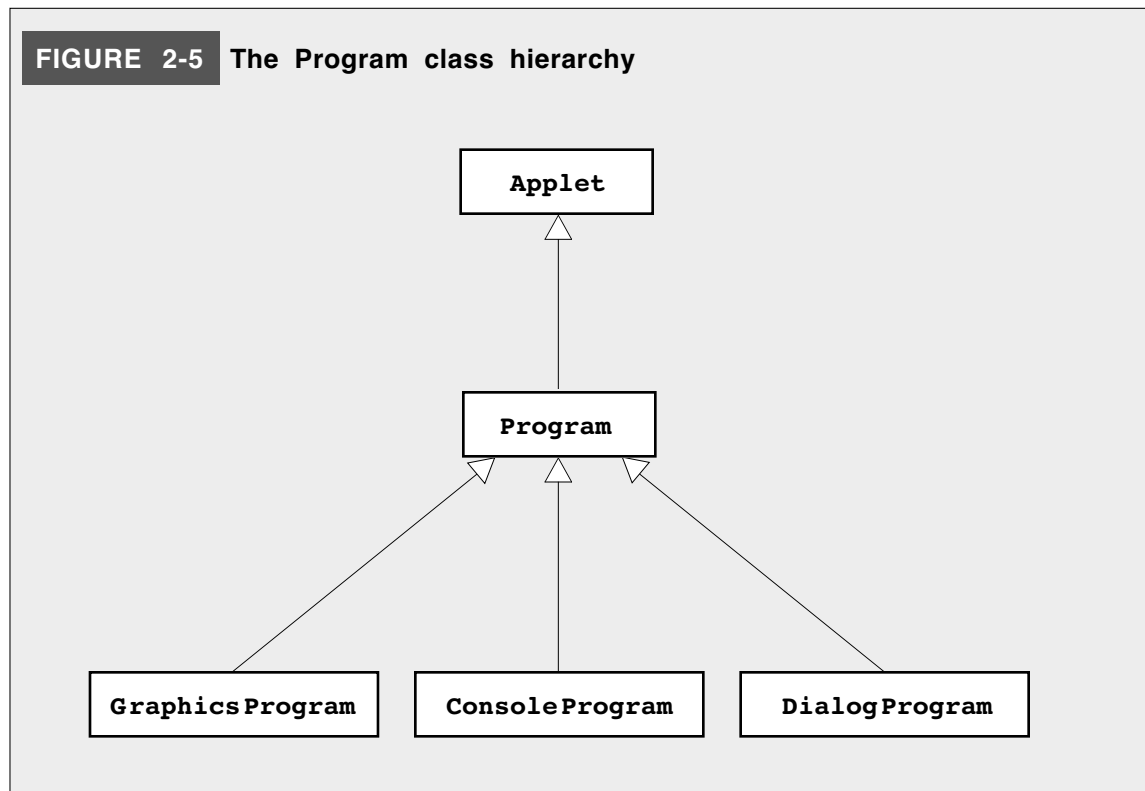
methods have a different interpretation. In a `ConsoleProgram`, methods like `println` and `readInt` specify user interaction with a console. In a `DialogProgram`, these same methods specify user interaction through interactive dialogs that pop up as the program runs. If you were to change the class header line shown in Figure 2-2 so that it read

```
public class Add2Integers extends DialogProgram {
```

the program would still add two numbers, but the interaction style would be quite different. Running this new version program produces the series of dialogs shown in Figure 2-6.

What's going on in this example is that `ConsoleProgram` and `DialogProgram` each define their own version of `println` and `readInt` so that they operate in the style appropriate to that class. Redefining an existing method so that it does something different from that in its superclass is called **overriding** that method.

The `GObject` class hierarchy

You've actually seen one instance of another class hierarchy that also serves as a good illustration of the idea of extension by subclassing. The `HelloProgram` presented at the very beginning of the chapter used the class `GLabel` to display the text `"hello, world"`. The `GLabel` class is only one of many subclasses of the `GObject` class, which is defined in the `acm.graphics` package to represent the universe of graphical objects. The portion of that hierarchy consisting of the collection of "shapes" that can be displayed in a `GraphicsProgram` appears in Figure 2-7.

FIGURE 2-6   **Dialogs that appear when running Add2Dialog**

Message	Input
This program adds two numbers.	Enter n2: 729
OK	OK

Input	Message
Enter n1: 1000	This total is 1729.
OK	OK
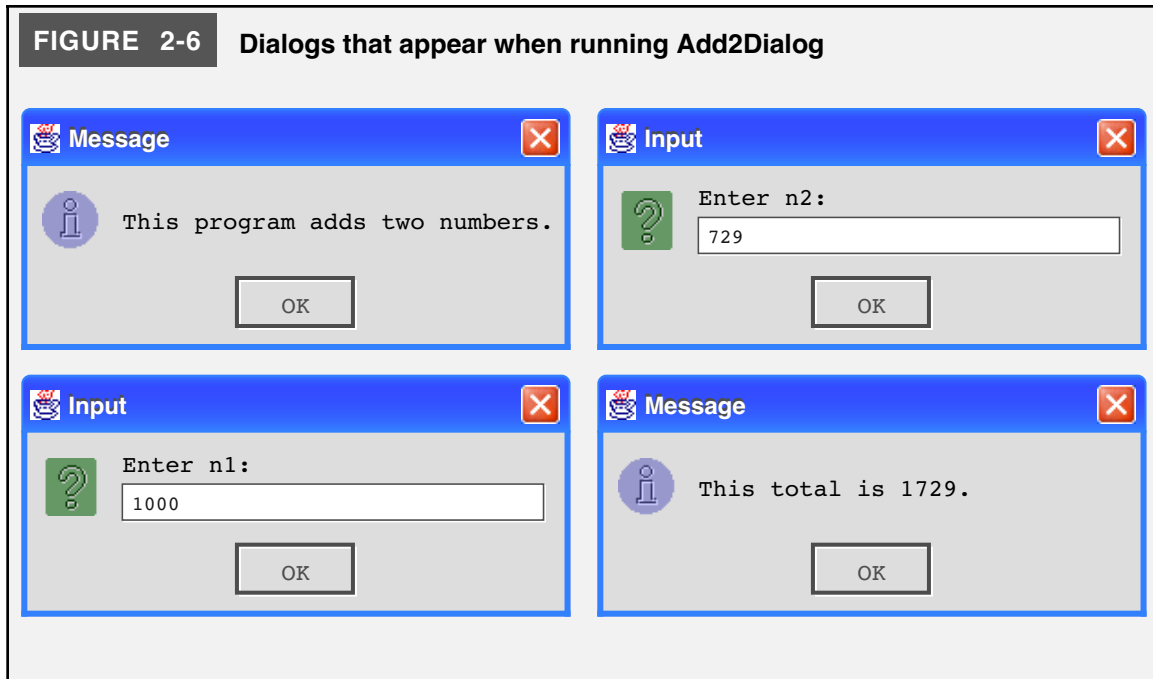
You will have a chance to learn about all these classes in Chapter 7, but it is possible to start using them even if you know only a very little bit about them. Consider, for example, the **GRect** and **GOval** classes. Each of these classes is constructed in a similar way. To create one of these objects, you use the keyword **new** followed by the name of the class and a sequence of four arguments specifying the *x* position, the *y* position, the width, and the height, respectively. As with all coordinates in Java, the *x* and *y* positions indicate the location of the upper-left corner, and all values are measured in pixel units.

FIGURE 2-7   **The shape classes in the GObject hierarchy**
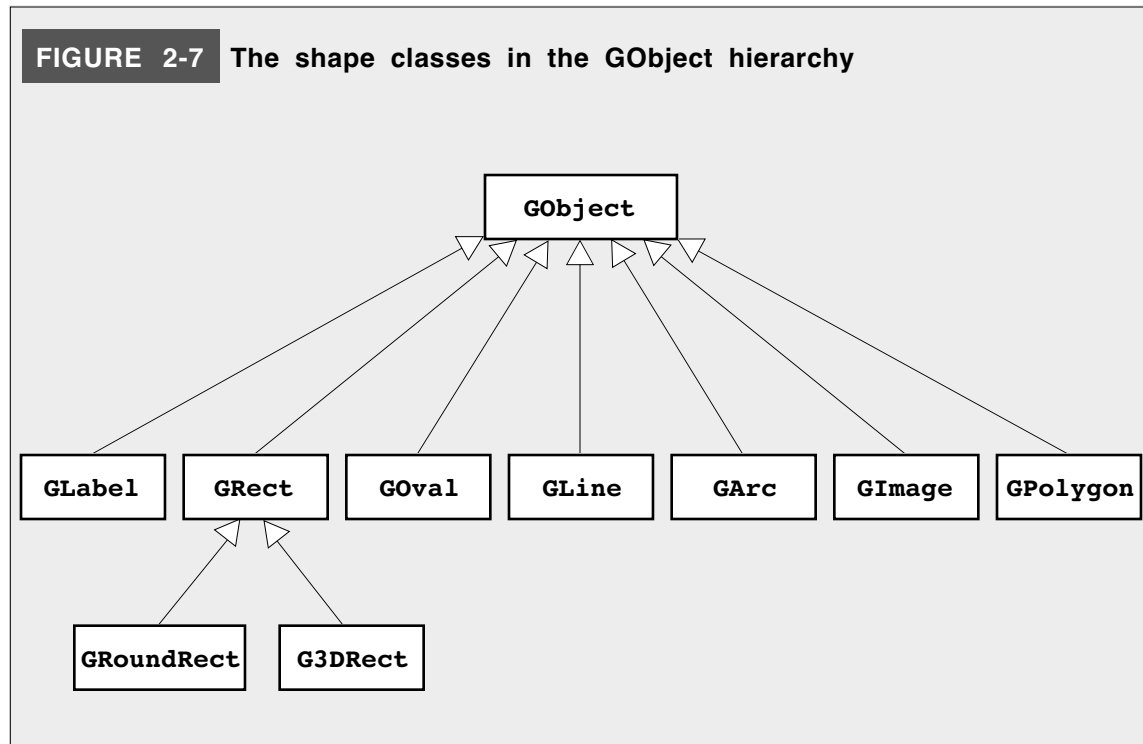
A `GRect` object draws the outline of rectangle that completely fills that area; a `GOval` object draws the outline of the largest oval that fits inside that box. You can use these classes to create more interesting graphical programs, such as the `DrawRobot` shown in Figure 2-8, which produces the following diagram:
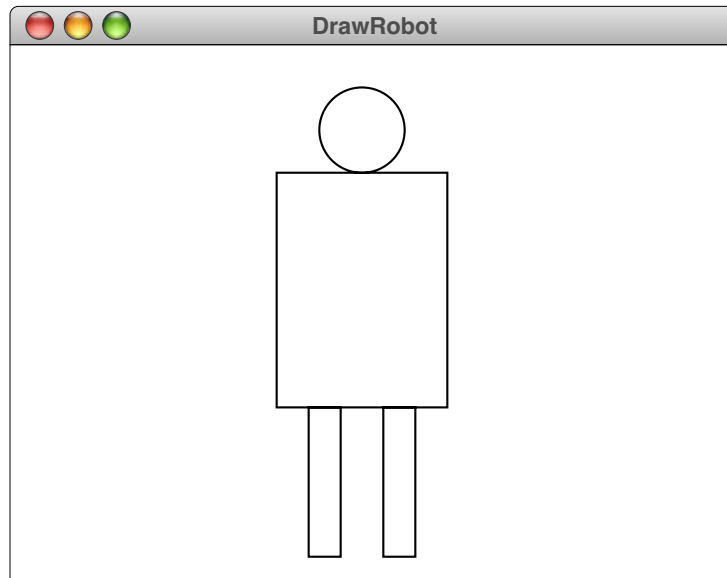
Because the goal of this chapter is to learn programming by example, and because there is no better way to learn programming than by doing it, you should set the book aside and try writing some simple programs that use the features described in this chapter. Several such problems are given in the exercises at the end of this chapter, but you can also use your own imagination.

FIGURE 2-8   **Program to draw the outline of a simple robot**

```
/*
 * File: DrawRobot.java
 * --------------------
 * This program draws a simple robot diagram in the window.  Its
 * programming style leaves much to be desired, mostly because
 * the coordinate values are specified explicitly and not defined
 * so that they automatically adjust according to specified
 * parameters of the image as a whole.  You will learn how to
 * improve the style in Chapter 7.
 */

import acm.graphics.*;
import acm.program.*;

public class DrawRobot extends GraphicsProgram {
   public void run() {
      add(new GOval(120, 20, 40, 40));
      add(new GRect(100, 60, 80, 110));
      add(new GRect(115, 170, 15, 70));
      add(new GRect(150, 170, 15, 70));
   }
}
```

Summary

In this chapter, you have had the opportunity to look at several complete Java programs to get an idea of their general structure and how they work. The details of those programs have been deferred to later chapters; at this point, your principal objective has been to focus on the programming process itself by adopting a holistic view. Even so, by building on the programming examples provided here, you should be ready at this point to write simple programs that involve only the following operations:

• Reading in numeric values entered by the user, either on a console or through a dialog
• Displaying text on a console
• Generating graphical programs formed from rectangles, ovals, and labels

   Important points about programming introduced in this chapter are:

• Well-written programs contain *comments* that explain in English what the program is doing.
• Most programs use *packages* that provide tools the programmer need not recreate from scratch. The programs in this chapter use two packages: **acm.program** and **acm.graphics**. Subsequent chapters will introduce additional packages.
• You gain access to packages by adding at the top of the program an **import** line for that package.
• Every Java program used in this text will consist of a class definition that extends one of the classes in the **acm.program** package. That class is called the *main class*.
• Every main class contains a method called **run**. When the program is executed, the statements in the body of **run** are executed in order.
• To accept input typed by the user, you use the methods **readInt** and **readDouble**, depending on the type of data.
• To display messages and data values on the computer screen, you use the method **println**.
• Classes form hierarchies that reflect the **extends** relationship. If class **A** extends class **B**, then **A** is a *subclass* of **B** and **B** is a *superclass* of **A.**
• Subclasses *inherit* the public behavior of their superclass.
• The **Program** class in **acm.program** has three defined subclasses: **GraphicsProgram**, **ConsoleProgram**, and **DialogProgram**.
• The **GObject** class in **acm.graphics** has many useful subclasses. Although you won't have a chance to see the details until Chapter 7, you can use the **GLabel**, **GRect**, and **GOval** classes to create simple pictures.

Review questions

1. What is the purpose of the comments shown at the beginning of each program in this chapter?

2. What is the role of a library package?

3. What is the name of the method that is executed when a Java program starts up under the control of the **acm.program** package?

4. To what does the word *argument* refer in programming? What purpose do arguments serve?

5. Describe the function of the **println** method. What is the significance of the letters **ln** at the end of its name?

6. What is the purpose of the **readInt** method? How would you use it in a program?

7. This chapter describes two uses for the **+** operator in Java programs. What are they, and how does Java determine which interpretation to use?

8. Describe the difference between the philosophical terms *holism* and *reductionism*. Why are these concepts important to programming?

9. What is the difference between a class and an object?

10. Define the terms *subclass, superclass*, and *inheritance*.

11. What is the difference between a **ConsoleProgram** and a **DialogProgram**?

Programming exercises

1. Type in the **HelloProgram.java** program exactly as it appears in this chapter and get it working. Change the message so that it reads **"I love Java"** instead. Add your name as a signature to the lower right corner.

2. The following program was written without comments or instructions to the user, except for a couple of input prompts:

```
import acm.program

public class MyProgram extends ConsoleProgram {
   public void run() {
      double b = readDouble("Enter b: ");
      double h = readDouble("Enter h: ");
      double a = (b * h) / 2;
      println("a = " + a);
   }
}
```

    Read through the program and figure out what it is doing. What result is it calculating? Rewrite this program so it is easier to understand, both for the user and for the programmer who must modify the program in the future.

3. Extend the **Add2Integers** program given in Figure 2-2 so that it adds three numbers instead.

4. Write a **GraphicsProgram** subclass that generates the following picture of a snowperson:
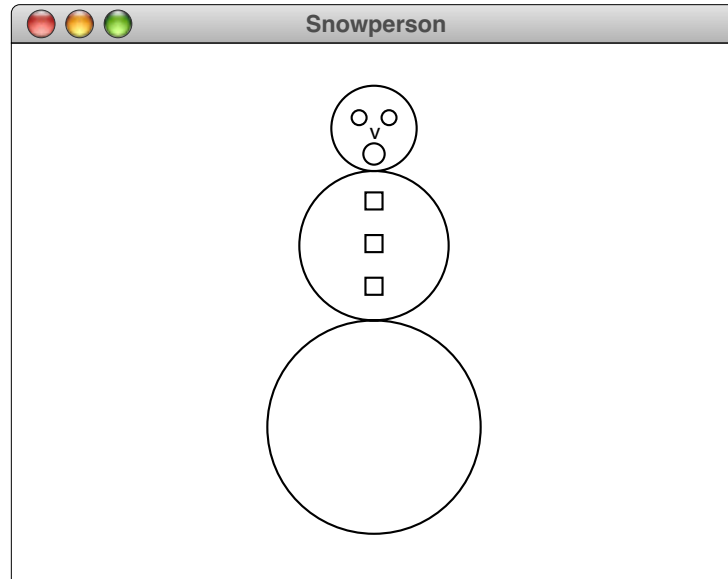
Play with the coordinates until you get something that looks more or less right. The nose is simply a **GLabel** containing the letter **v**.

Chapter 3
Expressions

"What's twice eleven?" I said to Pooh.
("Twice what?" said Pooh to Me.)
"I think *it ought to be twenty-two."*
"Just what I think myself," said Pooh.

— A. A. Milne, "Us Two," *Now We Are Six,* 1927

Dennis Ritchie

As a young research scientist at Bell Labs, Dennis Ritchie created the C programming language in 1970 to simplify the development of the Unix operating system that he developed jointly with Ken Thompson. Both Unix and C have had enormous impact on the field of computing. C, in particular, provided the foundation from which many of today's most important programming languages, including Java, got their start. Ritchie's design of C's expression syntax has been carried over into Java in almost exactly its original form. Ritchie and Thompson jointly received the ACM Turing Award—the computing profession's highest honor—in 1983.

In Chapter 2, you had the opportunity to see a couple of simple programs—**Add2Integer** and **Add2Double**—that perform arithmetic manipulation. The idea in Chapter 2 was for you to get a sense for what that type of program looks like. In this chapter, your goal is to learn the underlying rules that allow you to write Java programs that specify computation. In doing so, you will combine a reductionistic understanding of the details with the holistic understanding of the program as a whole.

In each of these programs, the actual computation is specified by the appearance in the code of

    n1 + n2

which indicates, naturally enough, that the computer should add the values stored in the variables **n1** and **n2**. This combination of variables and the **+** operator is an example of an **expression,** which is the symbolic representation in a programming language of the sequence of operations necessary to produce a desired result. To make programming easier for humans, these expressions tend to have a syntactic form that mirrors as closely as possible the representation used in traditional mathematical formulae. Indeed, the name FORTRAN, which was the first higher-level language to include expression syntax, is a contraction of the words *formula translation,* because that aspect of the language was widely seen as its principal advantage over machine-language programming.

As in most languages, an **expression** in Java is composed of *terms* and *operators*. A **term**, such as **n1** and **n2** in the expressions from Chapter 2, represents a single data value. An **operator**, such as the **+** sign, is a character (or sometimes a short sequence of characters) that indicates a computational operation. In an expression, a term must be one of the following:

- *A constant*. An explicit data value that appears as part of the program is called a **constant**. Numbers such as 0 or 3.14159 are examples of constants.
- *A variable*. Variables serve as placeholders for data that can change during the execution of a program.
- *A method call*. Values are often generated by calling other methods, possibly in library packages, that return data values to the original expression. In the **Add2Integer** program, for example, the method **readInt** is used to read in each of the input values; the method call **readInt** is therefore an example of a term. Method calls are discussed further in Chapter 5.
- *An expression in parentheses*. Parentheses may be used in an expression to indicate the order of operations, in the same way they are used in mathematics. From the compiler's point of view, the expression in parentheses becomes a term, which must be handled as a unit before computation can proceed.

When a program is run, the process of carrying out each of the specified operations in an expression is called **evaluation**. When an expression is evaluated, each operator is applied to the data values represented by the surrounding terms. After all the operators have been evaluated, what remains is a single data value that represents the result of the computation. For example, given the expression

    n1 + n2

the evaluation process consists of taking the values in the variables **n1** and **n2** and adding them together, and the result of the evaluation is whatever that sum happens to be.

3.1 Primitive data types

Before turning to the details of expression formation, it is useful to take a quick survey of the types of data that expressions can manipulate. In much of this text, you will work primarily with data objects that are representatives of a class. Those objects and classes represent the defining characteristic of the object-oriented paradigm on which Java is based. Expressions, however, tend to work with simpler types of data that—at least in Java—fall outside of the object hierarchy. These data types are all predefined as part of the language and represent the **primitive data types** from which more complex objects are built.

To be useful in a wide variety of applications, programs must be able to store many different types of data. As you saw in Chapter 2, it was possible to write programs like **Add2Integer** that work with whole numbers but also programs like **Add2Double** in which numbers had fractional parts, such as 1.5 or 3.1415926. When you use a word-processing program, the individual data values are characters, which are then assembled into larger units such as words, sentences, and paragraphs. As your programs get more complicated, you will begin to work with large collections of information structured in a variety of ways. All of these different classes of information constitute **data**.

Whenever you work with some piece of data—which might be an integer or a number with a fractional part or a character—the Java compiler needs to know its **data type**. Holistically speaking, a data type is defined by two properties: a set of values, or domain, and a set of operations. The **domain** is simply the set of values that are elements of that type. For example, the domain of the type **int** includes all integers (. . . –2, –1, 0, 1, 2 . . .) up to the limits defined by the Java language. For character data, the domain is the set of symbols that appear on the keyboard or that can be displayed on the terminal screen. The **set of operations** comprises the tools you have to manipulate values of that type. For example, given two integers, you might multiply them together or divide one by another. Given text data, on the other hand, it is hard to imagine what an operation like multiplication might mean. You would instead expect to use operations such as comparing two words to see if they are in alphabetic order or displaying a message on the screen. Thus, the operations must be appropriate to the elements of the domain, and the two components together—the domain and the operations—define the data type.

Java defines eight primitive data types, which appear in Figure 3-1. The first four types—**byte**, **short**, **int**, and **long**—represent integers with different maximum and minimum values that reflect the capacity of the memory cells in which they are stored. The next two—**float** and **double**—represent floating-point numbers, again with different dynamic ranges. Except in those rare cases in which a library method requires the use of one of the other types, this text will use **int** and **double** as its standard numeric types. The type **boolean** has a domain consisting of only two values: **true** and **false**. Boolean data plays a central role in controlling the flow of a program and is discussed in Chapter 4. The type **char** is used to represent character data and is covered in Chapter 8.

In addition to these primitive types, it is often useful to think of the Java type **String** as if it were primitive. It is, in fact, a class defined in the package **java.lang**, but it is integrated into the language in several ways that make it easy to regard as roughly analogous to the built-in primitive types. For example, Java provides a special syntax for string constants. Moreover, just as you could with any class, you can declare variables of type **String** in your programs and use them without considering their underlying structure. Having **String** available makes it easier to write more interactive programs and helps to counter the common predisposition to think of data as numeric.

FIGURE 3-1	Primitive types in Java	
Type	**Domain**	**Common operations**

`byte`	8-bit integers in the range –128 to 127	*The arithmetic operators:*
		  + add ***** multiply
		  – subtract **/** divide
		                           % remainder

`short`	16-bit integers in the range –32768 to 32767	
`int`	32-bit integers in the range –2147483648 to 2147483647	*The relational operators:* **==** equal **!=** not equal **<** less than **<=** less or equal **>** greater than **>=** greater or equal
`long`	64-bit integers in the range –9223372036854775808 to 9223372036854775807	
`float`	32-bit floating-point numbers in the range $\pm 1.4 \times 10^{-45}$ to $\pm 3.4028235 \times 10^{38}$	*The arithmetic operators except* **%**
`double`	64-bit floating-point numbers in the range $\pm 4.39 \times 10^{-322}$ to $\pm 1.7976931348623157 \times 10^{308}$	*The relational operators*
`boolean`	the values **true** and **false**	*The logical operators:* **&&** and **||** or **!** not
`char`	16-bit characters encoded using Unicode	*The relational operators*

3.2 Constants and variables

The two simplest types of terms in an expression are constants and variables. Constants appear when you need to use an explicit value that doesn't change over the course of the program. Variables are placeholders for data that can change during execution. The sections that follow show you how to write constants and variables and detail the rules that Java imposes on their specification.

Constants

When you write a formula in mathematics, some symbols in the formula typically represent unknown values while other symbols represent constants whose values are known. Consider, for example, the mathematical formula for computing the circumference (*C*) of a circle given its radius (*r*):

$$C = 2 \pi r$$

To translate this formula into a expression, you would use variables to record the radius and circumference. These variables change depending on the data. The values 2 and π, however, are constants—explicit values that never change. The value 2 is an integer constant, and the value π is a real number constant, which would be represented in a program by a floating-point approximation, such as 3.14159265358979323846. Because constants are an important building block for constructing expressions, it is important to be able to write constant values for each of the basic data types.

- *Integer constants*. To write an integer constant as part of a program or as input data, you simply write the digits that make up the number. If the integer is negative, you write a minus sign before the number, just as in mathematics. Commas are never used. Thus, the value one million must be written as 1000000 and not as 1,000,000.

- *Floating-point constants*. Floating-point constants in Java are written with a decimal point. Thus, if **2.0** appears in a program, the number is represented internally as a floating-point value; if the programmer had written **2**, this value would be an integer.

Floating-point values can also be written in a special programmer's style of scientific notation, in which the value is represented as a floating-point number multiplied by a integral power of 10. To write a number using this style, you write a floating-point number in standard notation, followed immediately by the letter **E** and an integer exponent, optionally preceded by a **+** or **-** sign. For example, the speed of light in meters per second is approximately

$$2.9979 \times 10^8$$

which can be written in Java as

`2.9979E+8`

where the **E** stands for the words *times 10 to the power*.

Boolean constants and character constants also exist and are described in subsequent chapters along with their corresponding types.

There is, however, one additional form of constant that you need to know about, even though it is not a primitive type:

- *String constants*. You write a string constant in Java by enclosing the characters that comprise the string in double quotation marks. For example, the very first example of data used in this text was the string

  `"hello, world"`

  in the **`HelloProgram`** example from Chapter 2. This string consists of the characters shown between the quotation marks, including the letters, the comma, and the space. The quotation marks are not part of the string but serve only to mark its beginning and end. There are several additional rules for writing string constants that allow you to include special characters (such as quotation marks) within the string. These rules are described in detail in Chapter 8.

Variables

A **variable** is a placeholder for a value and has three important attributes: a *name*, a *value*, and a *type*. To understand the relationship of these attributes, it is easiest to think of a variable as a box with a label attached to the outside. The name of the variable appears on the label and is used to tell the different boxes apart. If you have three boxes (or variables), you can refer to a particular one using its name. The value of the variable corresponds to the contents of the box. The name on the label of the box never changes, but you can take values out of a box and put new values in as often as you like. The type of the variable indicates what kind of data values can be stored in the box. For example, if you have a box designed to hold values of type **int**, you cannot put values of type **string** into that box.

Names in Java—for variables as well as other sorts of things such as classes and methods—are called **identifiers.** Identifiers must be constructed according to the following rules:

1. Identifiers must start with a letter or the underscore character (_). In Java, uppercase and lowercase letters appearing in an identifier are considered to be different, so the names **ABC**, **Abc**, and **abc** refer to three separate identifiers.
2. All other characters in an identifier must be letters, digits, or the underscore. No spaces or other special characters are permitted. Identifiers can be of any length.

3. Identifiers must not be one of the following **reserved words**, which are names that Java defines for a specific purpose:

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

In addition, good programming style requires two more rules, as follows:

4. Identifiers should make obvious to the reader the purpose of the variable, class, or method. Although names of friends, expletives, and the like may be legal according to the other rules, they do absolutely nothing to improve the readability of your programs.

5. Identifiers should match the case conventions that have become standard in Java. In particular, variable names should begin with a lowercase letter; class names should begin with an uppercase letter. Thus, **n1** is appropriate as the name of a variable, and **HelloProgram** is appropriate as the name of a class. Each additional English word appearing in the name is typically capitalized to improve readability, as in the variable name **numberOfStudents**. Names that are used as constants, which are discussed in the section entitled "Named constants" later in this chapter, use only uppercase letters and separate words with underscores, as in **PLANCKS_CONSTANT**.

Declarations

As noted in the discussion of the **Add2Integers** program in Chapter 2, you must explicitly specify the data type of each variable when you introduce it into a program. This process is known as declaring the variable. The syntax for declaring a variable is expressed in the following line:

> *type identifier* **=** *expression***;**

The code line above with its italicized pieces is an example of a **syntax template.** The words in italics represent items you can fill in with anything that fits the specification of that template. In writing an assignment statement, for example, you can declare variables of any type, use any identifier to the left of the equal sign, and supply any expression on the right. The boldface items in the template—in this case the equal sign and the semicolon—are fixed. Thus, in order to write a declaration, you start with the type name, followed by the variable name, an equal sign, an expression, and a semicolon, in that order.

Syntax for declarations:

> *type identifier* **=** *expression***;**

where:
> *type* is the type of the variable
> *identifier* is the name of the variable
> *expression* specifies the initial value

When new syntactic constructs are introduced in this text, they will be accompanied by a **syntax box** that defines their structure, such as the one shown on the right. Syntax

boxes contain a capsule summary of the grammatical rules for Java and serve as a handy reference.

The equal sign and the expression giving the initial value are in fact optional in a variable declaration. Leaving them out, however, can often lead to programming errors because the initial value of the variable depends on the context in which it is declared. In this text, all variables declared inside methods will be given an initial value to eliminate this source of error.

Variables can be declared in several different parts of the program. The variables you have seen up to this point have all been declared inside the body of a method. (So far, that method has always been `run`, but it is legal to declare variables in any method.) Variables declared inside a method are called **local variables** because they are available only to that method and not to any other parts of the code. You can, however, declare variables within the definition of a class but outside of any method. Variables defined at this level are called **instance variables,** or **ivars** for short, and are stored as part of each object. Instance variables must be used with a certain amount of care, and it is probably best to avoid them until you have a chance to learn about some of the appropriate strategies in Chapter 5. There is, however, one additional style of variable declaration that is particularly important for good programming style. These variables are called **class variables** and, as their name suggests, are defined at the level of the class rather than at the level of a method. Unlike instance variables, class variables are associated with all objects of a particular class and not with individual objects. Class variables must also be used with care, but they are precisely the right tool for introducing names for constants, as discussed in the following section.

Named constants

As you write your programs, you will find that you will often use the same constant many times in the same program. If, for example, you are performing geometrical calculations that involve circles, the constant π comes up frequently. Moreover, if those calculations require high precision, you might actually need all the digits that fit into a value of type `double`, which means you would be working with the value 3.14159265358979323846. Writing that constant over and over again is tedious at best, and error-prone if you try to type it in each time by hand instead of cutting and pasting the value. It would be better if you could give this constant a name and then use it everywhere in the program. You could, of course, simply declare it as a local variable by writing

```
double pi = 3.14159265358979323846;
```

but you would then be able to use it only within the method in which it was defined. A better strategy is to declare it as a class variable like this:

```
private static final double PI = 3.14159265358979323846;
```

The keywords at the beginning of this declaration each provide some information about the nature of the declaration. The `private` keyword indicates that this constant can be used only within the class that defines it. It often makes more sense to declare constants to be `public`, but good programming practice suggests that you should keep all declarations private unless there is a compelling reason to do otherwise. The `static` keyword indicates that this declaration introduces a class variable rather than an instance variable. The `final` keyword declares that the value will not change after the variable is initialized, thereby ensuring that the value remains constant. It would not be appropriate, after all, to change the value of π (despite the fact that a bill was introduced in 1897 into the Indiana State Legislature attempting to do just that). The rest of the declaration

consists of the type, the name, and the value, as before. The only difference is that the name is written entirely in uppercase to be consistent with Java's naming scheme.

Using named constants offers several advantages. First, it usually makes the program easier to read. More importantly, however, using constants makes the program easier to maintain. Many constants in a program specify things that might change from release to release, even though they will be constant during the execution of any program. Suppose, for example, that you were writing one of the early networking programs and decided that you needed to impose a limit—as the designers of the ARPANET did in 1969—of 127 computers (which were called *hosts* in the ARPANET days) that could be connected. If Java had existed in those days, you might have declared a constant that looks like this:

```
private static final int MAXIMUM_NUMBER_OF_HOSTS = 127;
```

At some later point, however, the explosive growth of networking would force you to raise this bound. That process is relatively easy if you use named constants in your programs. To raise the limit on the number of hosts to 1023, it might well be sufficient to change this declaration so that it read

```
private static final int MAXIMUM_NUMBER_OF_HOSTS = 1023;
```

If you adopted this approach and used **MAXIMUM_NUMBER_OF_HOSTS** everywhere in your program in which you needed to refer to that maximum value, then making this simple change would automatically propagate to every part of the program in which this name was used. The situation would be entirely different if you had written the constant 127 everywhere instead. In that case, you would need to search through the entire program and change all instances of 127 used for this purpose to the larger value. And if you missed one, you would likely have a very hard time tracking down the bug.

3.3 Operators and operands

In an expression, the actual computational steps are indicated by symbolic operators that connect the individual terms. The simplest operators to define are those used for arithmetic expressions, which use the standard operators from arithmetic. The arithmetic operators that apply to all numeric data types are:

+ Addition
− Subtraction (or negation, if written with no value to its left)
* Multiplication
/ Division

Each of these operators forms a new expression by connecting two smaller expressions, one to the left and one to right of the operator. These subsidiary expressions (or **subexpressions**) to which the operator is applied are called the **operands** for that operator. For example, in the expression

```
x + 3
```

the operands for the **+** operator are the subexpressions **x** and **3**. Operands are often individual terms, but they can also be more complicated expressions. For example, in the expression

```
(2 * x) + (3 * y)
```

the operands to **+** are the subexpressions **(2 * x)** and **(3 * y)**.

As in conventional mathematics, the operator – can be used in two forms. When it is positioned between two operands, it indicates subtraction, as in **x – y**. When used with no operand to its left, it indicates negation, so **-x** denotes the negative of whatever value **x** has. When used in this way, the – operator is called a **unary operator** because it applies to a single operand. The other operators (including – when it denotes subtraction) are called **binary operators** because they apply to a pair of operands.

These new operators make it possible to write programs that compute much more interesting and useful results than the sum of two numbers. For example, suppose you want to write a program to convert a length given in inches to its metric counterpart in centimeters. All you really need to know is that 1 inch equals 2.54 centimeters; you can construct the rest of the program just by adapting lines from the **Add2Double** example from Chapter 2 and putting them back together in the appropriate way. The final result is shown in Figure 3-2.

Combining integers and floating-point numbers

In Java, values of type **int** and **double** can be freely combined. If you use a binary operator with two values of type **int**, the result is of type **int**. If either or both operands are of type **double**, however, the result is always of type **double**. Thus, the value of the expression

        n + 1

is of type **int**, if the variable **n** is declared as type **int**. On the other hand, the expression

        n + 1.5

is always of type **double**. This convention ensures that the result of the computation is as accurate as possible. In the case of the expression **n + 1.5**, for example, there would be no way to represent the .5 if the result were computed using integer arithmetic.

FIGURE 3-2   **Program to convert inches to centimeters**

```
/*
 * File: InchesToCentimeters.java
 * ------------------------------
 * This program converts inches to centimeters.
 */

import acm.program.*;

public class InchesToCentimeters extends ConsoleProgram {

   public void run() {
      println("This program converts inches to centimeters.");
      double inches = readDouble("Enter value in inches: ");
      double cm = inches * CENTIMETERS_PER_INCH;
      println(inches + "in = " + cm + "cm");
   }

   private static final double CENTIMETERS_PER_INCH = 2.54;
}
```

Integer division and the remainder operator

The fact that applying a binary operator to integer operands gives back an integer leads to an interesting situation with respect to division. If you write an expression like

```
9 / 4
```

Java's rules specify that the result of this operation must be an integer, because both operands are of type **int**. When the program evaluates this expression, it divides 9 by 4 and throws away any remainder. Thus, the value of the expression is 2, not 2.25. If you want to compute the mathematically correct result, at least one of the operands must be a floating-point number. For example, the three expressions

```
9.0 / 4
9 / 4.0
9.0 / 4.0
```

each produce the floating-point value 2.25. The remainder is thrown away only if both operands are of type **int**.

There is an additional arithmetic operator that computes a remainder, which is indicated in Java by the percent sign (**%**). The **%** operator requires that both operands be of type **int**. It returns the remainder when the first operand is divided by the second. For example, the value of

```
9 % 4
```

is 1, since 4 goes into 9 twice, with 1 left over. The following are some other examples of the **%** operator:

```
0 % 4  =  0                        19 % 4  =  3
1 % 4  =  1                        20 % 4  =  0
4 % 4  =  0                      2001 % 4  =  1
```

The **/** and **%** operators turn out to be extremely useful in a wide variety of programming applications. The **%** operator, for example, is often used to test whether one number is divisible by another. For example, to determine whether an integer **n** is divisible by 3, you just check whether the result of the expression **n % 3** is 0.

As a side note, it is not always easy to understand what happens when one or both of the operands to **%** are negative. Java does define the result in such cases, but the definition was developed to be consistent with the way machine architectures work, which is unfortunately different from what mathematicians would like. To obviate any confusion, this text will avoid using **%** with negative operands, and it probably makes sense to adopt the same convention in your own code.

Precedence

If an expression has more than one operator, the order in which those operators are applied becomes an important issue. In Java, you can always specify the order by putting parentheses around individual subexpressions, just as you would in traditional mathematics. For example, the parentheses in the expression

```
(2 * x) + (3 * y)
```

indicate that Java should perform each of the multiplication operations before the addition. But what happens if the parentheses are missing? Suppose that the expression is simply

```
2 * x + 3 * y
```

How does the Java compiler decide the order in which to apply the individual operations?

In Java, as in most programming languages, that decision is dictated by a set of ordering rules chosen to conform to standard mathematical usage. They are called **rules of precedence**. For arithmetic expressions, the rules are:

1. The Java compiler first applies any unary minus operators (a minus sign with no operand to its left).
2. The compiler then applies the multiplicative operators (`*`, `/`, and `%`). If two of these operators apply to the same operand, the leftmost one is performed first.
3. It then applies the additive operators (`+` and `-`). Once again, if two operators at this level of precedence apply to the same operand, Java starts with the leftmost one.

Thus, in the expression

```
2 * x + 3 * y
```

the multiplication operations are performed first, even when the parentheses are missing. Using parentheses may make the order clearer, but in this case their use is not required because the intended order of operations matches the precedence assumptions of traditional mathematics. If you instead want the addition to be performed first, you must indicate that fact explicitly by using parentheses, as in

```
2 * (x + 3) * y
```

The rules of precedence apply only when two operators "compete" for a single operand. For instance, in the expression

```
2 * x + 3 * y
```

the operators `*` and `+` compete for the operand `x`. The rules of precedence dictate that the `*` is performed first because multiplication has higher precedence than addition. Similarly, looking at the two operators next to the value 3, you can again determine that the `*` is performed first, for precisely the same reason. In cases in which operators do not actually compete for the same operand, as is the case with the two multiplication operations in this expression, Java performs the computation in left-to-right order. Thus, Java will multiply `x` by 2 before multiplying `y` by 3.

Precedence rules can make a significant difference in the outcome of expression evaluation. Consider, for example, the expression

```
10 - 5 - 2
```

Because Java's precedence rules dictate that the leftmost − be performed first, the computation is carried out as if the expression had been written

```
(10 - 5) - 2
```

which yields the value 3. If you want the subtractions performed in the other order, you must use explicit parentheses:

```
10 - (5 - 2)
```

In this case, the result would be 7.

There are many situations in which parentheses are required to achieve the desired result. For example, suppose that, instead of adding two floating-point numbers the way **Add2Doubles** does, you wanted them averaged instead. The program is almost the same, as shown in Figure 3-3.

Note that the parentheses are necessary in the statement

```
double average = (n1 + n2) / 2;
```

to ensure that the addition is performed before the division. If the parentheses were missing, Java's precedence rules would dictate that the division be performed first, and the result would be the mathematical expression

$$n1 \ + \ \frac{n2}{2}$$

instead of the intended

$$\frac{n1 \ + \ n2}{2}$$

FIGURE 3-3   **Program to average two double-precision numbers**

```java
/*
 * File: Average2Doubles.java
 * ---------------------
 * This program averages two double-precision floating-point numbers.
 */

import acm.program.*;

public class Average2Doubles extends ConsoleProgram {

   public void run() {
      println("This program averages two numbers.");
      double n1 = readDouble("Enter n1: ");
      double n2 = readDouble("Enter n2: ");
      double average = (n1 + n2) / 2;
      println("The average is " + average + ".");
   }

}
```

Applying precedence rules

To illustrate precedence rules in action, let's consider the expression

```
8 * (7 – 6 + 5) % (4 + 3 / 2) – 1
```

Put yourself in the place of the computer. How would you go about evaluating this expression?

Your first step is to evaluate the parenthesized subexpressions, which happens in left-to-right order. To compute the value of `(7 – 6 + 5)`, you subtract 6 from 7 to get 1, and then add 5 to get 6. Thus, after evaluating the first subexpression, you are left with

```
8 *  6  % (4 + 3 / 2) – 1
```

where the box indicates that the value is the result of a previously evaluated subexpression.

You can then go on to evaluate the second parenthesized subexpression. Here, you must do the division first, since division and multiplication take precedence over addition. Thus, your first step is to divide 3 by 2, which results in the value 1 (remember that integer division throws away the remainder). You then add the 4 and 1 to get 5. At this point, you are left with the following expression:

```
8 *  6  %  5  – 1
```

From here, Java's precedence rules dictate that you perform the multiplication and remainder operations, in that order, before the subtraction: 6 times 8 is 48, and the remainder of 48 divided by 5 is 3. Your last step is to subtract 1, leaving 2 as the value of the complete expression.

Type conversion

You have already learned that it is possible to combine values of different numeric types within a Java program. When you do so, Java handles the situation by using **automatic type conversion**, a process by which values of one type are converted into another compatible type as an implicit part of the computation process. For example, whenever an integer and a floating-point value are combined using an arithmetic operator, the integer is automatically converted into the mathematically equivalent **double** before the operation is applied. Thus, if you write the expression

```
1 + 2.3
```

the integer 1 is converted internally into the floating-point number 1.0 before the addition is performed.

In Java, automatic type conversions are also performed whenever a variable is given a value of a more restrictive type. For example, if you write the declaration

```
double total = 0;
```

the integer 0 is converted into a **double** before it is assigned as the initial value of the variable **total**. Some programming languages (and some programmers) insist on writing this statement as

```
double total = 0.0;
```

In terms of its effect, this statement has the same meaning as the earlier one. On the other hand, the values 0 and 0.0 mean different things to mathematicians. Writing the value 0 indicates that the value is precisely 0, because integers are exact. When 0.0 appears in a statistical or mathematical context, however, the usual interpretation is that it represents a number close to zero, but one whose accuracy is known only to one significant digit after the decimal point. To avoid ambiguity, this text uses integers to indicate exactness, even in floating-point contexts.

It is, however, illegal in Java to use a value of type **double** as the initial value of a variable of type **int**. You can get around this restriction by using a syntactic construct called a **type cast**, which consists of the desired type in parentheses followed by the value you wish to convert. Thus, if you write the declaration

```
int n = (int) 1.9999;
```

Java will convert 1.9999 to an integer before assigning it as the initial value of **n**. You may, however, be surprised to learn that the integer it chooses is 1 rather than 2. Converting a value from floating-point to integer representation—which happens both with type casts and in integer division—simply throws away any fraction. That form of conversion is called **truncation.**

As an example of how truncation is useful, suppose that you have been asked to write a program that translates a metric distance in centimeters back into English units—the inverse of the **InchesToCentimeters.java** program in Figure 3-2. If all you need are the number of inches, the body of the program would look pretty much the same as before; the only difference in the computation is that you would divide by **CENTIMETERS_PER_INCH** instead of multiplying.

Suppose, however, that your employer wants you to display the answer not simply as the total number of inches, but as an integral number of feet plus the number of leftover inches. To compute the whole number of feet, you can divide the total number of inches by 12 and throw away any remainder. To calculate the number of inches left over, you can multiply the number of feet by 12 and subtract that quantity from the total number of inches. The entire program is shown in Figure 3-4.

The declaration

```
int feet = (int) (totalInches / INCHES_PER_FOOT);
```

throws away the remainder because the result is cast to be an **int**. The parentheses here are important because the type cast operator has very high precedence.

There are also cases in which you need to specify a type conversion even though the rules for automatic conversion do not apply. Suppose, for example, you have declared two integer variables, **num** and **den**, and you want to compute their mathematical quotient (including the fraction) and assign the result to a newly declared **double** variable named **quotient**. You can't simply write

```
double quotient = num / den;
```

because both **num** and **den** are integers. When the division operator is applied to two integers, it throws away the fraction. To avoid this problem, you have to convert at least

---

FIGURE 3-4 **Program to convert centimeters into feet and inches**

```
/*
 * File: CentimetersToFeetAndInches.java
 * ----------------------------------
 * This program converts centimeters to an integral number of feet
 * and any remaining inches.
 */

import acm.program.*;

public class CentimetersToFeetAndInches extends ConsoleProgram {

    public void run() {
        println("This program converts centimeters to feet and inches.");
        double cm = readDouble("Enter value in centimeters: ");
        double totalInches = cm / CENTIMETERS_PER_INCH;
        int feet = (int) (totalInches / INCHES_PER_FOOT);
        double inches = totalInches - INCHES_PER_FOOT * feet;
        println(cm + "cm = " + feet + "ft + " + inches + "in");
    }

    private static final int INCHES_PER_FOOT = 12;
    private static final double CENTIMETERS_PER_INCH = 2.54;
}
```

---

one of the values to `double` before the division is performed. There are several ways to do so. You could, for example, convert the denominator of the fraction by writing

```
double quotient = num / (double) den;
```

Since the denominator is now of type `double`, the division is carried out using floating-point arithmetic and the fraction is retained. Equivalently, you can convert the numerator by writing

```
double quotient = (double) num / den;
```

This statement has the same effect, but only because the precedence of a type cast is higher than that of division, which means that the type conversion is performed first.

3.4 Assignment statements

Variables in Java are given values in either of two ways. In the programs you have seen so far, the value of a variable has been set as part of the declaration. More sophisticated programs, however, often change the value of variables after they are declared, since one of the central advantages of variables is that their values can change over the lifetime of a program. To assign a new value to a variable, you need to use an **assignment statement,** using the syntax diagram shown on the right. The syntax is clearly very close to that used in declarations; the only difference is that the name of the type is missing.

Syntax for assignment statements:

>   *variable* **=** *expression***;**

where:
>   *variable* is the variable you wish to set
>   *expression* specifies the value

Diagramming the result of assignment statements

As noted in Chapter 2, drawing box diagrams can help you visualize the role of variables in a program. Whenever you declare a local variable as part of a method definition, you can draw a new box to hold its value and label the box with the variable name. For example, suppose that a method introduces three new variables—two of type `int` with the names `n1` and `n2`, and one of type `String` with the name `msg`—using the following declarations:

```
int n1 = 17;
int n2 = 0;
String msg = "Hello";
```

You can represent the variables in that method graphically by drawing a box for each variable, as follows.
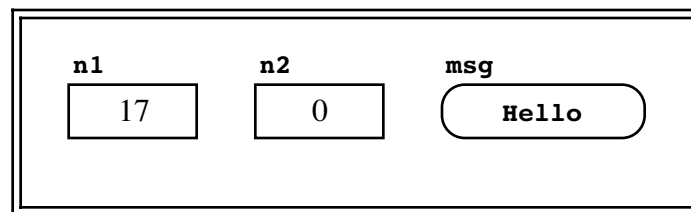
In this text, the double-line border surrounding all the variables is used to indicate that those variables are defined within the same method. In programming, that collection of variables associated with a particular invocation of a method is called a **stack frame,** for reasons that will be described in Chapter 6. I have also chosen to draw the box for the `String` variable with a different size and shape to emphasize that it holds values of a different type.

Assignment statements change the value of variables in the current stack frame. Thus, if your program executes the statement

```
n2 = 25;
```

you can represent this assignment in the diagram by writing 25 inside the box named `n2`:
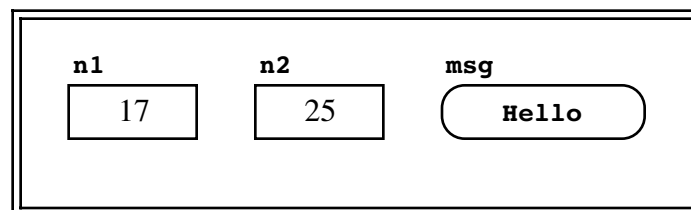
Similarly, you can indicate the effect of the statement

```
msg = "Welcome";
```

as follows:

Again, it is important to keep in mind that a variable can only hold a value of the appropriate type. If, for instance, you were to write the statement
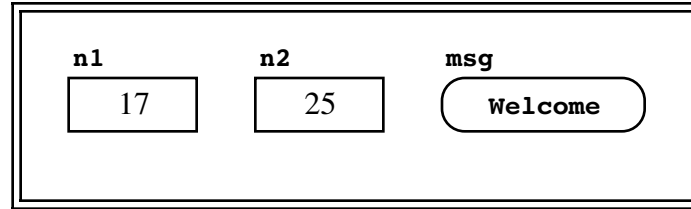
```
msg = 173;
```

in your program, the Java compiler would mark this statement as an error because the variable **msg** has been declared as a **String**.

The most important property illustrated by the diagram is that each variable holds precisely one value. Once you have assigned a value to a variable, the variable maintains that value until you assign it a new one. The value of one variable does not disappear if you assign its value to another variable. Thus the assignment

```
n2 = n1;
```

changes **n2** but leaves **n1** undisturbed:
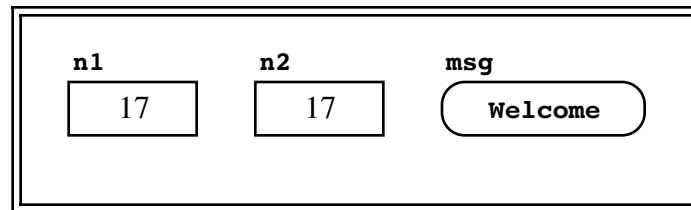
Assigning a new value to a variable erases its previous contents. Thus, the statement

```
msg = "Aloha!";
```

changes the picture to

The previous value of the variable **msg** is lost.

It is particularly important to recognize that assignment is an active operation and not a mathematical statement of equality. The mathematical equation

$$x = x + 1$$

is nonsensical. There are no values of *x* for which *x* is equal to *x* + 1. The corresponding assignment statement

```
x = x + 1;
```

has a perfectly legitimate—and extraordinarily useful—meaning. As in any assignment statement, the value of the expression on the right of the equal sign i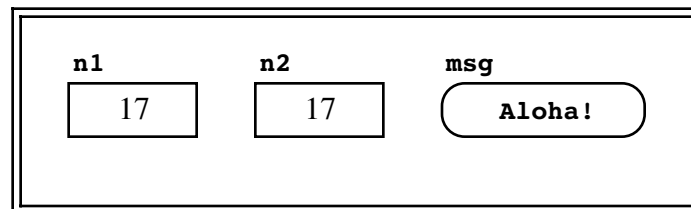s stored in the variable appearing on the left. The effect of this statement, therefore, is to replace the value of **x** with one more than its previous value or, in other words, to add 1 to **x**. This construct is so common in programming that Java—drawing on the structure of C before it—defines a special idiom for this purpose, as described in the following section.

3.5 Programming idioms and patterns

Before the invention of writing, history and religion passed from generation to generation as part of an oral tradition. *The Iliad* and *The Odyssey* of Homer, the Vedic literature of India, the Old Norse mythologies, the sermons and songs that kept African traditions alive through centuries of slavery—all are examples of an oral tradition. These works are characterized by the patterned repetition of phrases, which make it easier for singers, preachers, and storytellers to remember them. These repeated patterns, called **formulas**, provide the memory cues that make it possible to remember and make variations on a long and detailed story.

In its entirety, Java is itself a long and detailed story with many rules and techniques to remember. Even so, as you write your programs, you will notice that many formulaic structures come up repeatedly, just like the formulas in oral tradition. If you learn to recognize these formulas and think of them as conceptual units, you will soon discover that there is less to remember about programming in Java than you might have thought. In programming, such formulas are called **programming idioms** or **programming patterns.** To write programs effectively, you must learn how to apply these programming patterns to the task at hand. Eventually, you should be able to do so without devoting any conscious attention to the process. A general idea will come into your mind as part of a solution strategy, and you will automatically translate that idea into the appropriate pattern as you compose the program.

One of the most common patterns in Java consists of shorthand forms for certain frequently occurring assignment statements. In Java, you will find that you often change a variable by adding something to its current value, subtracting something from it, or some similar operation. For example, if you were writing a program to balance your checkbook, you might expect to use following assignment statement:

```
balance = balance + deposit;
```

This statement adds the value in the variable **deposit** to the current value of the variable **balance**, leaving the result in **balance**. In more colloquial English, the effect of this statement can be summarized as "add **deposit** to **balance**."

Although the statement

```
balance = balance + deposit;
```

has the desired effect, it is not the statement that a Java programmer would usually write. Statements that perform some operation on a variable and then store the result back in that same variable occur so frequently in programming that the designers of C—from

which Java's expression structure is derived—included an idiomatic shorthand for it. For any binary operator *op*, the statement

> *variable* **=** *variable op expression***;**

can be replaced by

> *variable op***=** *expression***;**

The combination of an operator with the **=** used for assignment form is called a **shorthand assignment operator**.

Using the shorthand assignment operator for addition, the more common form of the statement

```
balance = balance + deposit;
```

is therefore

```
balance += deposit;
```

which means, in English, "add `deposit` to `balance`."

Because this same shorthand applies to any binary operator in Java, you can subtract the value of `surcharge` from `balance` by writing

```
balance -= surcharge;
```

divide the value of **x** by 10 using

```
x /= 10;
```

or double the value of `salary` by using

```
salary *= 2;
```

Beyond the shorthand assignment operators, Java offers a further level of abbreviation for two particularly common programming operations—adding or subtracting 1 from a variable. Adding 1 to a variable is called **incrementing** that variable; subtracting 1 is called **decrementing** that variable. To indicate these operations in an extremely compact form, Java provides the operators **++** and **--**. For example, the statement

```
x++;
```

has the same ultimate effect as

```
x += 1;
```

which is itself short for

```
x = x + 1;
```

Similarly,

```
y--;
```

has the same effect as

```
y -= 1;
```

or

```
y = y - 1;
```

The **++** and **--** operators occur all the time in Java programs and are actually more complex in their operation than this presentation suggests. The form shown here, however, is sufficient for you to understand how these operators are used in the standard statement forms introduced in Chapter 4.

Summary

In this chapter, you have had the opportunity to learn about expressions in Java programs and how they work. Important points introduced in the chapter include:

- Data values come in many different types, each of which is defined by a *domain* and a *set of operations*.
- *Constants* are used to specify values that do not change within a program.
- *Variables* have three attributes: a name, a value, and a type. All variables used in a Java program must be *declared*, which establishes the name and type of the variable.
- You can use *class variables* to associate names with constants. Such declarations are written outside of any method and include the keywords **static** and **final**.
- Expressions are composed of individual *terms* connected by *operators*. The subexpressions to which an operator applies are called its *operands*.
- When an operator is applied to two operands of type **int**, the result is also of type **int**. If either or both operands are of type **double**, so is the result.
- If the **/** operator is applied to two integers, the result is the integer obtained by dividing the first operand by the second and then throwing the remainder away. The remainder can be obtained by using the **%** operator.
- The order of operations in an expression is determined by *rules of precedence*. The operators introduced so far fall into three precedence classes:

unary **–**	**(**type cast**)**	(highest)
* **/**	**%**	
+ **–**		(lowest)

  For the binary operators introduced so far, whenever two operators from the same precedence class compete for the same operand, those operators are applied in left-to-right order.
- Automatic conversion between numeric types occurs when values of different types are combined in an expression or when an assignment is performed to a variable of a more general type.
- Explicit conversion between numeric types can be indicated by using a type cast.

- Variables are given values through the use of *assignment statements*. Each variable can hold only one value at a time; when a variable is assigned a new value, any previous value is lost.

- Java includes a shorthand form of the assignment statement in which the statement

    variable **=** *variable op expression***;**

  can be replaced by

    *variable op***=** *expression***;**

- Java includes special operators **++** and **--**, which specify adding and subtracting 1 from a variable, respectively.

Review questions

1. What are the two attributes that define a data type?

2. Identify which of the following are legal constants in Java. For the ones that are legal, indicate whether they are integers or floating-point constants.

    a) **42**   g) **1,000,000**
    b) **-17**   h) **3.1415926**
    c) **2+3**   i) **123456789**
    d) **-2.3**   j) **0.000001**
    e) **20**   k) **1.1E+11**
    f) **2.0**   l) **1.1X+11**

3. Rewrite the following floating-point constants in Java's form for scientific notation:

    a) 6.02252×10^{23}
    b) **29979250000.0**
    c) **0.00000000529167**
    d) **3.1415926535**

    (By the way, each of these constants represents an approximation of an important value from chemistry, physics, or mathematics: (a) Avogadro's number, (b) the speed of light in centimeters per second, (c) the Bohr radius in centimeters, and (d) the mathematical constant π. In the case of π, there is no advantage in using the scientific notation form, but it is nonetheless possible and you should know how to do so.)

4. Indicate which of the following are legal variable names in Java:

    a) **x**   g) **total output**
    b) **formula1**   h) **aReasonablyLongVariableName**
    c) **average_rainfall**   i) **12MonthTotal**
    d) **%correct**   j) **marginal-cost**
    e) **short**   k) **b4hand**
    f) **tiny**   l) **_stk_depth**

5. Indicate the values and types of the following expressions:

    a) **2 + 3**   d) **3 * 6.0**
    b) **19 / 5**   e) **19 % 5**
    c) **19.0 / 5**   f) **2 % 7**

6. What is the difference between the unary minus operator and the binary subtraction operator?

7. By applying the appropriate precedence rules, calculate the result of each of the following expressions:

    a) **6 + 5 / 4 – 3**
    b) **2 + 2 * (2 * 2 – 2) % 2 / 2**
    c) **10 + 9 * ((8 + 7) % 6) + 5 * 4 % 3 * 2 + 1**
    d) **1 + 2 + (3 + 4) * ((5 * 6 % 7 * 8) – 9) – 10**

8. If the variable **k** is declared to be of type **int**, what value does **k** contain after the program executes the assignment statement

    ```
    k = (int) 3.14159;
    ```

    What value would **k** contain after the assignment statement
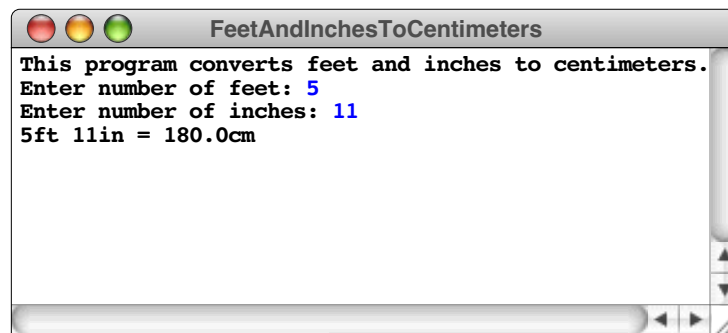
    ```
    k = (int) 2.71828;
    ```

9. In Java, how do you specify conversion between numeric types?

10. What idiom would you use to multiply the value of the variable **cellCount** by 2?

11. What is the most common way in Java to write a statement that has the same effect as the statement

    ```
    x = x + 1;
    ```
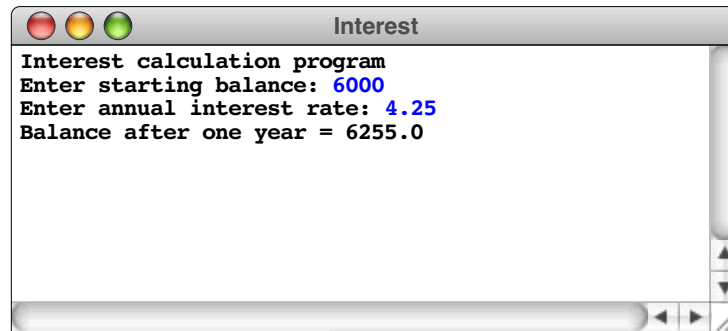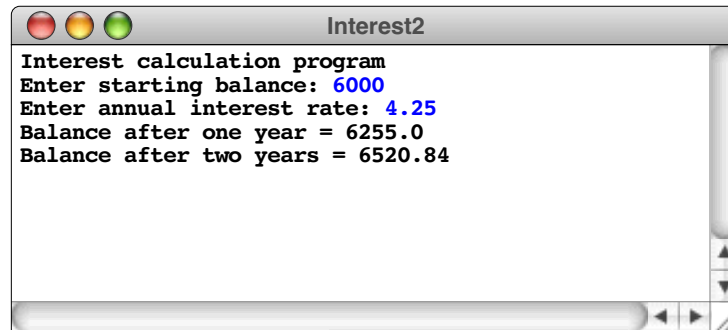
Programming exercises

1. Extend the **InchesToCentimeters** program given in Figure 3-2 so that it reads in two input values: the number of feet, followed on a separate line by the number of inches. Here is a sample run of the program:


```
This program converts feet and inches to centimeters.
Enter number of feet: 5
Enter number of inches: 11
5ft 11in = 180.0cm
```

2. Write a program that reads in two numbers: an account balance and an annual
    interest rate expressed as a percentage. Your program should then display the new
    balance after a year. There are no deposits or withdrawals—just the interest payment.
    Your program should be able to reproduce the following sample run:

```
●●●                        Interest
Interest calculation program
Enter starting balance: 6000
Enter annual interest rate: 4.25
Balance after one year = 6255.0
```

3. Extend the program you wrote in Exercise 2 so that it also displays the balance after
    two years have elapsed, as shown in the following sample run:

```
●●●                        Interest2
Interest calculation program
Enter starting balance: 6000
Enter annual interest rate: 4.25
Balance after one year = 6255.0
Balance after two years = 6520.84
```

Note that the interest used in this example is compounded annually, which means the
interest from the first year is added back to the bank balance and is therefore itself
subject to interest in the second year. In the first year, the $6,000 earns 4.25%
interest, or $255. In the second year, the account earns 4.25% interest on the entire
$6,255.

4. Write a program that asks the user for the radius of a circle and then computes the
    area of that circle (*A*) using the formula

$$A = \pi r^2$$

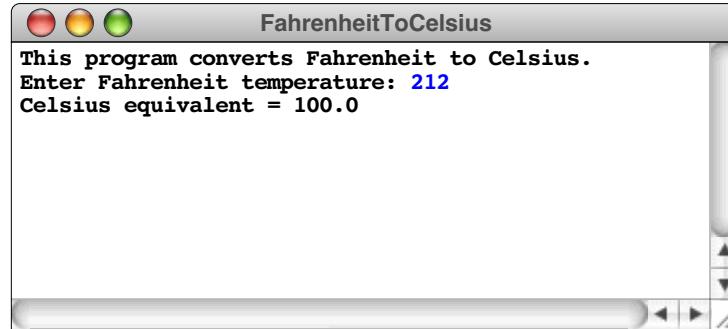Note that there is no "raise to a power" operator in Java. Given the arithmetic
operators you know Java has, how can you write an expression that achieves the
desired result?

5. Write a program that reads in a temperature in degrees Fahrenheit and returns the
    corresponding temperature in degrees Celsius. The conversion formula is

$$C = \frac{5}{9}(F - 32)$$

The following is a sample run of the program:

```
┌──────────────────────────────────────────────────────────┐
│ ● ● ●            FahrenheitToCelsius                       │
│ This program converts Fahrenheit to Celsius.              │
│ Enter Fahrenheit temperature: 212                          │
│ Celsius equivalent = 100.0                                 │
│                                                            │
│                                                            │
│                                                            │
│                                                            │
│                                                         ▲  │
│                                                         ▼  │
│ ◀ ▶                                                        │
└──────────────────────────────────────────────────────────┘
```

If you write this program carelessly, the answer always comes out 0. What bug causes this behavior?

6. In Norton Juster's children's story *The Phantom Tollbooth*, the Mathemagician gives Milo the following problem to solve:

    4 + 9 − 2 * 16 + 1 / 3 * 6 − 67 + 8 * 2 − 3 + 26 − 1 / 34 + 3 / 7 + 2 − 5

    According to Milo's calculations, which are corroborated by the Mathemagician, this expression "all works out to zero." If you do the calculation, however, the expression comes out to zero only if you start at the beginning and apply all the operators in strict left-to-right order. What would the answer be if the Mathemagician's expression were evaluated using Java's precedence rules? Write a program to verify your calculation.

7. Write a program that converts a metric weight in kilograms to the corresponding English weight in pounds and ounces. The conversion factors you need are

    1 kilogram = 2.2 pounds
    1 pound = 16 ounces

8. Write a program that computes the average of four integers.

9. There's an old nursery rhyme that goes like this:

    As I was going to St. Ives,
    I met a man with seven wives,
    Each wife had seven sacks,
    Each sack had seven cats,
    Each cat had seven kits:
    Kits, cats, sacks, and wives,
   How many were going to St. Ives?

    The last line turns out to be a trick question: only the speaker is going *to* St. Ives; everyone else is presumably heading in the opposite direction. Suppose, however, that you want to find out how many representatives of the assembled multitude—kits, cats, sacks, and wives—were coming *from* St. Ives. Write a Java program to calculate and display this result. Try to make your program follow the structure of the problem so that anyone reading your program would understand what value it is calculating.

Chapter 4
Statement Forms

The statements was interesting but tough.
— Mark Twain, *Adventures of Huckleberry Finn*, 1884
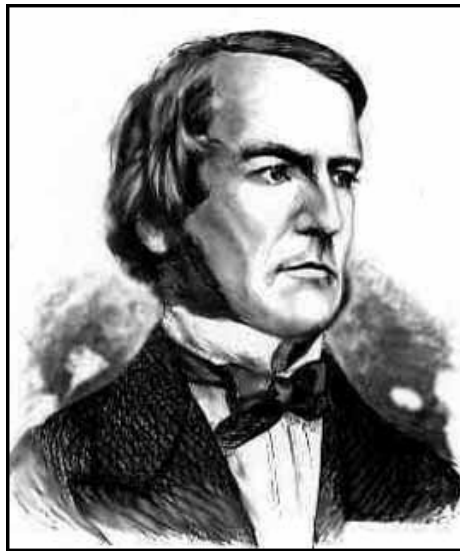
George Boole (1815-1864)

Even though he was largely self-taught and never earned a formal university degree, George Boole achieved sufficient prominence to be appointed Professor of Mathematics at Queen's College in County Cork, Ireland, and to be elected as a Fellow of the Royal Society. His most influential work was an 1854 book entitled *An Investigation into the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probabilities*. That book introduced a system of logic that has come to be known as *Boolean algebra* and which serves as the foundation for the control operations implemented in modern programming languages.

In Chapter 2, you saw several examples of Java programs that extend the **Program** hierarchy. Such programs operate by executing the statements contained within the body of a method called **run**. This chapter covers the different statement types available in Java and, in the process, extends the set of tools you have for solving problems.

As in most programming languages, statements in Java fall into one of two principal classifications: **simple statements**, which perform some action, and **control statements**, which affect the way in which other statements are executed. You have already seen a variety of simple statements in Java, such as assignments and calls to the **println** method. You have also encountered various control statements in your work with Karel. The **for** statement makes it possible to repea65 number of times, the **while** statement allows you to specify repetition until some condition occurs, and the **if** statement makes it possible to choose different paths through a program depending on some conditional test. Up to now, however, you have studied these statements in an informal, idiomatic way. To use the full power of these statements, you need a more detailed understanding of how each type of statement works and how it can be applied as part of your problem-solving repertoire.

4.1 Simple statements

In the programs in Chapters 2 and 3, you saw simple statements used to accomplish a variety of tasks. For example, the very first program in the text was

```
add(new GLabel("hello, world"), 100, 75);
```

which was a call to the **add** method in **GraphicsProgram**. Similarly, you have been introduced to assignment statements, such as the shorthand form

```
balance += deposit;
```

and statements that display information, such as

```
println("The total is " + total + ".");
```

Informally, it makes sense to think of these statement types as separate tools and to use them idiomatically. If you need to read an integer, all you need to do is remember that there is an idiom for that purpose, which you can then write down. If you need to display a value, you know that you should use the **println** method. Viewed formally, however, these simple statements all have a unified structure that makes it easy for the Java compiler to recognize a legal statement in a program. In Java, all simple statements—regardless of their function—fit the following rule:

> ### Simple Statement Rule
>
> A simple statement consists of an expression followed by a semicolon.

Thus, the template for a simple statement is simply this:

> *expression***;**

Adding the semicolon after the expression turns the expression into a legal statement form.

Even though any expression followed by a semicolon is a *legal* statement in Java, it is not true that every such combination represents a *useful* statement. To be useful, a statement must have some discernible effect. The statement

```
n1 + n2;
```

consists of the expression `n1 + n2` followed by a semicolon and is therefore a legal statement. It is, however, an entirely useless one because nothing is ever done with the answer; the statement adds the variables `n1` and `n2` together and then throws the result away. Simple statements in Java are typically assignments (including the shorthand assignments and increment/decrement operators) or calls to methods, such as `println`, that perform some useful operation.

It is easy to see that program lines such as

```
println("The total is " + total + ".");
```

are legal statements according to the Simple Statement Rule. In the definition of expression given in Chapter 2, method calls are legal expressions, so that the method call part of the above line—everything except the semicolon—is a legal expression. Putting the semicolon at the end of the line turns that expression into a simple statement.

But what about assignments? If a line like

```
total = 0;
```

is to fit the Simple Statement Rule, it must be the case that

```
total = 0
```

is itself an expression.

In Java, the equal sign used for assignment is simply a binary operator, just like `+` or `/`. The `=` operator takes two operands, one on the left and one on the right. For our present purposes, the left operand must be a variable name. When the assignment operator is executed, the expression on the right-hand side is evaluated, and the resulting value is then stored in the variable that appears on the left-hand side. Because the equal sign used for assignment is an operator,

```
total = 0
```

is indeed an expression, and the line

```
total = 0;
```

is therefore a simple statement.

Blocks

Simple statements allow programmers to specify actions. Except for the `HelloProgram` example in Chapter 2, however, every program you have seen so far requires more than one simple statement to do the job. For most programs, the solution strategy requires a coordinated action consisting of several sequential steps. The `Add2Integers` program, for example, had to first get one number, then get a second, then add the two together,

and finally display the result. Translating this sequence of actions into actual program steps required the use of several individual statements that all became part of the main program body.

To specify that a sequence of statements is part of a coherent unit, you can assemble those statements into a **block**, which is a collection of statements and declarations enclosed in curly braces, as follows:

```
{
    statement₁
    statement₂
    statement₃
      . . .
    statementₙ
}
```

Any of these statements can be replaced by a declaration, and it is probably easiest to think of declarations as simply another form of statement.

You have already seen blocks in several of the programming examples from the previous chapters. The body of each method is a block, as are the bodies of the control statements introduced later in this chapter.

The statements in the interior of a block are usually indented relative to the enclosing context. The compiler ignores the indentation, but the visual effect is helpful to the human reader, because it makes the structure of the program jump out at you from the format of the page. Empirical research has shown that using either three or four spaces at each new level makes the program structure easiest to see; the programs in this text use four spaces for each new level. Indentation is critical to good programming, so you should strive to develop a consistent indentation style in your programs.

When the Java compiler encounters a block, it treats the entire block as a single statement. Thus, whenever the notation *statement* appears in a syntax template or an idiomatic pattern, you can substitute for it either a single statement or a block. To emphasize that they are statements as far as the compiler is concerned, blocks are sometimes referred to as **compound statements**.

4.2 Control statements

In the absence of any directives to the contrary, statements in a Java program are executed one at a time in the order in which they appear. For most applications, however, this strictly top-to-bottom ordering is not sufficient. Solution strategies for real-world problems tend to involve such operations as repeating a set of steps or choosing between alternative sets of actions. Statements that affect the way in which other statements are executed are called *control statements*.

Control statements in Java fall into two basic classes:

1. *Conditionals*. In solving problems, you will often need to choose between two or more independent paths in a program, depending on the result of some conditional test. For example, you might be asked to write a program that behaves one way if a certain value is negative and some different way otherwise. The type of control statement needed to make decisions is called a **conditional**. In Java, there are two conditional statement forms: the `if` statement and the `switch` statement.

2. *Iteration*. Particularly as you start to work with problems that involve more than a few data items, your programs will often need to repeat an operation a specified number of times or as long as a certain condition holds. In programming, such repetition is called **iteration** and the portion of code that repeats is called a **loop.** In Java, the control statements used as the basis for iteration are the **while** statement and the **for** statement.

Each control statement in Java consists of two parts: the **control line,** which specifies the nature of the repetition or condition, and the **body,** which consists of the statements that are affected by the control line. In the case of conditional statements, the body may be divided into separate parts, where one set of statements is executed in certain cases and another set of statements is executed in others.

The body of each control statement consists of other statements. The effect of the control statement itself—no matter whether it specifies repetition or conditional execution—is applied to each of the statements in the body. Those statements, moreover, can be of any type. They may be simple statements, they may be compound statements, or they may themselves be control statements, which in turn contain other statements. When a control statement is used within the body of another control statement, it is said to be **nested**. The ability to nest control statements, one inside another, is one of the most important characteristics of modern programming languages.

4.3 Boolean data

In the course of solving a problem, it is often necessary to have the program test a particular condition that affects the subsequent behavior of the program. The **if** statement, along with many of the other facilities that control the execution of a program, use expressions whose values are either true or false. This type of data—for which the only values in the domain are **true** and **false**—is called **Boolean data**, after the mathematician George Boole, who developed an algebraic approach for working with such values.

Java defines several operators that work with **boolean** values, which is one of the primitive types introduced in Chapter 3. These operators fall into two major classes— relational operators and logical operators—which are discussed in the next two sections.

Relational operators

The **relational operators** are used to compare two values. Java defines six relational operators, which actually fall into two precedence classes. The operators that test the ordering relationship between two quantities are

> **>**   Greater than
> **<**   Less than
> **>=**   Greater than or equal to
> **<=**   Less than or equal to

These operators appear in the precedence hierarchy below the arithmetic operators, and are in turn followed by the operators

> **==**   Equal
> **!=**   Not equal

which test for equality and inequality.

When you write programs that test for equality, be very careful to use the **==** operator, which is composed of two equal signs. A single equal sign is the assignment operator. Since the double equal sign is not part of standard mathematics, replacing it with a single equal sign is a particularly common mistake. Fortunately, the Java compiler usually catches the error when it compiles your program, because the assignment is usually illegal in the context in which it appears.

> When writing programs that test for equality, be sure to use the **==** operator and not the single **=** operator, which signifies assignment.

The relational operators can only be used to compare **atomic data** values—data values that are not built up from smaller component parts. For example, integers, floating-point numbers, Boolean values, and characters constitute atomic data because they cannot be decomposed into smaller pieces. Strings, on the other hand, are not atomic because they are composed of individual characters. Thus, you can use relational operators to compare two values of the types **int**, **double**, **char**, or even **boolean** itself, but you cannot use them to compare two values of type **String**. You will learn how to compare strings in Chapter 8.

Logical operators

In addition to the relational operators, which take atomic values of any type and produce Boolean results, Java defines three operators that take Boolean operands and combine them to form other Boolean values:

!	Logical not (**true** if the following operand is **false**)
&&	Logical and (**true** if both operands are **true**)
||	Logical or (**true** if either or both operands are **true**)

These operators are called **logical operators** and are listed in decreasing order of precedence.

The operators **&&**, **||**, and **!** closely resemble the English words *and*, *or*, and *not*. Even so, it is important to remember that English can be somewhat imprecise when it comes to logic. To avoid that imprecision, it is often helpful to think of these operators in a more formal, mathematical way. Logicians define these operators using **truth tables**, which show how the value of a Boolean expression changes as the values of its operands change. For example, the truth table for the **&&** operator, given Boolean values **p** and **q**, is

p	q	p && q
false	false	false
false	true	false
true	false	false
true	true	true

The last column of the table indicates the value of the Boolean expression **p && q** given individual values of the Boolean variables **p** and **q** shown in the first two columns. Thus, the first line in the truth table shows that when **p** is **false** and **q** is **false**, the value of the expression **p && q** is also **false**.

The truth table for **||** is

p	q	p || q
false	false	false
false	true	true
true	false	true
true	true	true

Note that the `||` operator does not indicate *one or the other*, as it often does in English, but instead indicates *either or both*, which is its mathematical meaning.

The `!` operator has the following simple truth table:

p	!p
false	true
true	false

If you need to determine how a more complex logical expression operates, you can break it down into these primitive operations and build up a truth table for the individual pieces of the expression.

In most cases, logical expressions are not so complicated that you need a truth table to figure them out. The only common case that seems to cause confusion is when the `!` or `!=` operator comes up in conjunction with `&&` or `||`. When talking about situations that are not true (as is the case when working with the `!` and `!=` operators), conventional English is sometimes at odds with mathematical logic, and you should use some extra care to avoid errors.

> Be careful when using the `&&` and `||` operators with relational tests that involve the `!` and `!=` operators. English can be somewhat fuzzy in its approach to logic, but programming requires you to be precise.

For example, suppose you wanted to express the idea "x is not equal to either 2 or 3" as part of a program. Just reading from the English version of this conditional test, new programmers are very likely to write

```
if (x != 2 || x != 3) . . .
```

If you look at this conditional test from the mathematical point of view, you can see that the expression within the `if` test is `true` if either (a) `x` is not equal to 2 or (b) `x` is not equal to 3. No matter what value `x` has, one of the statements must be `true`, since, if `x` is 2, it cannot also be equal to 3, and vice versa. Thus, the `if` test as written above would always succeed.

To fix this problem, you need to refine your understanding of the English expression so that it states the condition more precisely. That is, you want the test in the `if` statement to succeed whenever "it is not the case that either `x` is 2 or `x` is 3." You could translate this statement directly to Java by writing

```
if (!(x == 2 || x == 3)) . . .
```

but the resulting statement is a bit ungainly. The question you really want to ask is whether *both* of the following conditions are `true`:

- `x` is not equal to 2, *and*
- `x` is not equal to 3.

If you think about the question in this form, you could write the test as

```
if (x != 2 && x != 3) . . .
```

This simplification is a specific illustration of the following more general relationship from mathematical logic:

> !(*p* || *q*)   *is equivalent to*   !*p* && !*q*

for any logical expressions *p* and *q*. This transformation rule and its symmetric counterpart

> !(*p* && *q*)   *is equivalent to*   !*p* || !*q*

are called **De Morgan's laws**. Forgetting to apply these rules and relying instead on the English style of logic is a common source of programming errors.

Another common mistake comes from forgetting to use the appropriate logical connective when combining several relational tests. In mathematics, one often sees an expression of the form

> 0 < *x* < 10

> To test whether a number is in a particular range, it is not sufficient to combine relational operators, as is conventional in mathematics. The two parts of the condition must be written explicitly using **&&**, as in
>
> **(0 < x) && (x < 10)**

While this expression makes sense in mathematics, it is not meaningful in Java. In order to test that the variable **x** is both greater than 0 and less than 10, you need to indicate both conditions explicitly, as follows:

```
0 < x && x < 10
```

Short-circuit evaluation

Java interprets the **&&** and **||** operators in a way that differs from the interpretation used in many other programming languages. In Pascal, for example, evaluating these operators (which are written as AND and OR) requires evaluating both halves of the condition, even when the result can be determined halfway through the process. The designers of Java took a different approach that is often more convenient for programmers.

Whenever a Java program evaluates any expression of the form

> *exp*₁ **&&** *exp*₂

or

> *exp*₁ **||** *exp*₂

the individual subexpressions are always evaluated from left to right, and evaluation ends as soon as the answer can be determined. For example, if *exp₁* is **false** in the expression involving **&&**, there is no need to evaluate *exp₂* since the final answer will always be **false**. Similarly, in the example using **||**, there is no need to evaluate the second operand if the first operand is **true**. This style of evaluation, which stops as soon as the answer is known, is called **short-circuit evaluation.**

A primary advantage of short-circuit evaluation is that it allows one condition to control the execution of a second one. In many situations, the second part of a compound condition is meaningful only if the first part comes out a certain way. For example, suppose you want to express the combined condition that (1) the value of the integer **x** is nonzero and (2) **x** divides evenly into **y**. You can express this conditional test in Java as

```
(x != 0) && (y % x == 0)
```

because the expression **y % x** is evaluated only if x is nonzero. The corresponding expression in Pascal fails to generate the desired result, because both parts of the Pascal condition will always be evaluated. Thus, if **x** is 0, a Pascal program containing this expression will end up dividing by 0 even though it appears to have a conditional test to check for that case. Conditions that protect against evaluation errors in subsequent parts of a compound condition, such as the conditional test

```
(x != 0)
```

in the preceding example, are called **guards**.

Flags

Variables of type **boolean** are so important that they have a special name: **flags**. For example, if you declare a Boolean variable using the declaration

```
boolean done;
```

the variable **done** becomes a flag, which you can use in your program to record whether or not you are finished with some phase of the operation. You can assign new values to flags just as you can to any other variable. For example, you can write

```
done = true;
```

or

```
done = false;
```

More importantly, you can assign any expression that has a Boolean value to a Boolean variable. For example, suppose the logic of your program indicates that you are finished with some phase of the operation as soon as the value of the variable **itemsRemaining** becomes 0. To set **done** to the appropriate value, you can simply write

```
done = (itemsRemaining == 0);
```

The parentheses in this expression are not necessary but are often used to emphasize the fact that you are assigning the result of a conditional test to a variable. The statement above says, "Calculate the value of (**itemsRemaining == 0**), which will be either **true** or **false**, and store that result in the variable **done**."

Avoiding redundancy in Boolean expressions

Even though the statement

```
done = (itemsRemaining == 0);
```

is sufficient to store the correct Boolean value in the variable **done**, this type of statement seems difficult for people to learn. New programmers have a tendency to achieve the same effect with the following **if** statement:

```
if (itemsRemaining == 0) {
    done = true;
} else {
    done = false;
}
```

Although these lines have the desired effect, they do not have the efficiency or the elegance you should seek to achieve in your programs. The second version requires five lines to do the work of one and will make your programs much longer than they need to be. As you work with Boolean data, it is important to remember that you can assign Boolean values just like any other values and that explicit tests are not necessary.

Be careful to avoid redundancy when using Boolean data. Standard warning signs include comparing a Boolean value against the constant **true** and the use of an **if** statement to produce a Boolean result that was already available as a conditional expression.

A similar problem occurs when you use a flag as part of a conditional test. To test whether **done** has the value **true**, an experienced programmer writes

```
if (done) . . .
```

and not

```
if (done == true) . . .
```

Even though this second expression also works, the equality test is redundant. The value of **done** is already guaranteed to be either **true** or **false**, which is precisely the sort of value the **if** statement wants. You don't need to ask whether **done** is equal to **true**, since the extra test provides no new information.

An example of Boolean calculation

As astronomers have known for centuries, the earth takes a little more than 365 days to make a complete revolution around the sun. Because it takes about a quarter of a day more than 365 days for it to complete its annual cycle, an extra day builds up every four years, which must then be added to the calendar, creating a leap year. This adjustment helps keep the calendar in sync with the sun's orbit, but it is still off by a slight amount. To ensure that the beginning of the year does not slowly drift through the seasons, the actual rule used for leap years is slightly more complicated. Leap years come every four years, except for years ending in 00, which are leap years only if they are divisible by 400. Thus, 1900 was not a leap year even though 1900 is divisible by 4. The year 2000, on the other hand, was a leap year because 2000 is divisible by 400.

Suppose you have been asked to write a program that reads in a year and determines whether that year is a leap year. How would you write the Boolean expression necessary to answer that question? In order to be a leap year, one of the following conditions must hold:

- The year is divisible by 4 but not divisible by 100, *or*
- The year is divisible by 400.

If the year is contained in the variable **y**, the following Boolean expression has the correct result:

```
((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0)
```

Given Java's rules of precedence, none of the parentheses in this expression are actually required, but using parentheses makes long Boolean expressions easier to read. If you take the result of this expression and store it in a flag called **isLeapYear**, you can then test the flag at other points in the program to determine whether the **isLeapYear** condition is true. A program that performs the leap-year calculation is shown in Figure 4-1.

4.4 The **if** statement

The simplest way to express conditional execution in Java is by using the **if** statement, which comes in two forms:

> **if** (*condition*) *statement*

> **if** (*condition*) *statement* **else** *statement*

The *condition* component of this template is a Boolean-valued expression. The statements can be either simple statements or blocks.

FIGURE 4-1   **Program to determine whether a year is a leap year**

```java
/*
 * File: LeapYear.java
 * -------------------
 * This program reads in a year and determines whether it is a
 * leap year.  A year is a leap year if it is divisible by four,
 * unless it is divisible by 100.  Years divisible by 100 are
 * leap years only if divisible by 400.
 */

import acm.program.*;

public class LeapYear extends ConsoleProgram {
   public void run() {
      println("This program checks for leap years.");
      int year = readInt("Enter year: ");
      boolean isLeapYear = ((year % 4 == 0) && (year % 100 != 0))
                           || (year % 400 == 0);
      if (isLeapYear) {
         println(year + " is a leap year.");
      } else {
         println(year + " is not a leap year.");
      }
   }
}
```

You use the first form of the **if** statement when your solution strategy calls for a set of statements to be executed only if a particular condition applies. If that condition does not apply, the statements that form the body of the **if** statement are simply skipped. You use the second form of the **if** statement for situations in which the program must choose between two independent sets of actions based on the result of a test. The decision is always based on the dynamics of the problem. If the English description of the problem might logically contain the word *otherwise,* there is a good chance that you'll need the **else** form. If the English description conveys no such notion, then the simple **if** statement is probably sufficient.

The use of the **if** statement with the **else** form is illustrated by the **LeapYear** program given in Figure 4-1, in which the program needs to print one message if the specified year is a leap year and a different message otherwise. If the problem were instead structured so that a message were printed only in leap years, you would use the simple **if** form instead, as follows:

```
if (isLeapYear) {
   println(year + " is a leap year.");
}
```

In an **if** statement, the block of statements executed when the conditional expression is **true** is called the **then clause** of the **if** statement. The block of statements executed when the condition is **false**—if one is specified—is called the **else clause**.

The fact that the **else** clause is optional in the **if** statement sometimes creates an ambiguity, which is called the **dangling-else problem**. If you write several **if** statements nested one within another, some of which have **else** clauses and some of which don't, it can be difficult to tell which **else** goes with which **if**. When faced with this situation, the Java compiler follows the simple rule that each **else** clause is paired with the most recent **if** statement that does not already have an **else** clause. While this rule is simple for the compiler, it can still be hard for human readers to recognize quickly where each **else** clause belongs. By adopting a more disciplined programming style than Java requires, it is possible to get rid of dangling-else ambiguities. The following rule governing how to use blocks within **if** statements eliminates the problem:

If/Else Blocking Rule

For any **if** statement that (1) requires more than a single line or (2) requires an **else** clause, always use curly braces to enclose in a separate block the statements under the control of the **if** statement.

Because this text follows the If/Else Blocking Rule, the **if** statement appears only in one of the following four forms:

1. A single-line **if** statement used for extremely short conditions
2. A multiline **if** statement in which the statements are enclosed in a block
3. An **if-else** statement that *always* uses blocks to enclose the statements controlled by the **if** statement, even if they consist of a single statement
4. A *cascading* **if** statement, used for expressing a series of conditional tests

Each of these forms is discussed in more detail in the sections that follow.

Single-line `if` statements

The simple one-line format shown in the syntax box to the right is used only for those `if` statements in which there is no `else` clause and in which the body is a single statement short enough to fit on the same line as the `if`. In this type of situation, using braces and extending the `if` statement from one to three lines would make the program longer and more difficult to read.

Multiline `if` statements

Whenever the body of an `if` statement consists of multiple statements or a single statement that is too long for a single line, the statements are enclosed in a block, as shown in the syntax box. In this form, the statements are executed if the condition is `true`. If the condition is `false`, the program takes no action at all and continues with the statement following the `if`.

The `if-else` statement

To avoid the dangling-else problem, the bodies of `if` statements that have `else` clauses are always enclosed within blocks, as shown in the syntax box to the right. Technically, the curly braces that surround the block are necessary only if there is more than one statement governed by that condition. By systematically using those braces, however, you can minimize the possibility of confusion and make your programs easier to maintain.

Cascading `if` statements

The syntax box on the right illustrates an important special case of the `if` statement that is useful for applications in which the number of possible cases is larger than two. The characteristic form is that the `else` part of a condition consists of yet another test to check for an alternative condition. Such statements are called **cascading if statements** and may involve any number of `else if` lines. For example, the program `SignTest` in Figure 4-2 uses the cascading `if` statement to report whether a number is positive, zero, or negative. Note that there is no need to check explicitly for the `n < 0` condition. If the program reaches that last `else` clause, there is no other possibility, since the earlier tests have eliminated the positive and zero cases.

Syntax for single-line `if` statements:

```
if (condition) statement;
```

where:
  condition is the Boolean value being tested
  statement is a single statement to be
      executed if *condition* is `true`

Syntax for multiline `if` statements:

```
if (condition) {
    statements;
}
```

where:
  condition is the Boolean value being tested
  statements is a block of statements to be
      executed if the condition is `true`

Syntax for `if-else` statements:

```
if (condition) {
    statementsT
} else {
    statementsF
}
```

where:
  condition is the Boolean value being tested
  statements$_T$ is a block of statements to be
      executed if *condition* is `true`
  statements$_F$ is a block of statements to be
      executed if *condition* is `false`

Syntax for *cascading* `if` statements:

```
if (condition1) {
    statements1
} else if (condition2) {
    statements2
} else if (condition3) {
    statements3
    . . .
} else {
    statementsnone
}
```

where:
  each *condition$_i$* is a Boolean expression
  each *statements$_i$* is a block of statements to
      be executed if *condition$_i$* is `true`
  statements$_{none}$ is the block of statements to
      be executed if every *condition$_i$* is `false`

FIGURE 4-2   **Program to classify an integer according to its sign**

```
/*
 * File: SignTest.java
 * ------------------
 * This program reads in an integer and classifies it as negative,
 * zero, or positive depending on its sign.
 */

import acm.program.*;

public class SignTest extends ConsoleProgram {
   public void run() {
      println("This program classifies an integer by its sign.");
      int n = readInt("Enter n: ");
      if (n > 0) {
         println("That number is positive.");
      } else if (n == 0) {
         println("That number is zero.");
      } else {
         println("That number is negative.");
      }
   }
}
```

In many situations, the process of choosing among a set of independent cases can be handled more efficiently using the **switch** statement, which is described in a separate section later in this chapter.

The **?:** operator

The Java programming language provides another, more compact mechanism for expressing conditional execution that can be extremely useful in certain situations: the **?:** operator. (This operator is referred to as *question-mark colon*, even though the two characters do not actually appear adjacent to one another.) Unlike any other operator in Java, **?:** is written in two parts and requires three operands. The general form of the operation is

> *condition* **?** *expression₁* **:** *expression₂*

When a Java program encounters the **?:** operator, it first evaluates the condition. If the condition turns out to be **true**, *expression₁* is evaluated and used as the value of the entire expression; if the condition is **false**, the value is the result of evaluating *expression₂*. The **?:** operator is therefore a shorthand form of the **if** statement

```
if (condition) {
   value = expression₁;
} else {
   value = expression₂;
}
```

where the value of the **?:** expression as a whole is whatever would have been stored in **value** in the expanded, **if**-statement form.

For example, you can use the **?:** operator to assign to **max** either the value of **x** or the value of **y**, whichever is greater, as follows:

```
max = (x > y) ? x : y;
```

The parentheses around the condition here are not technically required, but many Java programmers tend to include them in this context to enhance the readability of the code.

One of the most common situations in which the **?:** operator makes sense is in calls to **println** where the output you want differs slightly depending on some condition. For example, suppose that you are writing a program that counts the number of some item and that, after doing all the counting, stores the number of items in the variable **nItems**. How would you report this value to the user? The obvious way is just to call **println** using a statement like

```
println(nItems + " items found.");
```

But if you are a grammatical purist, you might be a little chagrined to read the output

when **nItems** happens to have the value 1. You could, however, correct the English by enclosing the **println** line in the following **if** statement:

```
if (nItems == 1) {
   println(nItems + " item found.");
} else {
   println(nItems + " items found.");
}
```

The only problem is that this solution strategy requires a five-line statement to express a relatively simple idea. As an alternative, you could use the **?:** operator as follows:

```
println(nItems + " item" + (nItems == 1 ? "" : "s") + " found.");
```

The string **"item"** in the output would then be followed by the empty string if **nItems** is equal to one and the string **"s"** otherwise. Note that the parentheses are necessary around the expression

```
(nItems == 1 ? "" : "s")
```

The **?:** operator has relatively low precedence with respect to **+**, which means that Java would try to do the concatenation first.

In Java, it is possible to overuse the **?:** operator. If an essential part of the decision-making structure with a program is hidden away in the **?:** operator, those decision-making operations can easily get lost in the rest of the code. On the other hand, if using **?:** makes it possible to handle some small detail without writing a complicated **if** statement, this operator can simplify the program structure considerably.

4.5 The `switch` statement

The `if` statement is ideal for those applications in which the program logic calls for a two-way decision point: some condition is either `true` or `false`, and the program acts accordingly. Some applications, however, call for more complicated decision structures involving more than two choices, where those choices can be divided into a set of mutually exclusive cases: in one case, the program should do *x*; in another case, it should do *y*; in a third, it should do *z*; and so forth. In many applications, the most appropriate statement to use for such situations is the `switch` statement, which is outlined in the syntax box on the right.

The header line of the `switch` statement is

> `switch (`*e*`)`

where *e* is an expression called the **control expression**. The body of the `switch` statement is divided into individual groups of statements introduced with one of two keywords: `case` or `default`. A `case` line and all the statements that follow it up to the next instance of either of these keywords are called a **case clause**; the `default` line and its associated statements are called the **default clause**. For example, in the template shown in the syntax box, the range of statements

> `case` *c₁*`:`
> > *statements₁*
> > `break;`

constitutes the first `case` clause.

Syntax for the `switch` statement:

```
switch (e) {
    case c₁:
        statements₁
        break;
    case c₂:
        statements₂
        break;
    . . . more case clauses . . .
    default:
        statementsdef
        break;
}
```

where:
  e is the control expression, which is used to choose what statements are executed
  each c_i is a constant value
  each *statementsᵢ* is a sequence of statements to be executed if c_i is equal to *e*
  statements_def is a sequence of statements to be executed if none of the c_i values match the expression *e*

When the program executes a `switch` statement, the control expression *e* is evaluated and compared against the values c_1, c_2, and so forth, each of which must be an integer constant (or, as you will see in Chapter 8, any value that behaves like an integer, such as a character). If one of the constants matches the value of the control expression, the statements in the associated `case` clause are executed. When the program reaches the `break` statement at the end of the clause, the operations specified by that clause are complete, and the program continues with the statement following the entire `switch` statement. If none of the case constants match the value of the control expression, the statements in the `default` clause are executed.

The template shown in the syntax box deliberately suggests that the `break` statements are a required part of the syntax. I encourage you to think of the `switch` syntax in precisely that form. Java is defined so that if the `break` statement is missing, the program starts executing statements from the next clause after it finishes the selected one. While this design can be useful in some cases, it tends to cause more problems than it solves. To reinforce the importance of remembering to include the `break` statement, every `case` clause in this text ends with an explicit `break` statement (or sometimes with a `return` statement, as discussed in Chapter 5).

The one exception to this rule is that multiple **case** lines specifying different constants can appear together, one after another, before the same statement group. For example, a **switch** statement might include the following code:

```
case 1:
case 2:
   statements
   break;
```

> It is good programming practice to include a **break** statement at the end of every **case** clause within a **switch** statement. Doing so will help you to avoid programming errors that can be extremely difficult to find. It is also good practice to include a **default** clause unless you are sure you have covered all the cases.

which indicates that the specified statements should be executed if the **select** expression is either 1 or 2. The Java compiler treats this construction as two **case** clauses, the first of which is empty. Because the empty clause contains no **break** statement, a program that selects that path simply continues on with the second clause. From a conceptual point of view, however, you are probably better off to think of this construction as a single **case** clause representing two possibilities.

The **default** clause is optional in the **switch** statement. If none of the cases match and there is no **default** clause, the program simply continues on with the next statement after the **switch** statement without taking any action at all. To avoid the possibility that the program might ignore an unexpected case, it is good programming practice to include a **default** clause in every **switch** statement unless you are certain you have enumerated all the possibilities.

Because the **switch** statement can be rather long, programs are easier to read if the **case** clauses themselves are short. If there is room to do so, it also helps to put the **case** identifier, the statements forming the body of the clause, and the **break** statement all together on the same line. This style is illustrated in the **CardRank** program in Figure 4-3, which shows an example of a **switch** statement that might prove useful in writing a program to play a card game. In this game, the cards within each suit are represented by the numbers 1 to 13. Displaying the number of the card is fine for the cards between 2 and 10, but this style of output is not very satisfying for the values 1, 11, 12, and 13, which should properly be represented using the names Ace, Jack, Queen, and King. The **CardRank** program uses the **switch** statement to display the correct symbol for each card.

The fact that the **switch** statement can only be used to choose between cases identified by an integer (or integer-like) constant does place some restrictions on its use. You will encounter situations in which you want to choose between several cases based on the value of a string variable or in which the values you want to use as case indicators are not constants. Since the **switch** statement cannot be used in such cases, you will instead need to rely on cascading **if** statements. In situations that allow the use of the **switch** statement, however, using it can make your program both more readable and more efficient.

4.6 The concept of iteration

Before looking at the details of Java's **while** and **for** statements, it is useful to take a step back and think holistically about the role that iteration—which is just a fancier computer science term for repetition—plays in the programming process. Think back for a moment to the **Add2Integers** program from Chapter 2. While that example helped illustrate how simple programs are written, it was hardly anything to get excited about. Most people are perfectly capable of adding two integers together with reasonable speed and accuracy. For a job that simple, a computer isn't necessary. But what if you needed to add ten

---

FIGURE 4-3   **Program to convert an integer to its rank as a playing card**

```java
/*
 * File: CardRank.java
 * ------------------
 * This program reads in an integer between 1 and 13 and
 * writes out the appropriate symbol for a playing card
 * of that rank.
 */

import acm.program.*;

public class CardRank extends ConsoleProgram {
   public void run() {
      println("This program converts integers to card ranks.");
      int n = readInt("Enter an integer between 1 and 13: ");
      switch (n) {
         case  1: println("Ace"); break;
         case 11: println("Jack"); break;
         case 12: println("Queen"); break;
         case 13: println("King"); break;
         default: println(n); break;
      }
   }
}
```

---

integers, or a thousand integers, or a million integers. At that point, the advantages of a computer become more clear.

But how would you go about changing the **Add2Integers** program so that it was able to add up integers on a larger scale? If you wanted to add four integers, you could simply add a few lines to the **run** method, as follows:

```java
public void run() {
   println("This program adds four integers.");
   int n1 = readInt("Enter n1: ");
   int n2 = readInt("Enter n2: ");
   int n3 = readInt("Enter n3: ");
   int n4 = readInt("Enter n4: ");
   int total = n1 + n2 + n3 + n4;
   println("The total is " + total + ".");
}
```

While this approach is suitable for adding four integers, it would quickly become tedious if you tried to apply the same strategy for ten integers, let alone a thousand or a million.

Suppose that you wanted to find a way to write the program for adding ten integers in a way that did not require the declaration of ten variables. Solving problems like this one is what makes computer programming hard; it is also what makes it interesting and fun. Think about the problem for a minute. Imagine that you are adding up 10 numbers—without a computer—and that I start calling those numbers out to you: 7, 4, 6, and so on. What would you do? You could write down the numbers and then add them at the end. This strategy is analogous to the one used in the **Add2Integers** program. It's effective, but it won't win any prizes for speed or cleverness. Alternatively, you could try adding

the numbers as you go: 7 plus 4 is 11, 11 plus 6 is 17, and so on. You don't have to keep track of each individual number, just the current total. When you hear the last number, you're all set to announce the answer.

The fact that you don't have to remember each individual number should help answer the question of how to add 10 integers without declaring 10 variables. With this new strategy, you should be able to write a new **Add10Integers** program using only two variables: one for each number as it comes in (**value**) and one for the sum of the numbers so far (**total**). Each time you read in a new number into **value**, you simply add it to **total**, which keeps track of the running total. Once you've done so, you can go back and reuse **value** to hold the next number, which is treated in precisely the same way.

This insight should enable you to begin the task of coding a program that uses the new strategy. For each input value, you know that you must execute the following steps:

1. Request an integer value from the user and store it in the variable **value.**
2. Add **value** to the running sum stored in the variable **total**.

You already know how to code the first step; it is a perfect example of the read-an-integer-from-the-user pattern that you saw many times in Chapter 2 and that has the following general form:

```
int variable = readInt(prompt);
```

You also know how to code the second step. Adding **value** to **total** is an instance of the shorthand assignment patterns introduced in Chapter 3. To add **value** to **total**, the idiom is

```
total += value;
```

The two patterns—one for reading in an integer and one for adding that integer to a running total—give you everything you need to code the operations that must occur for each input value in the **Add10Integers** program. For each of the 10 input values, the program must execute the following statements:

```
int value = readInt(" ? ");
total += value;
```

At this point, all you need to do is find some way to make the program execute this set of statements 10 times.

The repeat-N-times pattern

As you already learned in your encounters with Karel the Robot, Java contains an idiomatic pattern for repeating a block of code a predetermined number of times. That idiom is the **repeat-N-times pattern,** which has the following form:

```
for (int i = 0; i < N; i++) {
    statements to be repeated
}
```

In the repeat-N-times pattern, the value *N* indicates the number of repetitions you want. For example, if you replace *N* with 5, the statements enclosed within the braces will be executed five times. To use this pattern in the **Add10Integers** program, you need to replace *N* by 10. The statements enclosed in the braces are the statements that (1) read an

integer into **value** and (2) add that value to **total**. If you make these substitutions in the paradigm, you get the following code:

```
for (int i = 0; i < 10; i++) {
   int value = readInt(" ? ");
   total += value;
}
```

At this point, you are almost set to write the complete **Add10Integers** program, but there is a minor wrinkle that must still be addressed. The variable **value** is declared as an integer as part of the pattern for reading in an integer from the user. The variable **total**, however, is not yet declared. For this loop to work properly, **total** must be declared outside the loop and given an initial value of 0. Thus, before the **for** loop, you need to include the declaration

```
int total = 0;
```

to make sure that this variable can serve its function as a running total. Setting a variable to its proper starting value is called **initialization**. In many languages, failure to initialize variables is a common source of bugs. Java, however, is pretty good about checking for uninitialized variables and letting you know that such initialization is needed.

This final piece of the puzzle is all you need to complete the **Add10Integers** program. The **run** method you need would look like this:

```
public void run() {
   println("This program adds ten integers.");
   int total = 0;
   for (int i = 0; i < 10; i++) {
      int value = readInt(" ? ");
      total += value;
   }
   println("The total is " + total + ".");
}
```

This program, however, is not engineered as well as one would like. You won't always be in the position of adding exactly ten integers, and it should be easy to change your code so that it added a different number of integers. As things stand, you would have to go into the guts of the code and change both the initial message and the limit in the **for** loop. A better approach is to introduce a named constant as described in Chapter 3. If you choose to call this constant **NVALUES**, you would come up with a program that looked something like the **AddNIntegers** program in Figure 4-4.

The read-until-sentinel pattern

Even with the specification of the number of values as a named constant, the **AddNIntegers** program is unlikely to meet the needs of any significant number of users in its present form. Although making the change is comparatively easy, updating the program so that it adds a different number of integers still requires an explicit change in the program and a recompilation. What you really need is a more general program that can add any number of input values, where that number does not have to be specified in advance. From the user's point of view, having to count the numbers in advance is a bad idea. If you were using such a program, what you would like to do is enter the numbers until you've finished your list. At that point, you'd like to be able to tell the program that you have run out of numbers.

FIGURE 4-4   **Program to add a predefined number of integers**

```java
/*
 * File: AddNIntegers.java
 * ----------------------
 * This program adds a predefined number of integers and
 * then prints the sum at the end.  To change the number
 * of integers, change the definition of NVALUES.
 */

import acm.program.*;

public class AddNIntegers extends ConsoleProgram {

/* Specifies the number of values */
   private static final int NVALUES = 10;

/* Runs the program */
   public void run() {
      println("This program adds " + NVALUES + " integers.");
      int total = 0;
      for (int i = 0; i < NVALUES; i++) {
         int value = readInt(" ? ");
         total += value;
      }
      println("The total is " + total + ".");
   }

}
```

A common approach to solving this problem is to define a special input value and let the user enter that value to signal the end of the input list. A special value used to terminate a loop is called a **sentinel**. The choice of an appropriate value to use as a sentinel depends on the nature of the input data. The value chosen as a sentinel should not be a legitimate data value; that is, it should not be a value that the user would ever need to enter as normal data. For example, when adding a list of integers, the value 0 is an appropriate sentinel. There might be some 0s in a column of figures, but the user can always ignore them because they don't affect the final total. Note that the situation would be different if you were writing a program to average exam scores. Averaging in a 0 score does change the result, and some students have been known to get 0 scores from time to time. In this situation, 0 is a legitimate data value. To allow the user of the program to enter 0 as a score, it is necessary to choose a different sentinel value that does not represent an actual score. On most exams, it is impossible to have a negative score, so it would make sense to choose a value like –1 as the sentinel for that application.

To extend **AddNIntegers** into a new **AddIntegerList** program, the only change you need to make is in the loop structure. The **for** loop, which is most commonly used to execute a set of operations a predetermined number of times, is no longer appropriate. You need a new pattern that reads data until the special input sentinel is found. That pattern is the **read-until-sentinel pattern** and has the following form:

```java
while (true) {
     prompt user and read in a value
     if (value == sentinel) break;
     rest of body
}
```

This new pattern for a sentinel-based loop enables you to complete the **AddIntegerList** program, which is shown in Figure 4-5.

Later in this chapter, you will learn the details of the control statements out of which the read-until-sentinel pattern is formed. Even before you understand the details, you will find the pattern very useful. As you learn more about programming, however, you will discover that even expert programmers often use code that they don't understand in detail. In fact, one of the marks of an expert programmer is being able to use a library or a piece of code *without* understanding those details. As programs become more complex, the ability to use tools you understand only at the holistic level is an increasingly important skill.

FIGURE 4-5   **Program to add a list of integers where the end is marked by a sentinel**

```
/*
 * File: AddIntegerList.java
 * -------------------------
 * This program reads integers one per line until the
 * user enters a special sentinel value to signal the
 * end of the input.  At that point, the program
 * prints out the sum of the numbers entered so far.
 */

import acm.program.*;

public class AddIntegerList extends ConsoleProgram {

/* Specifies the value of the sentinel */
   private static final int SENTINEL = 0;

/* Runs the program */
   public void run() {
      println("This program adds a list of integers.");
      println("Enter values, one per line, using " + SENTINEL);
      println("to signal the end of the list.");
      int total = 0;
      while (true) {
         int value = readInt(" ? ");
         if (value == SENTINEL) break;
         total += value;
      }
      println("The total is " + total + ".");
   }

}
```

4.7 The **while** statement

The simplest iterative construct is the **while** statement, which repeatedly executes a simple statement or block until the conditional expression becomes **false**. The template for the **while** statement is shown in the syntax box on the right. As with the **if** statement, the Java compiler allows you to eliminate the curly braces surrounding the body if the body consists of a single statement. For the **while** loops used in this text, the body is always enclosed in braces to improve readability.

The entire statement, including both the **while** control line itself and the statements enclosed within the body, constitutes a **while loop**. When the program executes a **while** statement, it first evaluates the conditional expression to see if it is **true** or **false**. If it is **false**, the loop **terminates** and the program continues with the next statement after the entire loop. If the condition is **true**, the entire body is executed, after which the program goes back to the top to check the condition again. A single pass through the statements in the body constitutes a **cycle** of the loop.

There are two important principles to observe about the operation of a **while** loop:

1. The conditional test is performed before every cycle of the loop, including the first. If the test is **false** initially, the body of the loop is not executed at all.

2. The conditional test is performed only at the *beginning* of a loop cycle. If that condition happens to become **false** at some point during the loop, the program doesn't notice that fact until a complete cycle has been executed. At that point, the program evaluates the test condition again. If it is still **false**, the loop terminates.

> Syntax for the **while** statement:
>
> ```
>     while (condition) {
>         statements
>     }
> ```
>
> where:
>   *condition* is the conditional test used to determine whether the loop should continue for another cycle
>   *statements* are the statements to be repeated

Using the **while** loop

Suppose that you have been asked to write a program that adds up the digits in a positive integer. A sample run for this program might then be
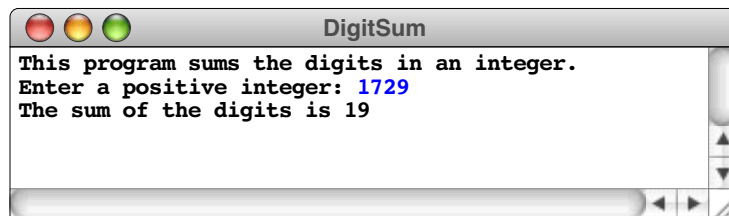
where the result of 19 comes from adding 1 + 7 + 2 + 9. How would you go about writing such a program?

In the **AddNIntegers** program from Figure 4-4 earlier in this chapter, you learned how to keep a running total. You need to declare a variable for the sum, initialize it to 0, go through a loop adding in digits, and finally display the sum at the end. That much of the structure, with the rest of the problem left written in English, is shown below:

```
public void run() {
   println("This program sums the digits in an integer.");
   int n = readInt("Enter a positive integer: ");
   int dsum = 0;
   For each digit in the number, add that digit to dsum.
   println("The sum of the digits is " + dsum);
}
```

The sentence

For each digit in the number, add that digit to **dsum**.

clearly specifies a loop structure of some sort, since there is an operation that needs to be repeated for each digit in the number. If it were easy to determine how many digits a number contained, you might choose to use a **for** loop and count up to the number of digits. Unfortunately, finding out how many digits there are in an integer is just as hard as adding them up in the first place. The best way to write this program is just to keep adding in digits until you discover that you have added the last one. Loops that run until some condition occurs are most often coded using the **while** statement.

The essence of this problem lies in determining how to break up a number into its component digits. The key insight is that the arithmetic operators **/** and **%** are sufficient to accomplish the task. The last digit of an integer **n** is simply the remainder left over when **n** is divided by 10, which is the result of the expression **n % 10**. The rest of the number—the integer that consists of all digits *except* for the last one—is given by **n / 10**. For example, if **n** has the value 1729, the **/** and **%** operators can be used to break that number into two parts, 172 and 9, as shown in the following diagram:
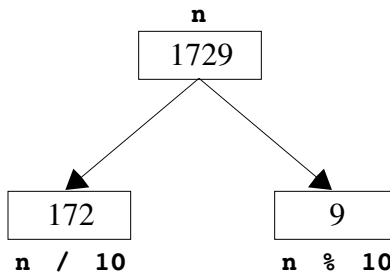
Thus, in order to add up the digits in the number, all you need to do is add the value **n % 10** to the variable **dsum** on each cycle of the loop and then divide the number **n** by 10. The next cycle will add in the second-to-last digit from the original number, and so on, until the entire number has been processed in this way.

But how do you know when to stop? Eventually, as you divide **n** by 10 in each cycle, you will reach the point at which **n** becomes 0. At that point, you've processed all the digits in the number and can exit from the loop. In other words, as long as the value of **n** is greater than 0, you should keep going. Thus, the **while** loop needed for the problem is

```
while (n > 0) {
   dsum += n % 10;
   n /= 10;
}
```

The entire **DigitSum** program is shown in Figure 4-6.

FIGURE 4-6   **Program to sum the digits in an integer**

```
/*
 * File: DigitSum.java
 * ------------------
 * This program sums the digits in a positive integer.
 * The program depends on the fact that the last digit of
 * a integer n is given by n % 10 and the number consisting
 * of all but the last digit is given by the expression n / 10.
 */

import acm.program.*;

public class DigitSum extends ConsoleProgram {

   public void run() {
      println("This program sums the digits in an integer.");
      int n = readInt("Enter a positive integer: ");
      int dsum = 0;
      while (n > 0) {
         dsum += n % 10;
         n /= 10;
      }
      println("The sum of the digits is " + dsum);
   }

}
```

Infinite loops

When you use a **while** loop in a program, it is important to make sure that the condition used to control the loop will eventually become **false**, so that the loop can exit. If the condition in the while control line always evaluates to **true**, the computer will keep executing cycle after cycle without stopping. This situation is called an **infinite loop**.

> Think carefully about the conditional expression you use in a **while** loop so that you can be sure the loop will eventually exit. A loop that never finishes is called an *infinite loop*.

As an example, suppose that you had carelessly written the **while** loop in the **DigitSum** program with a **>=** operator in the control line instead of the correct **>** operator, as shown below:

```
while (n >= 0) {
   dsum += n % 10;
   n /= 10;
}
```

The loop no longer stops when **n** is reduced to 0, as it does in the correctly coded example. Instead, the computer keeps executing the body over and over and over again, with **n** equal to 0 every time.

To stop an infinite loop, you must type a special command sequence on the keyboard to interrupt the program and forcibly cause it to quit. This command sequence differs

from machine to machine, and you should be sure to learn what command to use on your own computer.

Solving the loop-and-a-half problem

The **while** loop is designed for situations in which there is some test condition that can be applied at the beginning of a repeated operation, before any of the statements in the body of the loop are executed. If the problem you are trying to solve fits this structure, the **while** loop is the perfect tool. Unfortunately, many programming problems do not fit easily into the standard **while** loop template. Instead of allowing a convenient test at the beginning of the operation, some problems are structured in such a way that the test you would like to write to determine if the loop is complete falls most naturally somewhere in the middle of the loop.

Consider for example, the problem of reading input data until a sentinel value appears, which was discussed in the section on "Sentinel-based loops" earlier in this chapter. When expressed in English, the structure of the sentinel-based loop consists of repeating the following steps:

1. Read in a value.
2. If the value is equal to the sentinel, exit from the loop.
3. Perform whatever processing is required for that value.

Unfortunately, there is no test you can perform at the very beginning of the loop to determine whether the loop is finished. The termination condition for the loop is reached when the input value is equal to the sentinel; in order to check this condition, the program must have first read in some value. If the program has not yet read in a value, the termination condition doesn't make sense. Before the program can make any meaningful test, it must have executed the part of the loop that reads in the input value. When a loop contains some operations that must be performed before testing for completion, it represents an instance of what programmers call the **loop-and-a-half problem**.

One way to solve the loop-and-a-half problem in Java is to use the **break** statement, which, in addition to its use in the **switch** statement, has the effect of immediately terminating the innermost enclosing loop. By using **break**, it is possible to code the loop structure for the sentinel problem in a form that follows the natural structure of the problem:

```
while (true) {
    prompt user and read in a value
    if (value == sentinel) break;
    process the data value
}
```

The initial line

```
while (true)
```

needs some explanation. The **while** loop is defined so that it continues until the condition in parentheses becomes **false**. The symbol **true** is a constant, so it can never become **false**. Thus, as far as the **while** statement itself is concerned, the loop will never terminate. The only way this program can exit from the loop is by executing the **break** statement inside it.

It is possible to code this sort of loop without using the **while (true)** control line or the **break** statement. To do so, however, you must change the order of operations within the loop and request input data in two places: once before the loop begins and then again inside the loop body. When structured in this way, the template for the sentinel-based loop is

> *prompt user and read in the first value*
> **while (***value* **!=** *sentinel***) {**
>   *process the data value*
>   *prompt user and read in a new value*
> **}**

Figure 4-7 shows how this template can be used to implement the **AddIntegerList** program presented in Figure 4-5 without using a **break** statement.

Unfortunately, there are two drawbacks to using this strategy. First, the order of operations in the loop is not what most people would expect. In any English explanation of the solution strategy, the first step is to get a number and the second is to add it to the total. The **while** loop template used in Figure 4-5 reverses the order of the statements within the loop and makes the program more difficult to follow. The second problem is that this template requires two copies of the statements that read in a number.

FIGURE 4-7   **Rewrite of AddIntegerList without a break statement**

```java
/*
 * File: AddIntegerList.java
 * -------------------------
 * This program reads integers one per line until the
 * user enters a special sentinel value to signal the
 * end of the input.  At that point, the program
 * prints out the sum of the numbers entered so far.
 */

import acm.program.*;

public class AddIntegerList extends ConsoleProgram {

/* Specifies the value of the sentinel */
   private static final int SENTINEL = 0;

/* Runs the program */
   public void run() {
      println("This program adds a list of integers.");
      println("Enter values, one per line, using " + SENTINEL);
      println("to signal the end of the list.");
      int total = 0;
      int value = readInt(" ? ");
      while (value != SENTINEL) {
         total += value;
         value = readInt(" ? ");
      }
      println("The total is " + total + ".");
   }

}
```

Duplication of code presents a serious maintenance problem because subsequent edits to one set of statements might not be made in the other. Empirical studies have shown that students who learn to solve the loop-and-a-half problem using the **break** statement are far more likely to write correct programs than those who don't.

Despite the problems that arise when students are unable to code exits from the interior of a loop, some instructors dislike using **break** to solve the loop-and-a-half problem. The principal reason for doing so is that it is easy to overuse the **break** statement in Java. One way to guard against the overuse of the **break** statement is disallow its use entirely. To me, such an approach seems overly draconian. In this text, I use the **break** statement within a **while** loop only to solve the loop-and-a-half problem and not in other, more complex situations where its use is likely to obscure the program's structure.

4.8 The **for** statement

One of the most important control statements in Java is the **for** statement, which is most often used in situations in which you want to repeat an operation a particular number of times. The general form of the **for** statement is shown in the syntax box to the right.

> Syntax for the **for** statement:
>
>   for (init; test; step) {
>   statements
>   }
>
> where:
>   *init* is a declaration that initializes the loop index variable
>   *test* is a conditional test used to determine whether the loop should continue, just as in the **while** statement
>   *step* is an expression used to prepare for the next loop cycle
>   *statements* are the statements to be repeated

The operation of the **for** loop is determined by the three italicized expressions on the **for** control line: *init*, *test*, and *step*. The *init* expression indicates how the **for** loop should be initialized, usually by declaring an index variable and setting its initial value. For example, if you write

```
for (int i = 0; . . .
```

the loop will begin by declaring the index variable **i** and setting it to 0. If the loop begins

```
for (int i = -7; . . .
```

the variable **i** will start as **-7**, and so on.

The *test* expression is a conditional test written exactly like the test in a **while** statement. As long as the test expression is **true**, the loop continues. Thus, in the loop that has served as our canonical example up to now

```
for (int i = 0; i < n; i++)
```

the loop begins with **i** equal to 0 and continues as long as **i** is less than **n**, which turns out to represent a total of **n** cycles, with **i** taking on the values $0, 1, 2,$ and so forth, up to the final value $n - 1$. The loop

```
for (int i = 1; i <= n; i++)
```

begins with **i** equal to 1 and continues as long as **i** is less than or equal to **n**. This loop also runs for **n** cycles, with **i** taking on the values $1, 2,$ and so forth, up to **n**.

The *step* expression indicates how the value of the index variable changes from cycle to cycle. The most common form of step specification is to increment the index variable

using the **++** operator but this is not the only possibility. For example, one can count backward by using the **--** operator or count by twos by using **+= 2** instead of **++**.

As an illustration of counting in the reverse direction, the **Countdown** program in Figure 4-8 counts down from 10 to 0. When **Countdown** is run, it generates the following sample run:
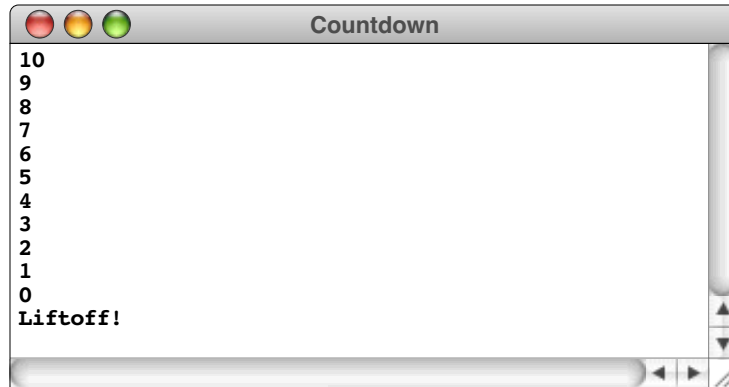
The **Countdown** program demonstrates that any variable can be used as an index variable. In this case, the variable is called **t**, presumably because that is the traditional variable for a rocket countdown, as in the phrase "T minus 10 seconds and counting." In any case, the index variable must be declared at the beginning of the program, just like any other variable.

FIGURE 4-8   **Program to count down to 0 as in a rocket launch sequence**

```
/*
 * File: Countdown.java
 * --------------------
 * This program counts backwards from the value START
 * to zero, as in the countdown preceding a rocket
 * launch.
 */

import acm.program.*;

public class Countdown extends ConsoleProgram {

/* Specifies the value from which to start counting down */
   private static final int START = 10;

/* Runs the program */
   public void run() {
      for (int t = START; t >= 0; t--) {
         println(t);
      }
      println("Liftoff!");
   }

}
```

   The expressions *init*, *test*, and *step* are each optional, but the semicolons must appear. If *init* is missing, no initialization is performed. If *test* is missing, it is assumed to be **true**. If *step* is missing, no action occurs between loop cycles. Thus the control line

```
for (;;)
```

is identical in operation to

```
while (true)
```

The relationship between **for** and **while**

As it happens, the **for** statement

```
for (init; test; step) {
     statements;
}
```

is identical in operation to the **while** statement

```
init;
while (test) {
     statements;
     step;
}
```

   Even though the **for** statement can easily be rewritten using **while**, there are considerable advantages to using the **for** statement whenever it is possible to do so. With a **for** statement, all the information you need to understand exactly which cycles will be executed is contained in the header line of the statement. For example, whenever you see the statement

```
for (int i = 0; i < 10; i++) {
     . . . body . . .
}
```

in a program, you know that the statements in the body of the loop will be executed 10 times, once for each of the values of **i** between 0 and 9. In the equivalent **while** loop form

```
int i = 0;
while (i < 10) {
     . . . body . . .
     i++;
}
```

the increment operation at the bottom of the loop can easily get lost if the body is large.

Using **for** with floating-point data

Because the *init*, *test*, and *step* components of the **for** loop can be arbitrary expressions, there is no obvious reason why the loop index in a **for** loop has to be an integer. The fact that it is possible to count from 0 to 10 by twos using the **for** loop

```
for (int i = 0; i <= 10; i += 2) . . .
```

suggests that it might also be possible to count for 1.0 to 2.0 in increments of 0.1 by declaring the loop index variable as a **double**. For example, you might try to display the values in this range by writing

```
for (double x = 1.0; x <= 2.0; x += 0.1) {
    println(x);
}
```

If you run this program in Java, you don't get a list that begins 1.0, 1.1, 1.2, and so on up to 2.0, but instead see the following output:

```
ForLoopWithDoubles
1.0
1.1
1.2000000000000002
1.3000000000000003
1.4000000000000004
1.5000000000000004
1.6000000000000005
1.7000000000000006
1.8000000000000007
1.9000000000000008
```

There are a lot of seemingly extraneous digits and there is nothing even close to the value 2.0.

The problem here is that floating-point numbers are not exact. The value 0.1 is very close to the mathematical fraction $\frac{1}{10}$ but is almost certainly not precisely equal to it. As 0.1 is added to the index variable **x**, the inaccuracy can accumulate to the point that, when **x** is tested against 2.0 to determine whether the loop is finished, its value may be 2.000000001 or something similar, which is not less than or equal to 2.0. The condition in the **for** loop is therefore not satisfied, and the loop terminates after running for what seems to be one too few cycles. The best way to fix this problem is to restrict yourself to using integers as index variables in **for** loops. Because integers are exact, the problem never arises.

> Be careful when testing floating-point numbers for equality. Because floating-point numbers are only approximations, they might not behave in the same way as real numbers in mathematics. In general, it is best to avoid using a floating-point variable as a **for** loop index.

The same warning about comparing floating-point numbers for equality applies in many other circumstances besides the **for** loop. Numbers that seem as if they should be exactly equal might not be, given the limitations on the accuracy of floating-point numbers.

Summary

In Chapter 2, you looked at the process of programming from a holistic perspective. Along the way, you learned about several control statements in an informal way. In this chapter, you were able to investigate how those statements work in more detail. You were also introduced to a new type of data called *Boolean data*. Although this data type

contains only two values—**true** and **false**—being able to use Boolean data effectively is extremely important to successful programming and is well worth a little extra practice.

The important points introduced in this chapter include:

• *Simple statements* consist of an expression followed by a semicolon.
• The = used to specify assignment is an operator in Java. Assignments are therefore legal expressions, which makes it possible to write *embedded* and *multiple assignments*.
• Individual statements can be collected into *compound statements*, more commonly called *blocks*.
• Control statements fall into two classes: *conditional* and *iterative*.
• Java defines a data type called **boolean** that is used to represent Boolean data. The type **boolean** has only two values: **true** and **false**.
• You can generate Boolean values using the *relational operators* (<, <=, >, >=, ==, and !=) and combine them using the *logical operators* (&&, ||, and !).
• The logical operators && and || are evaluated in left-to-right order in such a way that the evaluation stops as soon as the program can determine the result. This behavior is called *short-circuit evaluation*.
• The **if** statement is used to express conditional execution when a section of code should be executed only in certain cases or when the program needs to choose between two alternate paths.
• The **switch** statement is used to express conditional execution when a problem has the following structure: in case 1, do this; in case 2, do that; and so forth.
• The **while** statement specifies repetition that occurs as long as some condition is met.
• The **for** statement specifies repetition in which some action is needed on each cycle in order to update the value of an index variable.

Review questions

1. Is the construction

        17;

   a legal statement in Java? Is it useful?

2. Describe the effect of the following statement, assuming that **i**, **j**, and **k** are declared as integer variables:

        i = (j = 4) * (k = 16);

3. What single statement would you write to set both **x** and **y** (which you may assume are declared to be type **double**) to 1.0?

4. What is a *block?* What important fact about blocks is conveyed by the term *compound statement*, which is another name for the same concept?

5. What are the two classes of control statements?

6. What does it mean to say that two control statements are *nested?*

7. What are the two values of the data type **boolean**?

8. What happens when a programmer tries to use the mathematical symbol for equality in a conditional expression?

9. What restriction does Java place on the types of values that can be compared using the relational operators?

10. How would you write a Boolean expression to test whether the value of the integer variable **n** was in the range 0 to 9, inclusive?

11. Describe in English what the following conditional expression means:

```
(x != 4) || (x != 17)
```

For what values of **x** is this condition **true**?

12. What does the term *short-circuit evaluation* mean?

13. Assuming that **myFlag** is declared as a Boolean variable, what is the problem with writing the following **if** statement?

```
if (myFlag == true) . . .
```

14. What are the four different formats of the **if** statement used in this text?

15. Describe in English the general operation of the **switch** statement.

16. Suppose the body of a **while** loop contains a statement that, when executed, causes the condition for that **while** loop to become **false**. Does the loop terminate immediately at that point or does it complete the current cycle?

17. Why is it important for the **DigitSum** program in Figure 4-6 to specify that the integer is positive?

18. What is the *loop-and-a-half problem*? What two strategies are presented in the text for solving it?

19. What is the purpose of each of the three expressions that appear in the control line of a **for** statement?

20. What **for** loop control line would you use in each of the following situations:

    a) Counting from 1 to 100.
    b) Counting by sevens starting at 0 until the number has more than two digits.
    c) Counting backward by twos from 100 to 0.

21. Why is it best to avoid using a floating-point variable as the index variable in a **for** loop?

Programming exercises

1. As a way to pass the time on long bus trips, young people growing up in the United States have been known to sing the following rather repetitive song:

    > 99 bottles of beer on the wall.
    > 99 bottles of beer.
    > You take one down, pass it around.
    > 98 bottles of beer on the wall.
    >
    > 98 bottles of beer on the wall. . . .

    Anyway, you get the idea. Write a Java program to generate the lyrics to this song. (Since you probably never actually finished singing it, you should decide how you want the song to end.) In testing your program, it would make sense to use some constant other than 99 as the initial number of bottles.

2. While we're on the subject of silly songs, another old standby is "This Old Man," for which the first verse is

    > This old man, he played 1.
    > He played knick-knack on my thumb.
    > With a knick-knack, paddy-whack,
    > Give your dog a bone.
    > This old man came rolling home.

    Each subsequent verse is the same, except for the number and the rhyming word at the end of the second line, which gets replaced as follows:

2—shoe	5—hive	8—pate
3—knee	6—sticks	9—spine
4—door	7—up to heaven	10—shin

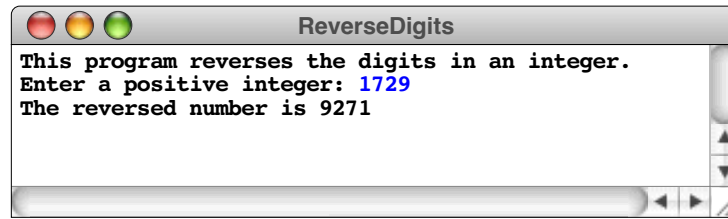   Write a program to display all 10 verses of this song.

3. Write a program that reads in a positive integer N and then calculates and displays the sum of the first N odd integers. For example, if N is 4, your program should display the value 16, which is $1 + 3 + 5 + 7$.

4.
    > *Why is everything either at sixes or at sevens?*
    > — Gilbert and Sullivan, *H.M.S. Pinafore,* 1878

    Write a program that displays the integers between 1 and 100 that are divisible by either 6 or 7.

5. Repeat exercise 4, but this time have your program display only those numbers that are divisible by 6 or 7 but not both.

6. Using the **AddIntegerList** program from Figure 4-5 as a model, write a program called **AverageList** that reads in a list of integers representing exam scores and then prints out the average. Because some unfortunate student might actually get a score of 0, your program should use –1 as the sentinel to mark the end of the input.

7. Rewrite the **DigitSum** program given in Figure 4-6 so that instead of adding the digits in the number, it generates the number that has the same digits in the reverse order, as illustrated by this sample run:

```
ReverseDigits
This program reverses the digits in an integer.
Enter a positive integer: 1729
The reversed number is 9271
```

8. Rewrite the **Countdown** program given in Figure 4-8 so that it uses a **while** loop instead of a **for** loop.

9. In mathematics, there is a famous sequence of numbers called the Fibonacci sequence after the thirteenth-century Italian mathematician Leonardo Fibonacci. The first two terms in this sequence are 0 and 1, and every subsequent term is the sum of the preceding two. Thus the first several numbers in the Fibonacci sequence are as follows:

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_2 &= 1 \quad (0 + 1) \\
F_3 &= 2 \quad (1 + 1) \\
F_4 &= 3 \quad (1 + 2) \\
F_5 &= 5 \quad (2 + 3) \\
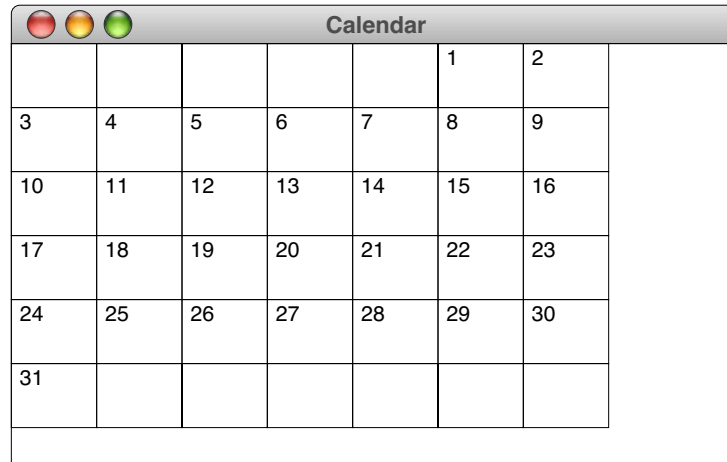F_6 &= 8 \quad (3 + 5)
\end{aligned}
$$

Write a program to display the values in this sequence from F_0 through F_{15}.

10. Modify the program in the preceding exercise so that instead of specifying the index of the final term, the program displays those terms in the Fibonacci sequence that are less than 10,000.

11. Write a **GraphicsProgram** that uses two nested **for** loops to create the following checkerboard diagram in the upper left corner of the canvas:
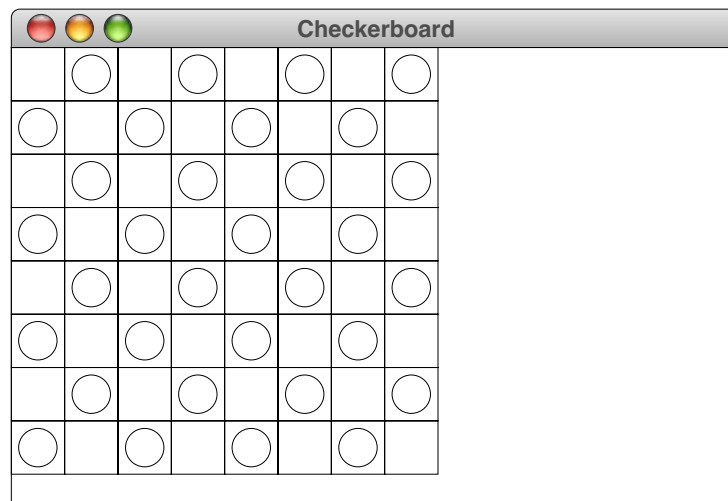
The individual squares are all instances of the class **GRect** introduced in Chapter 2. Alternate squares both horizontally and vertically contain a somewhat smaller **GOval** to create the checkerboard effect.

12. Using much the same strategy as you did in the preceding exercise, write a **GraphicsProgram** that creates a simple calendar diagram similar to the one shown in the following diagram:

Your program should use the following named constants to control the format of the calendar display:

```
/* The number of days in the month */
private static final int DAYS_IN_MONTH = 31;

/* The day of the week on which the month starts */
/* (Sunday = 0, Monday = 1, Tuesday = 2, and so on) */
private static final int DAY_MONTH_STARTS = 5;

/* The width in pixels of a day on the calendar */
private static final int DAY_WIDTH = 40;

/* The height in pixels of a day on the calendar */
private static final int DAY_HEIGHT = 30;
```

Your display should generate exactly the number of rows necessary to display the days of the month. Here, in a 31-day month that begins on a Friday, the calendar needs six rows; if you were generating a calendar for a non-leap-year February that began on a Sunday, the calendar would require only four rows.

Chapter 5
Methods

With method and logic one can accomplish anything.
— Agatha Christie, *Poirot Investigates,* 1924
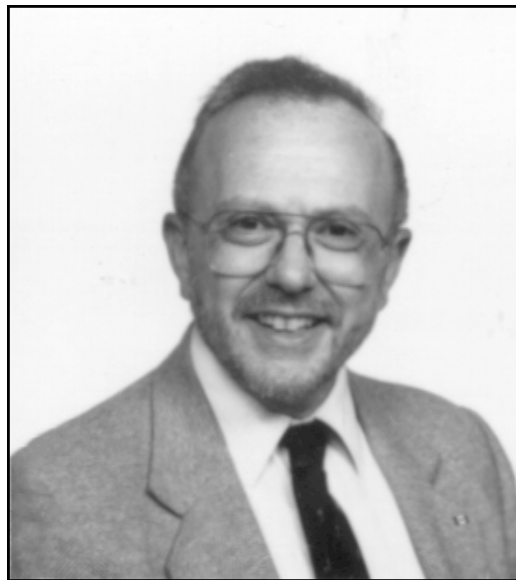
David Parnas

David Parnas is Professor of Computer Science at Limerick University in Ireland, where he directs the Software Quality Research Laboratory, and has also taught at universities in Germany, Canada, and the United States. His most influential contribution to software engineering is his groundbreaking 1972 paper entitled "On the criteria to be used in decomposing systems into modules," which laid much of the first structured programming. Professor Parnas also attracted considerable public attention in 1985 when he resigned from a Department of Defense panel investigating the software requirements of the proposed Strategic Defense Initiative—more commonly known as "Star Wars"—on the grounds that the requirements of the system were impossible to achieve.

This chapter examines in more detail the concept of a method, which was first introduced in Chapter 2. A method is a set of statements that have been collected together and given a name. By allowing the programmer to signify the entire set of operations with a single name, programs become much shorter and much simpler. Without methods, simple programs would become unmanageable as they increased in size and sophistication.

In order to appreciate how methods reduce the complexity of programs, you need to understand the concept in two ways. From the reductionistic perspective, you need to understand how methods work in an operational sense so you can predict their behavior. At the same time, you must be able to take a step backward and look at methods holistically, so that you can also understand why they are important and how to use them effectively.

5.1 A quick overview of methods

You have been working with methods ever since Chapter 2. Before turning to the details of how methods work, it helps to review the basic terminology for methods that was introduced in that chapter. First of all, a **method** consists of a set of statements that have been collected together and given a name. The act of executing the set of statements associated with a method is known as **calling** that method. To indicate a method call in Java, you write the name of the method, followed by a list of expressions enclosed in parentheses. These expressions are called **arguments** and allow the caller to pass information to the method.

As an example, consider the first line from the **Add2Integers** program that appeared as Figure 2-2, which looks like this:

```
println("This program adds two integers.");
```

This statement represents a call to the **println** method, which prints out a line of information on the console. To make any sense of that idea, however, the **println** method has to know what information to display. Here, that information is provided through the argument, which consists of the string **"This program adds two integers."**.

Once called, a method takes the data supplied as arguments, does its work, and then returns to the point in the program at which the call was made. Remembering what the calling program was doing and being able to get back precisely to that point is one of the defining characteristics of the method-calling mechanism. The operation of going back to the calling program is called **returning** from the method. As part of the return operation, methods can also send results back to the calling program, as illustrated by the **readInt** method in the now-familiar declaration

```
int n1 = readInt("Enter n1: ");
```

After the **readInt** method performs its task of reading in an integer from the user, it passes that integer back to the calling program as the value of the call. This operation is called **returning a value**.

Methods as mechanisms for hiding complexity

Take another look at the line

```
println("This program adds two integers.");
```

which begins the **Add2Integers** program. When this program first appeared back in Chapter 2, the text described what the **println** method does in very simple terms: it takes the string you have provided as an argument and makes it appear on the console that is supplied as part of every **ConsoleProgram**. The text, however, was silent on the question of how **println** accomplishes that operation. The details of the underlying implementation remain largely a mystery, which to a certain extent makes the operation of **println** seem like magic.

In J. K. Rowling's *Harry Potter and the Chamber of Secrets,* Arthur Weasley warns that you should "never trust anything that can think for itself if you can't see where it keeps its brain." While that may be sage advice in the wizarding world, programmers must often do precisely that. You did not write **println** and indeed would have no way of doing so at this point in your study of programming. It would be extremely difficult even to understand how it works. Even so, there is nothing to stop you from using it effectively.

In fact, one of the great advantages of methods is that you can use them without having to understand the complexity that goes on underneath. Methods provide a way of hiding lower-level implementation details so that the caller need not be bothered by them. In computer science, this technique is called **information hiding.** The fundamental idea, which was championed by David Parnas in the early 1970s, is that the complexity of programming systems is best managed by making sure that details are visible only to those levels of the program at which they are relevant. The programmers who implement **println** and make sure that it works need to know those details. The programmers that merely use **println** can remain blissfully unaware of all that underlying mechanism.

Methods as tools for programmers rather than users

Students who are just beginning their study of programming sometimes have difficulty understanding the distinction between methods and programs. To some extent, both programs and methods have the effect of collecting a sequence of statements and giving it a single name. If you look at the **Add2Integers** program, it acts as if it is a shorthand for the set steps contained in its **run** method.

It is, however, important to keep these concepts distinct. The principal difference between a method and a program lies in who or what makes use of it. When, as a user, you sit down in front of your computer and start up an application, you are running a program that performs some action on your behalf. Thus, programs are invoked by and serve the needs of an external *user*. Methods, on the other hand, provide a mechanism by which a *program* can invoke a set of previously defined operations on its behalf. The operation of a method is thus internal to the program domain.

A similar confusion often arises between the arguments and program input on the one hand, and return values and program output on the other. It is easy to see input data entered using **readInt** as analogous to values passed as arguments. Both, after all, represent a way of passing data into some piece of a program. Despite that conceptual similarity, it is critically important to make a sharp distinction between input operations, such as **readInt**, and the use of

> Be careful to differentiate in your mind the ideas of input and output in the program domain and the related concepts of arguments and results in the method domain. Input and output refer to communication between a *program* and its *user*. Arguments and results represent communication between a *method* and its *caller*.

arguments in the method domain. A method like **readInt** provides a mechanism for getting input from the *user*. When **readInt** needs an input value, whoever is sitting in front of the terminal must physically enter that value on the keyboard. Arguments to a

method, on the other hand, provide a means for a method to receive input from its caller, which is simply another part of the program. Data passed in the form of arguments may have been entered by the user at an earlier point in the program, but could just as easily have been calculated as part of the program operation. You should also be careful to differentiate the use of output operations, such as **println**, from the technique of returning a result. When you use **println**, the output appears on the console. When a method returns a result, that information goes back to its caller, which is free to use it in whatever way makes sense for the program. New programmers have a tendency to use input/output operations within methods when the logic of the situation calls for using arguments and results.

Method calls as expressions

In Chapter 3, method calls were listed as one of the categories of Java expressions. As you try to understand how methods fit into the overall Java framework, it will often be helpful to remember that a method call is simply an expression and can be used in any context in which an expression can appear. Moreover, the arguments to a method are also expressions, which can themselves contain method calls or any other operations that would be legal in an expression.

To illustrate that methods and their arguments are expressions, it is useful to introduce several standard methods from the **Math** class, which are listed in Figure 5-1. As you can see from the list of available methods, the **Math** class includes many of the standard

FIGURE 5-1	Selected methods from the Math class

Math.abs(x)   The absolute value of **x**, which can be of any numeric type.
Math.min(x, y)   The smaller of **x** and **y**.
Math.max(x, y)   The larger of **x** and **y**.
Math.sqrt(x)   The square root of the value **x**.
Math.log(x)   The natural logarithm of **x**, which uses the mathematical constant e as its base.
Math.exp(x)   The inverse logarithm of **x**, which is e^x.
Math.pow(x, y)   The value **x** raised to the **y** power.
Math.sin(theta)   The trigonometric sine of the angle **theta**; in the **Math** class, all angles are measured in radians.
Math.cos(theta)   The cosine of the angle **theta**.
Math.tan(theta)   The tangent of the angle **theta**.
Math.asin(x)   The angle whose sine is **x**.
Math.acos(x)   The angle whose cosine is **x**.
Math.atan(x)   The angle whose tangent is **x**.
Math.toRadians(degrees)   Converts an angle from degrees to radians.
Math.toDegrees(radians)   Converts an angle from radians to degrees.

mathematical functions you learned in high-school algebra and trigonometry. For example, the **Math** class includes the method **sqrt** for taking the square root of its argument, as well as **sin** and **cos** for trigonometric sines and cosines. Each of these methods takes a **double** as an argument and returns a result, also of type **double**. You can use these methods in simple statements, such as

```
double root3 = Math.sqrt(3.0);
```

or in more complicated ones. For example, you can compute the distance from the origin to the point (x, y) using the standard distance formula for points in a plane:

$$distance = \sqrt{x^2 + y^2}$$

In Java, this formula corresponds to the statement

```
double distance = Math.sqrt(x * x + y * y);
```

Similarly, you can compute a tangent using the trigonometric identity

$$\tan \theta = \frac{\sin \theta}{\cos \theta}$$

which can be written in Java as

```
double tangent = Math.sin(theta) / Math.cos(theta);
```

5.2 Methods and the object-oriented paradigm

The discussion of methods to date, however, has left out one of the most important characteristics of methods in an object-oriented language. In Chapter 1, I identified object-oriented languages as those in which programs are viewed as a collection of "objects" for which the data and the operations acting on that data are encapsulated into integrated units. In Java, operations on data are indeed specified using methods, but there has been no clear sense so far that these operations are in any way bound together with data into objects. It is time to remedy that omission.

Method calls as messages

In an object-oriented language like Java, methods are defined as part of a class. Any object that is an instance of that class has access to the methods defined within that class as well as any public methods it inherits from its superclasses. Within the context of that object, method calls look precisely as you have seen them in the various examples in the previous chapters:

> *name* **(** *arguments* **)**

That simple format, however, tells only part of the story. In the object-oriented world, objects communicate by sending information and requests from one object to another. Collectively, these transmissions among objects are called **messages.** The conceptual act of sending a message corresponds to having one object invoke a method that belongs to a different object. In general, the code is running as part of the sending object. To make it possible to send a message, the program must specify the receiving object. Identifying the receiver is accomplished by using a slightly different format for the call, as follows:

> *receiver* **.** *name* **(** *arguments* **)**

The only difference here is that the receiver has been made explicit by including it as part of the call.

In the examples you've seen so far, the receiver has always been the current object, in which case the receiver specification can be omitted. For example, the **println** method is defined as a public method in the **Program** class. That means that every subclass of **Program** can invoke the **println** method simply by writing

> **println(***value***)**

If you wanted to make the receiver explicit—as some instructors do—you could use the Java keyword **this**, which refers to the current object, as follows:

> **this.println(***value***)**

Using GObjects as receivers

The easiest way to illustrate the use of receivers is in the context of a **GraphicsProgram**. The very first program in Chapter 2 contained the following **run** method:

```
public void run() {
    add(new GLabel("hello, world"), 100, 75);
}
```

Let's decompose the body of this method into its constituent parts. The **add** method—which is defined in **GraphicsProgram** and therefore can be called without specifying a receiver from a **GraphicsProgram**—has three arguments. The first is an expression that creates a new **GLabel** object, and the other two are integers specifying the coordinates 100 and 75. The expression

```
new GLabel("hello, world")
```

is an example of the invocation of a special form of method called a **constructor,** which is used to create a new object.

Objects are values in Java, so it would be possible to take this new **GLabel** and, instead of passing it immediately to the **add** method, assign it to a variable. Thus, the following two-line **run** method gives rise to a program that has the same effect as the original:

```
public void run() {
    GLabel label = new GLabel("hello, world");
    add(label, 100, 75);
}
```

This change, however, enables all sorts of new possibilities. As soon as you have your hands on an object—in this case, the **GLabel** stored in the variable **label**—you can begin to do things with it. In particular, you can send it messages. Such messages consist of method invocations that specify **label** as the receiver. Moreover, since label is an instance of the class **GLabel**, the methods you invoke have to be appropriate to that class, either because they are defined in **GLabel** itself or because they are inherited from the **GObject** class from which **GLabel** is derived.

So the question then becomes what methods might you call. In Chapter 7, you'll learn the full set of capabilities offered by the classes in acm.graphics, but for now it's worth

experimenting with a few simple ones. The output of the current version of the program looks like this:

This output is not so exciting. The label is so small that it is hard to read. Perhaps it's possible to make it larger.

As with any change you'd like to make in the output, the way you can make the label larger is to send a message to the object that asks it to do so. For objects of type **GLabel**, the size is controlled by a property called its *font*. These days, you're almost certainly familiar with fonts from the word processing programs you've used. A **font** is simply an encoding that maps characters into images that appear on the screen. In Java, a font is determined by a **family name** (Times, Helvetica, Courier, and so forth), a **style** (bold or italic, for example), and a **point size** (an integer expressing the size of the characters). The **GLabel** class includes a method called **setFont** that takes these properties in the form of a string. For example, you could change the font to Helvetica-24 by adding a line to the run method, as follows:

```
public void run() {
   GLabel label = new GLabel("hello, world");
   label.setFont("Helvetica-24");
   add(label, 100, 75);
}
```

After making this change, the display would look like this:

Much more readable, but perhaps not any more exciting.

As you probably know from using your word processor, it can be fun to play around with different fonts. On the Macintosh, for example, there is a font called **London** that produces Old English script. Thus, if you were to change the **setFont** call to

```
label.setFont("London-36");
```

you might see the output

You might see that, but only on a Macintosh. On most systems, there isn't a font called **London**, and the program would choose some default font instead. If you're interested in writing portable programs that can run on many different architectures, it would be best to avoid nonstandard fonts. In fact, if you want to be as portable as possible, it is best to stick with the following font names:

Serif	A traditional newspaper-style font in which the characters are embellished by various frills called **serifs.** The most common example of a serif font is Times.
SansSerif	A simpler, unadorned font style lacking serifs, of which the most common example is Helvetica.
Monospaced	A typewriter-style font in which all characters are the same width. The most common monospaced font is Courier.
Dialog	The default font for output text in dialogs, which may differ depending on the platform.
DialogInput	The default font for user input in dialogs, which may differ depending on the platform.

The remaining chapters in this text will use only these standard fonts, but as long as the **London** font is already there in the example, it's probably reasonable to show a few additional examples starting from where we are.

What else might you want to change? To generate a fancier display, you might certainly want to change the color of the text by using the method **setColor**, which is defined for all subclasses of **GObject**. Although you can work with a wide variety of colors in Java, the **Color** class in **java.awt** defines a set of standard colors that are sufficient for most purposes. These predefined colors have the following names:

> **BLACK**, **DARK_GRAY**, **GRAY**, **LIGHT_GRAY**, **WHITE**
> **RED**, **PINK**, **ORANGE**, **YELLOW**, **GREEN**, **CYAN**, **BLUE**, **MAGENTA**

Note that these colors are named constants and are therefore written entirely in uppercase in accordance with the standard conventions of Java. Also, because these colors are defined as constants in a separate class, you need to include the name of the defining class, as in **Color.RED**. Finally, you need to be sure that your application has access to the **Color** class itself by importing the **java.awt** package. For example, to change the color of the label to magenta by adding yet another line to the **run** method, like this:

```
public void run() {
   GLabel label = new GLabel("hello, world");
   label.setFont("London-36");
   label.setColor(Color.MAGENTA);
   add(label, 100, 75);
}
```

There is, however, one more method that you might like to call to improve the overall appearance of the display. At this point, the label is in danger of running off the right edge of the window. Things would be considerably more readable if you could move the label to the center. At some level, all you need to do is change the coordinates specified in the **add** method call. The only question is how to figure out what values to use.

The most important aspect of any answer to this question is that *you* shouldn't figure out what the appropriate coordinates are, but rather that the *computer* should figure out those coordinates according to the specifications you set forth in your program. In this case, the desired outcome is that the label be centered in the window. The coordinates at which you draw the label therefore depend not only on the size of the window, but also on the size of the label. In the *x* direction, the label needs to start half of its own width to the left of the center of the window. In the *y* direction, the label needs to be moved down half the distance the characters extend upward from the baseline, which is called their **ascent**. The *x* and *y* values for the initial position of the label can therefore be calculated using the following lines:

```
double x = (getWidth() - label.getWidth()) / 2;
double y = (getHeight() + label.getAscent()) / 2;
```

Look carefully at the expression on the first line. There are two calls to the method **getWidth**, the first of which has no receiver and the second is directed to **label**. If **getWidth** is called with no receiver, it must refer to the object making the call, which is the **FancyHello** program itself. In that context, **getWidth** returns the width of the window in which the program is running. The expression

```
label.getWidth()
```

on the other hand, asks **label** to tell us how wide it is. If you subtract that width from the width of the entire window and divide by 2, you get the coordinate value—at least in the *x* direction—at which the label should start. The calculation in the vertical dimension is similar, but it is important to remember that Java coordinates increase as you move down the window. Thus, to shift the baseline down a little, you need to add the appropriate offset and not subtract it.

The entire set of changes to the original **HelloProgram** is shown in Figure 5-2, which produces the following output:

FIGURE 5-2   **Program to draw a fancier version of Hello World**

```
/*
 * File: FancyHello.java
 * --------------------
 * This program displays a fancier "hello, world" message.
 */

import acm.graphics.*;
import acm.program.*;

public class FancyHello extends GraphicsProgram {

   public void run() {
      GLabel label = new GLabel("hello, world");
      label.setFont("London-36");
      label.setColor(Color.MAGENTA);
      double x = (getWidth() - label.getWidth()) / 2;
      double y = (getHeight() + label.getAscent()) / 2;
      add(label, x, y);
   }

}
```

5.3 Writing your own methods

The methods introduced so far in this book have all been defined as part of some library package. Library methods are certainly useful, but they do not tell the whole story of methods. As a programmer, you will often be content to use library methods without knowing any of the internal details. But you will also want to use methods that are not part of any library. In those cases, you have no choice but to define the methods yourself.

For example, suppose that you have been assigned the task of writing a program that generates a conversion table from the Celsius scale used in most countries to the Fahrenheit scale used in the United States. You will probably want to define a simple conversion method that you can then use in other parts of the program. The computation is relatively easy, because the process of converting one temperature scale to another is simply a matter of applying the formula

$$F = \frac{9}{5} C + 32$$

Format of a method definition

Defining a new method in Java begins by writing its header, as specified in the syntax box on the right. Until you start defining your own classes, the distinction between public and private methods doesn't mean a great deal. At the same time, it is usually best to keep things private if possible, so that any method used only within a single class should be declared as **private**. The *type* element in the pattern represents the type of value the method returns; if the method doesn't need to return a value, the *type* element should be specified

Syntax for a method header:

> *visibility type name* **(***parameters***)**

where:
  visibility is typically **public** or **private**
  type is the type of value returned, or **void**
      if the method returns no value
  name is the name of the method
  parameters is a list of parameter
      declarations

as **void**. The *name* element specifies the name of the method, which may be any identifier formed according to the rules given in Chapter 2. Most importantly, the name should make it easy for anyone reading the program to determine what the method does. The code inside the parentheses, which is labeled *parameters* in this pattern, specifies what arguments the method requires. The parameter list has the same form as variable declarations except that no initial value is assigned. Methods that take more than one argument separate these declarations with commas.

In this example, you are writing a method that converts from Celsius to Fahrenheit. The header line therefore looks like this:

```
private double celsiusToFahrenheit(double c)
```

The header line is then followed by the **method body**, which is always a block and therefore consists of statements enclosed in curly braces. The statements in the block may include variable declarations such as the ones used in the example programs you have already seen.

The **return** statement

If a method returns a result, the statements in the method body must include at least one **return statement**, which specifies the value to be returned. The paradigmatic form for the **return** statement is shown in the syntax box to the right.

In most cases, the **return** statement includes an expression that indicates the value of the result, although that expression is omitted if the method does not return a value. When it is used in the

> Syntax for the **return** statement:
>
>   **return** *expression***;**
>
> where:
>   *expression* is the value to be returned.
>
> If the method has no result, the syntax is:
>
>   **return;**

```
return expression;
```

form, the **return** statement causes the method to return the indicated value immediately. As such, the **return** statement encompasses both of the following English ideas: "I'm done now" and "Here is the answer." In some programming languages, such as Pascal and Fortran, indicating that the execution of a method is complete and specifying its result are separate operations. If you have had experience with such languages, it may take some time to get used to the **return** statement in Java.

The **return** statement completes the list of tools you need to write the implementation of the **celsiusToFahrenheit** method:

```
private double celsiusToFahrenheit(double c) {
    return 9.0 / 5.0 * c + 32;
}
```

To create the table of temperature conversions, you need to create a **ConsoleProgram** subclass whose **run** method calls **celsiusToFahrenheit** for each entry in the table. The complete program to do so is shown in Figure 5-3.

Methods involving internal control structures

Methods are not usually as simple as **celsiusToFahrenheit**. In many cases, calculating a method requires making some tests or writing a loop. Such details add to the complexity of the implementation but do not change its basic form. For example, the **abs**

FIGURE 5-3   **Program to generate a temperature-conversion table**

```java
/*
 * File: TemperatureConversionTable.java
 * -----------------------------------
 * This program creates a table of Celsius to Fahrenheit
 * equivalents using a function to perform the conversion.
 */

import acm.program.*;

public class TemperatureConversionTable extends ConsoleProgram {

/* Lower limit for the table */
   private static final int LOWER_LIMIT = 0;

/* Upper limit for the table */
   private static final int UPPER_LIMIT = 100;

/* Step size for the table */
   private static final int STEP_SIZE = 5;

/* Runs the program */
   public void run() {
      println("Celsius to Fahrenheit table.");
      for (int c = LOWER_LIMIT; c <= UPPER_LIMIT; c += STEP_SIZE) {
         int f = (int) celsiusToFahrenheit(c);
         println(c + "C = " + f + "F");
      }
   }

/*
 * Returns the Fahrenheit equivalent of the Celsius temperature c.
 */
   private double celsiusToFahrenheit(double c) {
      return 9.0 / 5.0 * c + 32;
   }
}
```

---

method in the **Math** class computes the absolute value of its argument, which for the moment you may assume is an integer. But suppose for a moment that the method did not exist. How would you write its implementation? The definition of absolute value indicates that if the argument is negative, the method should return its negation, which is a positive number. If the argument is positive or zero, the method should simply return the argument value unchanged. Thus, you can implement the **abs** method as follows:

```java
private int abs(int n) {
    if (n < 0) {
        return -n;
    } else {
        return n;
    }
}
```

As this implementation shows, a **return** statement can occur anywhere in the method body, and may appear more than once.

Similarly, you could define a method **min** to return the smaller of two floating-point arguments as follows:

```
private double min(double x, double y) {
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

The control structure used within a method can be much more complex than the simple examples above. Suppose that you want to define a method called **factorial** that takes an integer **n** and returns the product of the integers between 1 and **n**. The first several factorials are shown in the following list:

```
factorial(0)    =     1        (by definition)
factorial(1)    =     1   =   1
factorial(2)    =     2   =   1 × 2
factorial(3)    =     6   =   1 × 2 × 3
factorial(4)    =    24   =   1 × 2 × 3 × 4
factorial(5)    =   120   =   1 × 2 × 3 × 4 × 5
factorial(6)    =   720   =   1 × 2 × 3 × 4 × 5 × 6
```

Factorials are usually designated in mathematics using an exclamation point, as in $n!$, and have extensive applications in statistics, combinatorial mathematics, and computer science. A method to compute factorials is a useful tool for solving problems in those domains.

The **factorial** method takes an integer and returns an integer, so its header line looks like this:

```
int factorial(int n)
```

Implementing **factorial**, however, requires some work. As a programming problem, the task of computing a factorial is similar in many respects to adding a list of numbers. In the **AddList** program from Chapter 4, a variable called **total** is declared to keep track of the running total. At the beginning of the program, **total** is initialized to 0. As each new value comes in, it is added to **total** so that **total** continues to reflect the sum of the numbers entered so far. In the current problem, the situation is much the same, except that you have to keep track of a product rather than a sum. To do so, you can:

1. Declare a variable called **result**.
2. Initialize it to 1.
3. Multiply it by each of the integers between 1 and **n**.
4. Return the final value of **result** as the result of the method.

To cycle through each of the integers required in step 3, you need a **for** loop, which begins at 1 and continues until it reaches **n**. The **for** loop will require an index variable, for which the traditional choice of **i** seems quite appropriate. The variable **result** holds the running product, and **i** holds the index.

The implementation of **factorial** is short enough to present all at once without explaining the details step by step:

```
private int factorial(int n) {
   int result = 1;
   for (int i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}
```

Methods that return nonnumeric values

The examples of methods presented so far in this section all return numeric results, but that is by no means necessary. Methods in Java can return values of any data type. For example, if you were writing a program to work with dates, it might be useful to have a method to convert a numeric month between 1 and 12 into the **String** that indicates the corresponding month name between January and December. While numeric values are easier to work with internally (if, for example, you needed to compare two dates to see which came earlier), the output display may be more readable with the traditional English names. To solve this problem, you could define the method **monthName** as follows:

```
private String monthName(int month) {
   switch (month) {
      case  1: return ("January");
      case  2: return ("February");
      case  3: return ("March");
      case  4: return ("April");
      case  5: return ("May");
      case  6: return ("June");
      case  7: return ("July");
      case  8: return ("August");
      case  9: return ("September");
      case 10: return ("October");
      case 11: return ("November");
      case 12: return ("December");
      default: return ("Illegal month");
   }
}
```

   To use this method, you would call **monthName** from some other part of the program and then use **println** to display the result. For example, if the integer variables **month**, **day**, and **year** contain the values 7, 20, and 1969 (the date of the Apollo 11 landing on the moon), the statement

```
println(monthName(month) + " " + day + ", " + year);
```

would generate the output
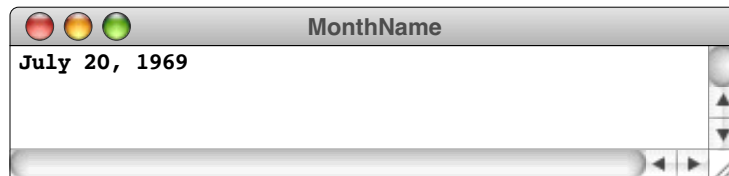
   In the **switch** statement within the **monthName** method, the **return** statements in each **case** clause automatically exit from the entire method and make an explicit **break** statement unnecessary. As indicated in the section on the **switch** statement in Chapter 4, you can avoid a lot of pain in the debugging process if you design your programs so that every **case** clause ends with either a **break** or a **return** statement.

Predicate methods

The examples in the preceding section illustrate that methods can return values of different data types. The method **factorial**, for example, returns a value of type **int**, and the method **monthName** returns a value of type **string**. Although methods in Java can return values of any type, there is one result type that deserves special attention. That type is the data type **boolean**, which was introduced in Chapter 4 and is defined by including the **genlib** library. Methods that return values of type **boolean** are called **predicate methods** and play an important role in modern programming.

Recall that there are only two values of type **boolean**: **true** and **false**. Thus a predicate method—no matter how many arguments it takes or how complicated its internal processing may be—must eventually return one of these two values. The process of calling a predicate method is therefore analogous to asking a yes/no question and getting an answer.

Consider the following method definition, which, given an integer **n**, answers the question "is **n** an even number?":

```
private boolean isEven(int n) {
    return (n % 2 == 0);
}
```

A number is even if there is no remainder when that number is divided by two. If **n** is even, the expression

```
n % 2 == 0
```

therefore has the value **true**, which is returned as the result of the **isEven** method. If **n** is odd, the method returns **false**. Because **isEven** returns a Boolean result, you can use it directly in a conditional context. For example, the following **for** loop uses **isEven** to list all the even numbers between 1 and 100:

```
for (int i = 1; i <= 100; i++) {
    if (isEven(i)) println(i);
}
```

The **for** loop runs through each number, and the **if** statement asks the simple question "is this number even?" If it is, **println** displays it on the screen; if not, nothing happens.

The standard class definitions in Java often include predicate methods. One particularly important example is the **equals** method, which is defined in the universal class **Object** and can therefore be applied to any object. The **equals** method asks whether one object has the same value as another. By contrast, the **==** operator asks whether its operands are precisely the same object, which is not ordinarily as useful.

The distinction between **equals** and **==** is easiest to illustrate with respect to strings. Suppose, for example, that your program contains the following statement to ask the user a simple yes-or-no question:

```
String answer = readLine("Would you like instructions? ");
```

Once the user enters a response, the question is how would you determine whether that answer was **"yes"** or **"no"** or perhaps some other string entirely. The following approach doesn't work:

```
if (answer == "yes") . . .
```

The value read in from the user may well be a string object composed of the characters **'y'**, **'e'**, and **'s'** but it will not be the *same* object as the constant string **"yes"** that appears in the program. What you need to ask instead is whether the strings contain the same sequence of characters, which is precisely what the **equals** method does. Thus, the statement you need to write is

```
if (answer.equals("yes")) . . .
```

Note that this statement uses the receiver syntax and can therefore be interpreted as sending an **equals** message to the **answer** object and then having it send back the result of the comparison.

When new programmers use predicate methods, they often make the errors described in the section on "Avoiding redundancy in Boolean expressions" in Chapter 4. Until you get more experience with Boolean data and predicate methods, you may find yourself tempted to put an **if** statement inside the implementation of **isEven** or to make unnecessary comparisons against **true**, such as

```
if (isEven(i) == true) . . .
```

If you find yourself making such errors, you may want to review the discussion of Boolean data in Chapter 4.

As another example of a predicate method, you could write one that tests whether a given year is a leap year, as follows:

```
boolean isLeapYear(int year) {
   return ( ((year % 4 == 0) && (year % 100 != 0))
            || (year % 400 == 0) );
}
```

You encountered the Boolean expression to determine whether **year** is a leap year in Chapter 4. By taking this expression and putting it into a method, you no longer have to include the entire calculation explicitly to make this test. Once the method is defined, the rest of the program can simply use statements of the form:

```
if (isLeapYear(year)) . . .
```

5.4 Mechanics of the method-calling process

So far in this chapter, you have looked at methods mostly from a holistic perspective. Thinking about methods in this way helps you understand how they are used and what they provide as a programming resource. To develop confidence that the methods you write will work as they should, however, you also need to develop an understanding of how methods operate internally.

As a first step toward understanding the mechanics of methods, consider the program shown in Figure 5-4, which includes both the **factorial** method presented earlier in this

FIGURE 5-4   **Program to generate a table of factorials**

```java
/*
 * File: FactorialTable.java
 * ------------------------
 * This file generates a table of factorials.
 */

import acm.program.*;

public class FactorialTable extends ConsoleProgram {

/* Lower limit for the table */
   private static final int LOWER_LIMIT = 0;

/* Upper limit for the table */
   private static final int UPPER_LIMIT = 10;

/* Runs the program */
   public void run() {
      for (int i = LOWER_LIMIT; i <= UPPER_LIMIT; i++) {
         println(i + "! = " + factorial(i));
      }
   }

/*
 * Returns the factorial of n, which is defined as the
 * product of all integers from 1 up to n.
 */
   private int factorial(int n) {
      int result = 1;
      for (int i = 1; i <= n; i++) {
         result *= i;
      }
      return result;
   }
}
```

chapter and a main program that displays a factorial table. The **FactorialTable** program makes good sense if you look at it as two separate pieces. The main program simply counts from **LOWER_LIMIT** to **UPPER_LIMIT**. On each cycle, it calls **factorial** on the index **i** and then displays the result. By this point, you are accustomed to using the name **i** to indicate an otherwise unremarkable index variable used to count cycles in a **for** loop. No surprises here. The **factorial** method is likewise straightforward. There is a little more going on than in the main program, but not much more. You can easily understand what this method is doing. In particular, you should recognize that **n** is the number whose factorial we're computing, that **result** holds the accumulating product on each cycle, and that **i** is once again an unremarkable index variable used to track the progress of the **for** loop. Thus, each piece of the program makes sense in and of itself.

For new students, confusion arises only when looking at the program as a whole. If you do that without understanding how to think about methods, you can run into some interesting problems. First, there are two variables named **i**, one in the main program and one in the **factorial** method. Each variable is used as a loop index, but the two

variables will have different values. Second—and equally confusing—is the fact that there are other parts of the program in which two different names are used to refer to the same value. In the **run** method, the number whose factorial you are seeking to compute is stored in the loop index **i**. In the **factorial** method, that same value is called **n**. How do you make sense out of all this confusion?

Even if the program is confusing as a whole, it is important to remember that the methods make sense when you look at them one at a time. As long as you look only at **run** or only at **factorial**, none of these points of confusion exist. Each method definition makes sense by itself. That fact—as commonplace as it might seem—is of fundamental importance. As programs grow, there is no way you can comprehend them as a whole. Your only hope of making sense of a large program is to break it down into pieces, each of which is small enough to make sense by itself. In the **FactorialTable** program, the problem of making a list of factorials has been separated into two easy-to-understand pieces.

Parameter passing

To understand how the two pieces of the program work together, you will find it helpful to develop a sense of how Java itself handles the confusion. That it manages to do so is important for us as programmers and enables us to think about the individual methods separately. How does Java make sense of the facts that the same name may be used for different values and that a single conceptual value may be represented by different names? To help you understand the answer to this question, it is useful to introduce a semantic distinction between argument values in the caller and the variables used to hold those values in the context of a method.

When the main program calls **factorial** (which itself appears as one of the arguments to the method **println**), the argument is the expression **i**. When this statement is executed, the effect is to look up the current value of **i** and pass it to the method **factorial**, which has the header line

```
    int factorial(int n)
```

The header line declares a variable named **n** of type **int**—a variable whose purpose is to serve as a placeholder for the actual argument. A variable defined in a method header that serves as a placeholder is called a **formal parameter**.
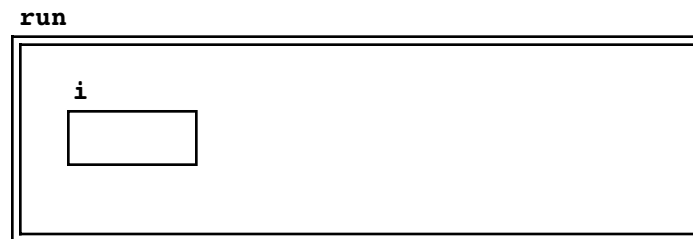
When a method is called, the following steps are taken:

1. The values of each argument are computed as part of the operation of the calling program. Because the arguments are expressions, this computation can involve operators and other methods, all of which are evaluated before the new method is actually called.

2. The value of each argument is copied into the corresponding formal parameter variable. If there is more than one argument, the arguments are copied into the parameters in order; the first argument is copied to the first parameter, and so forth. If necessary, automatic type conversions are performed between the argument values and the formal parameters as in an assignment statement. For example, if a value of type **int** is passed to a method where the parameter is declared as a **double**, the integer is converted into the equivalent floating-point value before it is copied into the parameter variable.

3. The statements in the method body are evaluated until a **return** statement appears.

4. The value of the **return** expression is evaluated and converted, if necessary, to the result type specified for the method.

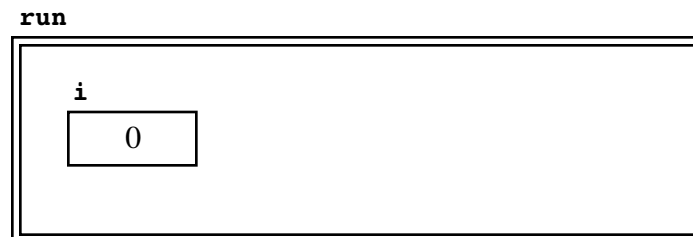5. The calling program continues, with the returned value substituted in place of the call.

Every call to a method results in the creation of a separate set of variables. When variables were represented graphically in Chapter 2 as boxes, all the boxes were enclosed within a larger box representing the method **run**, which was the only method in the program. If you want to follow the computer's operation for a larger program with more than one method, you need to draw a new set of variable boxes each time one method calls another. As was true in the case of the method **run**, there must be one box for every variable that method declares, including the formal parameters. These variables are meaningful only within the program that declares them and are therefore called **local variables**.

For example, when the main program runs in the **FactorialTable** example, you first need to create space for the variables in the method **run**. The method **run** declares only one variable—the loop index **i**—so that the variables for **run** can be represented as follows:

run

```
+-----------------------------------------+
| +-------------------------------------+ |
| |   i                                 | |
| |   +----------------+                | |
| |   |                |                | |
| |   +----------------+                | |
| |                                     | |
| +-------------------------------------+ |
+-----------------------------------------+
```

The double lines around the variable boxes are used to enclose all the variables associated with a particular method call. This collection of variables is called the **frame**—or, for reasons that will soon be apparent, the **stack frame**—for that method.

Assume that **LOWER_LIMIT** is defined to be 0, as it was in the program listing. In this case, on the first cycle through the **for** loop, **i** has the value 0. As before, you can represent this condition in the frame by noting the value inside the box for that variable.

run

```
+-----------------------------------------+
| +-------------------------------------+ |
| |   i                                 | |
| |   +----------------+                | |
| |   |      0         |                | |
| |   +----------------+                | |
| |                                     | |
| +-------------------------------------+ |
+-----------------------------------------+
```

The main program then calls **println** and, as part of evaluating the arguments to **println**, computes the result of the expression

```
factorial(i)
```

To represent the computer's actions in the frame diagram, you begin by looking up the value of **i** in the current frame, where you discover that the value is 0. You must then create a new frame for **factorial**, for which the value 0 is the first (and only) argument.

The **factorial** method has three variables: the formal parameter **n** and the local variables **result** and **i**. Your frame for **factorial** therefore needs three variable cells:

factorial

n	result	i
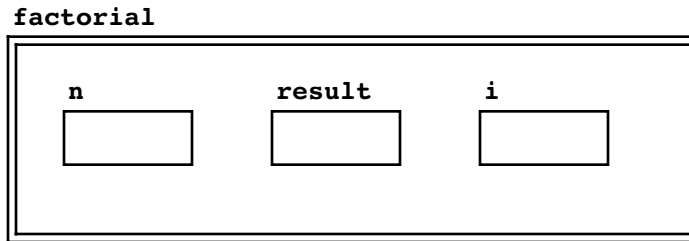
The first thing that happens is that the formal parameter **n** is initialized with the value of the argument, which was 0. The contents of the frame then look like this:
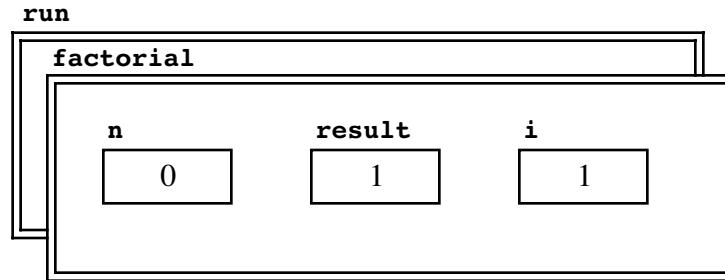
factorial

n	result	i
0		
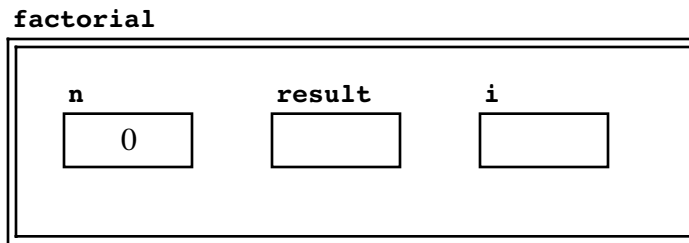
When you create the frame for **factorial**, the frame for **run** does not go away entirely. Instead that frame is set aside temporarily until the operation of **factorial** is complete. To indicate this situation in the conceptual model represented by the box diagram, the best technique is to draw each new frame diagram on an index card and then to place the new card on top of the card for the previous frame, thereby covering it up. For example, when you call the method **factorial**, the index card representing the **factorial** frame goes on top of the frame for **run**.

run

factorial

n	result	i
0		
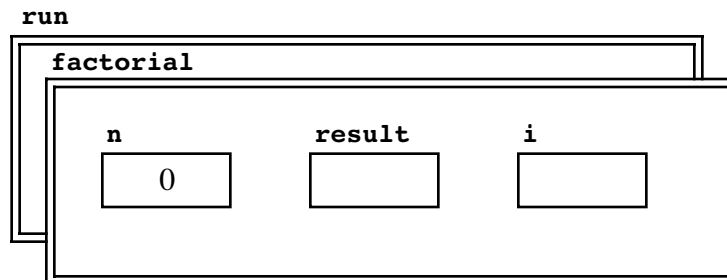
As the diagram shows, the entire set of frames forms a stack with the most recent frame on top, which is the origin of the term *stack frame*. The frame for **run** is still there; you just can't see any of it as long as the method **factorial** is active. In particular, the name **i** no longer refers to the variable declared in **run** but to the variable named **i** in **factorial**.

The next step in the process is to execute the body of **factorial** by running through each of its steps in the current frame. The variable **result** is initialized to 1, and the program reaches the **for** loop. In the **for** loop, the variable **i** is initialized to 1, but since the value is already larger than **n**, the body of the **for** loop is not executed at all. Thus, when the program reaches the **return** statement, the frame looks like this:

run

factorial

n	result	i
0	1	1

When **factorial** returns, the value of **result** is passed back to the caller as the result of the method. Returning from a method also implies throwing away its frame, exposing once again the variables in **run**:

run

i

0
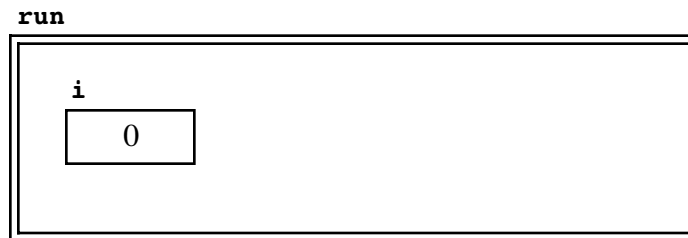
The result, 1, is then passed to the **println** method. The **println** method goes through the same process; the details of that operation, however, are hidden from you because you don't know how **println** works internally. Eventually, **println** displays the value 1 on the screen and then returns, after which the method **run** goes on to execute the next **for** loop cycle.

Calling methods from within other methods

Much of the power of methods comes from the fact that once a method has been defined, you can use it not only in the context of a main program, but also as a tool to implement other methods that can be used as more sophisticated tools. These methods can then be called from other methods, and so on, creating an arbitrarily complex hierarchy.

Suppose you have a group of **n** distinct objects on a table in front of you. From that group of objects, you would like to select **k** objects. The question you want to answer is: How many different ways are there to select **k** objects out of the original collection of **n** distinct objects?

To make the problem concrete, suppose that the objects on the table are five U.S. coins: a penny, a nickel, a dime, a quarter, and a half-dollar. You want to know how many different ways there are to take, for example, two coins from the table. You could take the penny and the nickel, the penny and the dime, the nickel and the quarter, or any of several other combinations. If you list all the possibilities, you discover that there are 10 different combinations, as follows:

penny + nickel	nickel + quarter
penny + dime	nickel + half-dollar
penny + quarter	dime + quarter
penny + half-dollar	dime + half-dollar
nickel + dime	quarter + half-dollar

In this example, **n** is 5 and **k** is 2. The solution can be expressed as a method of the values **n** and **k**. That method comes up frequently in probability and statistics and is called the **combinations function**, which is often written in mathematics as

$$\binom{n}{k}$$

or, in functional notation, as

C(n, k)

As it turns out, the combinations method has a simple definition in terms of factorials:

$$\text{C}(n, k) \quad = \quad \frac{n!}{k! \times (n{-}k)!}$$

For example, you can verify that c(5, 2) is indeed 10 by working out the mathematics:

$$\text{C}(5, 2) \quad = \quad \frac{5!}{2! \times 3!}$$

$$= \quad \frac{120}{2 \times 6}$$

$$= \quad 10$$

If you want to implement this method in Java, it would probably be best to use a longer name than the single letter C used in mathematics. The one-character name might well cause confusion, if for no other reason than that the fact that its name is uppercase suggests a constant—possibly the speed of light. As a general rule, method names used in this text tend to be longer and more expressive than variable names. Method calls often appear in parts of the program that are far removed from the point at which those methods are defined. Since the definition may be hard to locate in a large program, it is best to choose a method name that conveys enough information about the method so that the reader does not need to look up the definition. Local variables, on the other hand, are used only within the body of a single method, and it is therefore easier to keep track of what they mean. In the interest of having the name of the combinations method make sense immediately when anyone looks at it, we will use the name **combinations** as the method name, rather than the letter **c**.

To implement the **combinations** method using the definition based on factorials, you can take advantage of the fact that you already have an implementation of **factorial**. Given **factorial**, the implementation of **combinations** is a straightforward translation of its mathematical definition:

```
private int combinations(int n, int k) {
   return factorial(n) / (factorial(k) * factorial(n - k));
}
```

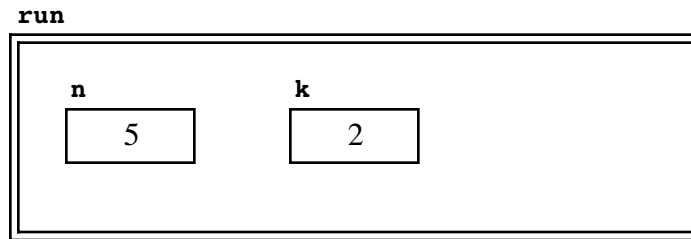You can then write a simple main program to test the **combinations** method as follows:

```
public void run() {
   int n = readInt("Enter number of objects in the set (n): ");
   int k = readInt("Enter number to be chosen (k): ");
   println("C(" + n + ", " + k + ") = " + combinations(n, k));
}
```

The complete **Combinations** program is shown in Figure 5-5. The following output illustrates one possible sample run of the program:

```
●●●                    Combinations
Enter number of objects in the set (n): 5
Enter number to be chosen (k): 2
C(5, 2) = 10
```
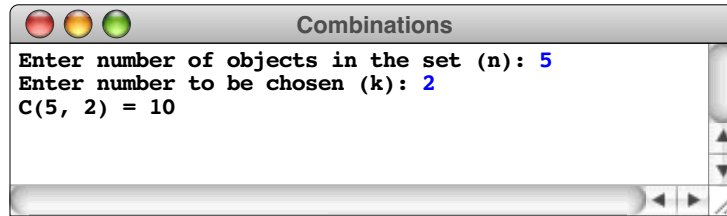
   What happens inside the computer when this program runs? Just as in the factorial example, a frame is created for the method **run**, which now declares two variables, **n** and

---

FIGURE 5-5   **Program to compute the combinations function C(n, k)**

```java
/*
 * File: Combinations.java
 * -----------------------
 * This program computes the mathematical combinations function
 * C(n, k), which is the number of ways of selecting k objects
 * from a set of n distinct objects.
 */

import acm.program.*;

public class Combinations extends ConsoleProgram {

/* Runs the program */
   public void run() {
      int n = readInt("Enter number of objects in the set (n): ");
      int k = readInt("Enter number to be chosen (k): ");
      println("C(" + n + ", " + k + ") = " + combinations(n, k));
   }

/*
 * Returns the mathematical combinations function C(n, k),
 * which is the number of ways of selecting k objects
 * from a set of n distinct objects.
 */
   private int combinations(int n, int k) {
      return factorial(n) / (factorial(k) * factorial(n - k));
   }

/*
 * Returns the factorial of n, which is defined as the
 * product of all integers from 1 up to n.
 */
   private int factorial(int n) {
      int result = 1;
      for (int i = 1; i <= n; i++) {
         result *= i;
      }
      return result;
   }
}
```
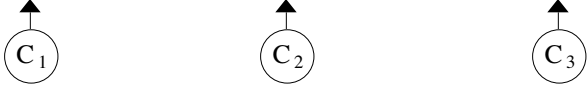
k. After the user enters the two values and the program reaches the **println** statement, the variables in the frame have the following values:

```
            run
┌──────────────────────────────────────────┐
│ ┌──────────────────────────────────────┐ │
│ │                                      │ │
│ │   n                  k               │ │
│ │  ┌─────────┐        ┌─────────┐      │ │
│ │  │    5    │        │    2    │      │ │
│ │  └─────────┘        └─────────┘      │ │
│ │                                      │ │
│ └──────────────────────────────────────┘ │
└──────────────────────────────────────────┘
```

To execute the **println** statement, the computer must evaluate the call to the **combinations** method, which results in the creation of a new frame that overlays the previous one:

```
        run
    ┌────────────────────────────────────────┐
    │ combinations                           │
    │ ┌────────────────────────────────────┐ │
    │ │                                    │ │
    │ │   n                  k             │ │
    │ │  ┌─────────┐        ┌─────────┐    │ │
    │ │  │    5    │        │    2    │    │ │
    │ │  └─────────┘        └─────────┘    │ │
    │ │                                    │ │
    └─│                                    │ │
      └────────────────────────────────────┘ │
        └──────────────────────────────────────┘
```
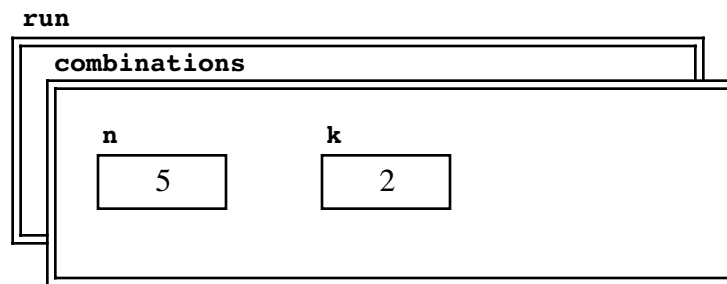
   In this example, which has more method calls than the factorial example does, each new frame must record precisely what the program was doing before it made the call. Here, for example, the call to **combinations** comes from the last line in **run**, as indicated by the tag R_1 in the program text that follows:

```
public void run() {
    int n = readInt("Enter number of objects in the set (n): ");
    int k = readInt("Enter number to be chosen (k): ");
    println("C(" + n + ", " + k + ") = " + combinations(n, k));
}
                                                          ↑
                                                        ( R₁ )
```

When the computer executes a new method call, it keeps track of where execution should continue in the calling program once this call is completed. The point at which execution should continue is called the **return address**, which is represented in these diagrams using a circled tag. To help you remember where you are in the execution of the program, you need to record in the frame diagram the point from which the call was made, like this:

```
        run
    ┌────────────────────────────────────────┐
    │ combinations                           │
    │ ┌────────────────────────────────────┐ │
    │ │                                    │ │
    │ │   n                  k             │ │
    │ │  ┌─────────┐        ┌─────────┐    │ │
    │ │  │    5    │        │    2    │    │ │
    │ │  └─────────┘        └─────────┘    │ │
    │ │                                    │ │
    └─│  Returning to ( R₁ )               │ │
      └────────────────────────────────────┘ │
        └──────────────────────────────────────┘
```
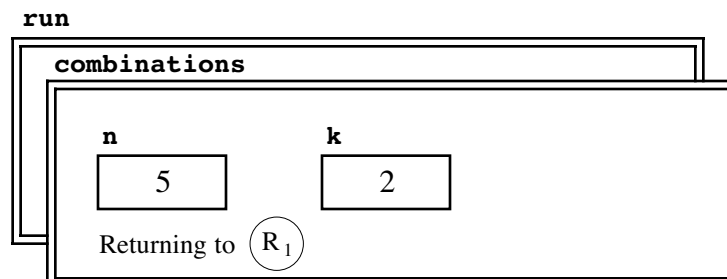
Once the new frame has been created, the program begins to execute the body of the **combinations** method, which is reprinted with each call to **factorial** noted with a tag as follows:

```
private int combinations(int n, int k) {
    return factorial(n) / (factorial(k) * factorial(n - k));
}
```

C₁   C₂   C₃

To execute this statement, the computer must make the three indicated calls to the method **factorial**. According to the rules specified by Java, the compiler must make these calls in left-to-right order. The first call requests the computer to calculate **factorial(n)** and results in the creation of the following frame:
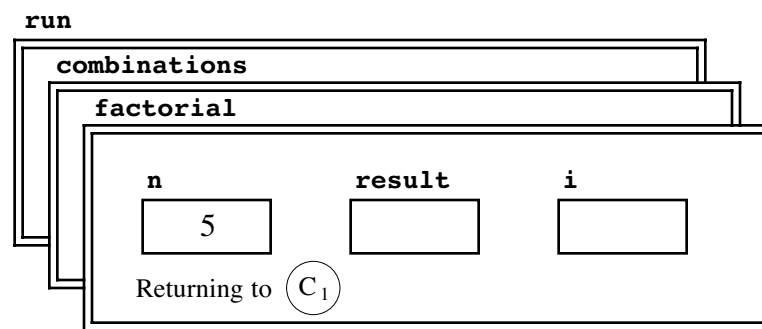
The **factorial** method runs through its operation just as it did before and returns the value 120 to the point in its caller indicated by the tag C_1. The frame for the **factorial** method disappears, and you are left in the frame for **combinations** ready to go on to the next phase of the computation. You can illustrate the current state of things by going back and filling in the returned value in place of the original call like this:

```
return  120  / (factorial(k) * factorial(n - k));
```

C₂   C₃

The box around 120 in the diagram indicates that the enclosed value is not part of the program but the result of a previous computation.

From this point, the computer goes on to evaluate the second call to **factorial**, where the argument is **k**. Since **k** has the value 2, this call causes the following frame to be created:
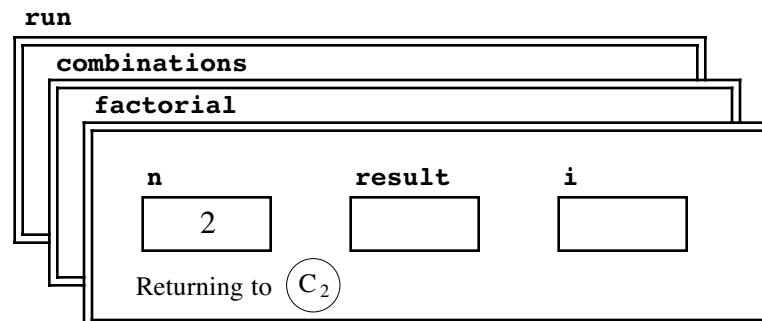
Once again, the **factorial** method completes its operation without making further calls, and the method returns to point C_2 with the value of 2!, which is 2. Inserting this value into the expression that records the result so far shows the computation in the following state:

```
return  120  / ( 2  * factorial(n - k));
```
                                          C_3

There is now only one more call to make, which begins by evaluating the argument expression **n - k**. Given the values of **n** and **k** in the **combinations** frame, the argument expression has the value 3, which leads to the creation of another frame:

run

combinations

factorial

n	result	i
3		

Returning to C_3
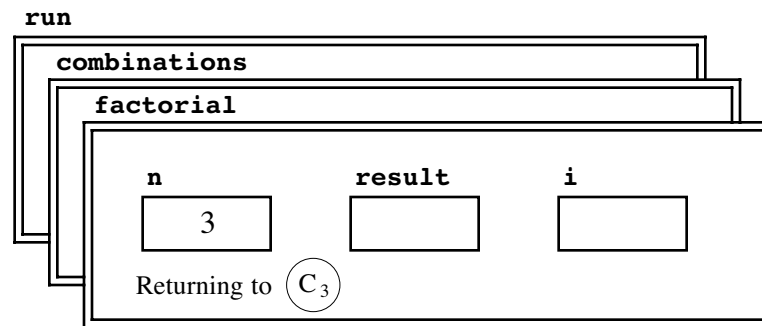
From this point, **factorial** calculates the value of 3! and returns the value 6 to position C_3 in the caller, resulting in the following position:

```
return  120  / ( 2  *  6 );
```

The **combinations** method now has all the values it needs to calculate the result, which is 10. To find out what to do with this result, you need to consult the **combinations** frame, which is once again the top frame on the stack:

run

combinations

n	k
5	2

Returning to R_1
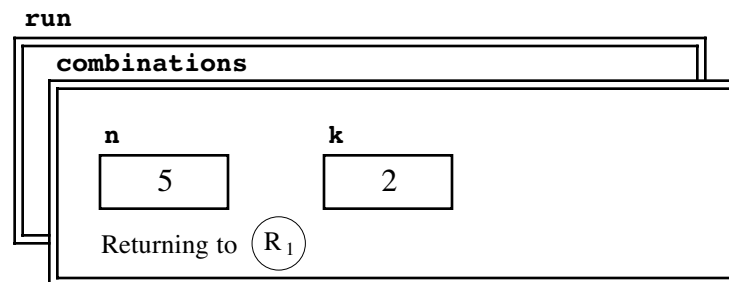
The frame indicates that the program should take the return value and substitute it in place of the call at R_1 in the method **run**. If you take 10 and substitute it for the call to **combinations** in **println** statement, you get the following state:

```
println("C(" + n + ", " + k + ") = " +  10 );
```

Given this result, **println** can happily generate the desired output.

This exercise of going through all the internal details is intended to help you understand the method-calling mechanism in Java. You might find it helpful to trace through your own programs once or twice at this level of detail, but you should not make a habit of it. Instead, you should learn to think about methods more informally and try to develop an intuitive sense of how they work. When a program calls a method, the method performs its operation and the program then continues from the point at which the call was made. If the method returns a result, the calling program is free to use that result in subsequent computation. As a programmer, you need to get to the point where you feel comfortable thinking about the process without worrying about the details. The computer, after all, is taking care of them for you.

5.5 Algorithmic methods

In addition to their role as a tool for simplifying programs by allowing you to refer to a set of statements using a single name, methods are important to programming because they provide a basis for the implementation of algorithms, which were introduced briefly in Chapter 1. An algorithm is an abstract strategy; writing a method is the conventional way to express that algorithm in the context of a programming language. Thus, when you want to implement an algorithm as part of a program, you will typically write a method—which may in turn call other methods to handle part of its work—to carry out that algorithm.

Although you have seen several simple algorithms implemented in the context of the sample programs, you have not had a chance to focus on the nature of the algorithmic process itself. Most of the programming problems you have seen so far are simple enough that the appropriate solution technique springs immediately to mind. As problems become more complex, however, their solutions require more thought, and you will need to consider more than one strategy before writing the final program.

As an illustration of how algorithmic strategies take shape, the sections that follow consider two solutions to a problem from classical mathematics, which is finding the greatest common divisor of two integers. Given two integers x and y, the **greatest common divisor** (or **gcd** for short) is the largest integer that divides evenly into both. For example, the gcd of 49 and 35 is 7, the gcd of 6 and 18 is 6, and the gcd of 32 and 33 is 1.

Suppose that you have been asked to write a method that accepts the integers x and y as input and returns their greatest common divisor. From the caller's point of view, what you want is a method `gcd(x, y)` that takes two integers as arguments and returns another integer that is their greatest common divisor. The header line for this method is therefore

```
int gcd(int x, int y)
```

How might you go about designing an algorithm to perform this calculation?

The "brute force" approach

In many ways, the most obvious approach to calculating the gcd is simply to try every possibility. To start, you simply "guess" that `gcd(x, y)` is the smaller of `x` and `y`, because any larger value could not possibly divide evenly into a smaller number. You then proceed by dividing your guess into `x` and `y` and seeing if it divides evenly into both. If it does, you have the answer; if not, you subtract 1 from your guess and try again. A strategy that tries every possibility is often called a **brute force approach.**

The brute-force approach to calculating the `gcd` function looks like this in Java:

```
int gcd(int x, int y) {
    int guess = Math.min(x, y);
    while (x % guess != 0 || y % guess != 0) {
        guess--;
    }
    return guess;
}
```

Before you decide that this implementation is in fact a valid algorithm for computing the **gcd** function, you must ask yourself several questions about the code. Will the brute-force implementation of **gcd** always give the correct answer? Will it always terminate, or might the method continue forever?

To see that the program gives the correct answer, you need to look at the condition in the **while** loop

```
x % guess != 0 || y % guess != 0
```

As always, the **while** condition indicates under what circumstances the loop will continue. To find out what condition causes the loop to terminate, you have to negate the **while** condition. Negating a condition involving **&&** or **||** can be tricky unless you remember how to apply De Morgan's law, which was introduced in the section on "Logical operators" in Chapter 4. De Morgan's law indicates that the following condition must hold when the **while** loop exits:

```
x % guess == 0 && y % guess == 0
```

From this condition, you can see immediately that the final value of **guess** is certainly a common divisor. To recognize that it is in fact the greatest common divisor, you have to think about the strategy embodied in the **while** loop. The critical factor to notice in the strategy is that the program counts *backward* through all the possibilities. The greatest common divisor can never be larger than **x** (or **y**, for that matter), and the brute-force search therefore begins with that value. If the program ever gets out of the **while** loop, it must have already tried each value between **x** and the current value of **guess**. Thus, if there were a larger value that divided evenly into both **x** and **y**, the program would already have found it in an earlier iteration of the **while** loop.

To recognize that the method terminates, the key insight is that the value of **guess** must eventually reach 1, even if no larger common divisor is found. At this point, the **while** loop will surely terminate, because 1 will divide evenly into both **x** and **y**, no matter what values those variables have.

Euclid's algorithm

Brute force is not, however, the only effective strategy. In fact, the brute-force algorithm used in the **gcd** implementation presented in the preceding section is a poor choice if you are at all concerned with efficiency. Consider what happens, for example, if you call the method with the integers 1,000,005 and 1,000,000. The brute-force algorithm will run through the body of the **while** loop almost a million times before it comes up with 5—an answer that you can easily determine just by thinking about the two numbers.

What you need to find is an algorithm that is guaranteed to terminate with the correct answer but that requires fewer steps than the brute-force approach. This is where cleverness and a clear understanding of the problem pay off. Fortunately, the necessary creative insight has already been supplied by the Greek mathematician Euclid, whose

Elements (book 7, proposition II) contains an elegant solution to this problem. In modern English, Euclid's algorithm can be described as follows:

1. Divide **x** by **y** and compute the remainder; call that remainder **r**.
2. If **r** is zero, the procedure is complete, and the answer is **y**.
3. If **r** is not zero, set **x** equal to the old value of **y**, set **y** equal to **r**, and repeat the entire process.

You can easily translate this algorithmic description into the following Java code:

```java
int gcd(int x, int y) {
    int r = x % y;
    while (r != 0) {
        x = y;
        y = r;
        r = x % y;
    }
    return y;
}
```

This implementation of the **gcd** method also correctly finds the greatest common divisor of two integers. It differs from the brute-force implementation in two respects. On the one hand, it computes the result much more quickly. On the other, it is more difficult to prove correct.

Defending the correctness of Euclid's algorithm

Although a formal proof of correctness for Euclid's algorithm is beyond the scope of this book, you can easily get a feel for how the algorithm works by adopting the mental model of mathematics the Greeks used. In Greek mathematics, geometry held center stage, and numbers were thought of as distances. For example, when Euclid set out to find the greatest common divisor of two whole numbers, such as 55 and 15, he framed the problem as one of finding the longest measuring stick that could be used to mark off each of the two distances involved. Thus, you can visualize the specific problem by starting out with two sticks, one 55 units long and one 15 units long, as follows:

The problem is to find a new measuring stick that you can lay end to end on top of each of these sticks so that it precisely covers each of the distances, **x** and **y**.

Euclid's algorithm begins by marking off the large stick in units of the shorter one:

Unless the smaller number is an exact divisor of the larger one, there is some remainder, as indicated by the shaded section of the upper stick. In this case, 15 goes into 55 three times with 10 left over, which means that the shaded region is 10 units long. The fundamental insight that Euclid had is that the greatest common divisor for the original two distances must also be the greatest common divisor of the length of the shorter stick and the distance represented by the shaded region in the diagram.

Given this observation, you can solve the original problem by reducing it to a simpler problem involving smaller numbers. Here, the new numbers are 15 and 10, and you can find their greatest common divisor by reapplying Euclid's algorithm. You start by representing the new values, x' and y', as measuring sticks of the appropriate length. You then mark off the larger stick in units of the smaller one.
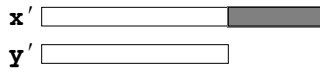
Once again, this process results in a leftover region, which this time has length 5. If you then repeat the process one more time, you discover that the shaded region of length 5 is itself the common divisor of x' and y' and, therefore, by Euclid's proposition, of the original numbers x and y. That this new value is indeed a common divisor of the original numbers is demonstrated by the following diagram:

Euclid supplies a complete proof of his proposition in the *Elements*. If you are intrigued by how mathematicians thought about such problems almost 2500 years ago, you may find it interesting to look up the original source.

Comparing the efficiency of the two algorithms

To illustrate the difference in efficiency of the two algorithmic strategies for computing the greatest common divisor, consider once again the integers 1,000,005 and 1,000,000. To find the greatest common divisor of these two integers, the brute-force algorithm requires a million steps; Euclid's algorithm requires only two. At the beginning of Euclid's algorithm, x is 1000005, y is 1000000, and r is set to 5 during the first cycle of the loop. Since the value of r is not 0, the program sets x to 1000000, sets y to 5, and starts again. On the second cycle, the new value of r is 0, so the program exits from the **while** loop and reports that the answer is 5.

The two strategies for computing greatest common divisors presented in this chapter offer a clear demonstration that the choice of algorithm can have a profound effect on the efficiency of the solution. If you continue your study of computer science, you will learn how to quantify such differences in performance along with several general approaches for improving algorithmic efficiency.

Summary

In this chapter, you learned about *methods*, which enable you to refer to an entire set of operations by using a simple name. By allowing the programmer to ignore the internal details and concentrate only on the effect of a method as a whole, methods provide a critical tool for reducing the conceptual complexity of programs.

The important points introduced in this chapter include:

- A *method* consists of a set of program statements that have been collected together and given a name. Other parts of the program can then *call* that method, possibly passing it information in the form of *arguments* and receiving a result *returned* by that method.
- A method that returns a value must have a **return** statement that specifies the result. Methods may return values of any type.

- Methods in an object-oriented language like Java are often applied to other objects. When that occurs, you need to specify a receiver as part of the method call, like this:

    receiver . *name* (*arguments*)

- Methods that return Boolean values are called *predicate methods* and play an important role in programming.
- Within the body of a method, the variables that act as placeholders for the argument values are called *formal parameters*.
- Variables declared with a method are local to that method and cannot be used outside of it. Internally, all the variables declared within a method, including the parameters, are stored together in a *stack frame*.
- When a method returns, it continues from precisely the point at which the call was made. The computer refers to this point as the *return address* and keeps track of it in the stack frame.
- Because methods tie together a collection of statements so as to have a specific effect, methods provide the standard framework for expressing algorithms in Java.
- There are often many different algorithms to solve a particular problem. Choosing the algorithm that best fits the application is an important part of your task as a programmer.

Review questions

1. Explain in your own words the difference between a method and a program.

2. Define the following terms as they apply to methods: *call*, *argument*, *return*.

3. What is the difference between passing information to a method by using arguments and reading input data using methods like `readInt`? When would each action be appropriate?

4. How do you specify the result of a method in Java?

5. Can there be more than one `return` statement in the body of a method?

6. How do you indicate that you want to apply a method to another object?

7. Why has it been unnecessary to specify receivers in the programs presented in the earlier chapters?

8. Why was it unnecessary to include a `break` statement at the end of each `case` clause in the `monthName` method presented in the chapter?

9. What is a predicate method?

10. How do you tell whether two strings contain the same characters?

11. What is the relationship between arguments and formal parameters?

12. Variables declared within a method are said to be local variables. What is the significance of the word *local* in this context?

13. What does the term *return address* mean?

14. What is a brute-force algorithm?

15. Use Euclid's algorithm to compute the greatest common divisor of 7735 and 4185. What values does the local variable **r** take on during the calculation?

16. In the examples of Euclid's algorithm to calculate **gcd(x, y)** that appear in this chapter, **x** is always larger than **y**. Does this condition matter? What happens if **x** is smaller than **y**?

Programming exercises

1. Write a program that displays the value of the mathematical constant

$$\phi = \frac{1 + \sqrt{5}}{2}$$

This constant ϕ is called the *golden ratio*. Classical mathematicians believed that this number represented the most aesthetically pleasing ratio for the dimensions of a rectangle, but it also turns up in computational mathematics.

2. In high-school algebra, you learned that the standard quadratic equation

$$ax^2 + bx + c = 0$$

has two solutions given by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The first solution is obtained by using + in place of ±; the second is obtained by using – in place of ±.

Write a Java program that accepts values for **a**, **b**, and **c**, and then calculates the two solutions. If the quantity under the square root sign is negative, the equation has no real solutions, and your program should display a message to that effect. You may assume that the value for **a** is nonzero. Your program should be able to duplicate the following sample run:

```
Quadratic
Enter coefficients for the quadratic equation:
a: 1
b: -5
c: 6
The first solution is 3
The second solution is 2
```
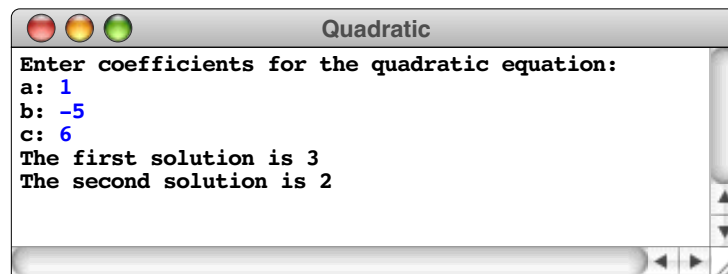
3. The Fibonacci sequence, in which each new term is the sum of the preceding two, was introduced in Chapter 4, exercise 9. Rewrite the program requested in that exercise, changing the implementation so that your program calls a method **fibonacci(n)** to calculate the nth Fibonacci number. In terms of the number of mathematical calculations required, is your new implementation more or less efficient that the one you used in Chapter 4?

4. Write a method **raiseIntToPower** that takes two integers, **n** and **k**, and returns n^k. Use your method to display a table of values of 2^k for all values of **k** from 0 to 10.

5. Write a method **raiseRealToPower** that takes a floating-point value **x** and an integer **k** and returns x^k. Implement your method so that it can correctly calculate the result when **k** is negative, using the relationship

$$x^{-k} \;=\; \frac{1}{x^k}$$

Use your method to display a table of values of π^k for all values of **k** from –4 to 4.

6. Write a method **nDigits(n)** that returns the number of digits in the integer **n**, which you may assume is positive. Design a main program to test your method. For hints about how to write this program, you might want to look back at the **DigitSum** program that was given in Figure 4-6.

7. Write a predicate method **isPerfectSquare(n)** that returns **true** if the integer **n** is a perfect square. Remember that the method **Math.sqrt** returns a **double**, which is therefore only an approximation of the actual square root.

8. Write a predicate method **askYesNoQuestion(str)** that prints out the string **str** as a question for the user and then waits for a response. If the user enters the string **"yes"**, the **askYesNoQuestion** method should return **true**; if the user enters **"no"**, the method should return false. If the user enters anything else, **askYesNoQuestion** should remind the user that it is seeking a yes-or-no answer and then repeat the question. For example, if the program includes the statement

```
if (askYesNoQuestion("Would you like instructions")) . . .
```

the interaction with the user might look like this:
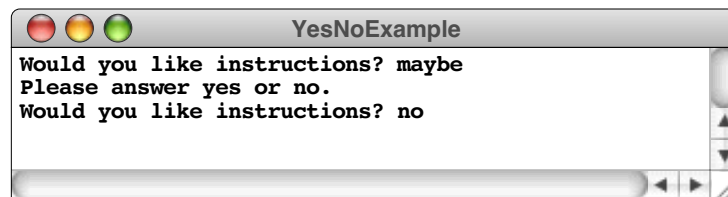
9. The values of the **combinations** method used in the text are often displayed in the form of a triangle using the following arrangement:

<div align="center">

C(0,0)

C(1,0)   C(1,1)

C(2,0)   C(2,1)   C(2,2)

C(3,0)   C(3,1)   C(3,2)   C(3,3)

C(4,0)   C(4,1)   C(4,2)   C(4,3)   C(4,4)

</div>

and so on. This figure is called Pascal's Triangle after the seventeenth-century French mathematician Blaise Pascal, who described it, even though it was known by Chinese mathematicians over 2000 years ago. Pascal's Triangle has the interesting property that every interior entry is the sum of the two entries above it.

Write a Java **GraphicsProgram** that uses **GLabel** objects to display the first eight rows of Pascal's Triangle.

10. An integer greater than 1 is said to be **prime** if it has no divisors other than itself and one. The number 17, for example, is prime, because it is divisible only by 1 and 17. The number 91, however, is not prime because it is divisible by 7 and 13. Write a predicate method **isPrime(n)** that returns **true** if the integer **n** is prime, and **false** otherwise. As an initial strategy, implement **isPrime** using a brute-force algorithm that simply tests every possible divisor. Once you have that version working, try to come up with improvements to your algorithm that increase its efficiency without sacrificing its correctness.

11. Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of *n* is any divisor less than *n* itself). They called such numbers **perfect numbers**. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14.

Write a predicate method **isPerfect(n)** that returns **true** if the integer **n** is perfect, and **false** otherwise. Test your implementation by writing a main program that uses the **isPerfect** method to check for perfect numbers in the range 1 to 9999 by testing each number in turn. Whenever it identifies a perfect number, your program should display that number on the screen. The first two lines of output should be 6 and 28. Your program should find two other perfect numbers in that range as well.

12. Although Euclid's algorithm and the problem of finding perfect numbers from the preceding exercise are both drawn from the domain of mathematics, the Greeks were fascinated with algorithms of other kinds as well. In Greek mythology, for example, Theseus of Athens escapes from the Minotaur's labyrinth by taking in a ball of string, unwinding it as he goes along, and then following the path of string back to the exit. Theseus's strategy represents an algorithm for escaping from a maze, but it is not the only algorithm he could have used to solve this problem. For example, if a maze has no internal loops, you can always escape by keeping your right hand against a wall at all times. This algorithm is called the **right-hand rule.**

For example, imagine that Theseus is in the maze shown below at the position marked by the Greek letter theta (Θ):

To get out, Theseus walks along the path shown by the dotted line in the next diagram, which he can do without taking his right hand off the wall.

Suppose you have been asked to write a program for a robot named Theseus to escape from a maze. You have access to a library that contains these methods:

```
void moveForward();      /* Move forward to the next square   */
void turnRight();        /* Turn right without moving         */
void turnLeft();         /* Turn left without moving          */
boolean isFacingWall();  /* True if Theseus is facing a wall  */
boolean isOutside();     /* True if Theseus has left the maze */
```

Use these methods to write an algorithmic method **solveMaze** that implements the right-hand rule.

Chapter 6
Objects and Classes

To beautify life is to give it an object.

— José Martí, *On Oscar Wilde*, 1882

Kristen Nygaard (1926-2002)
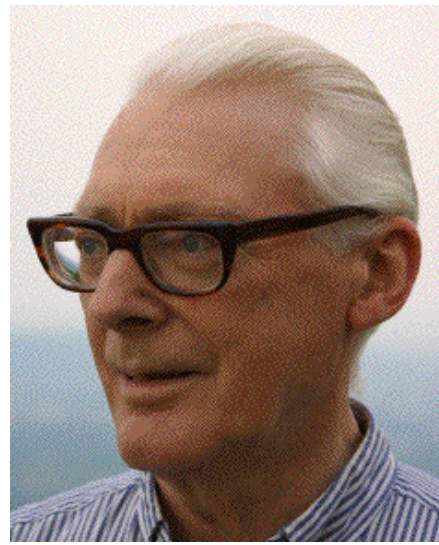
Ole-Johan Dahl (1931-2002)

Norwegian computer scientists Kristen Nygaard and Ole-Johan Dahl developed the central ideas of object-oriented programming more than 40 years ago as part of their work on the programming language SIMULA. Early versions of SIMULA appeared in the early 1960s, but the stable version of the language that brought these concepts to the attention of the world appeared in 1967. The initial work on SIMULA was carried out at the Norwegian Computing Center, a state-f156way focusing on developing better software-engineering techniques. Both later joined the faculty at the University of Oslo. Although their work took several decades to become established in the industry, interest in object-oriented techniques has grown considerably in the last two decades, particularly after the release of modern object-oriented languages like C++ and Java. For their contributions, Nygaard and Dahl received both the 2001 Turing Award from the Association for Computing Machinery and the John von Neumann Medal for the same year from the Institute of Electrical and Electronic Engineers.

After starting off with a holistic look at programs in Chapter 2, the three most recent chapters have taken a reductionistic tack, teaching you how to use the fundamental building blocks of programming—expressions, statements, and methods—starting from the ground up. In a way, this chapter extends that progression by looking at objects and classes, which form the next level in the hierarchical structure of the Java language. At the same time, it is important to recognize that objects and classes are the primary tools for thinking about programs in a high-level, abstract way. In its pursuit of that high-level perspective, this chapter represents a return to the holistic style of Chapter 2.

To emphasize the holistic perspective, this chapter begins with an example that illustrates the use of classes and is also useful in its own right—the **RandomGenerator** class in the **acm.util** package. The discussion of **RandomGenerator** focuses on how one uses existing classes in a program, deferring the details of writing your own classes to the next two sections. Section 6.2 introduces a very simple **Stoplight** class that models the behavior of a traffic light. Section 6.3 then presents a more substantial example in the form of a **Rational** class used to represent rational numbers.

Although being able to adopt a holistic perspective about classes and objects is essential to using them effectively, that perspective is usually not sufficient to understand how objects behave in practice. For most students, it is also important to develop a mental model of objects that allows you to visualize how they are represented inside the machine. To avoid cluttering the high-level vision with too much detail, this book defers those issues of internal representation to Chapter 7.

6.1 Using the RandomGenerator class

Until now, every program in this text has behaved **deterministically**, which means that its actions are predictable given any set of input values. The behavior of such programs is repeatable. If a program produces one result when you run it today, it will produce the same result tomorrow.

In some programming applications, such as games or simulations, it is important that the behavior of your programs not be so predictable: a computer game that always had the same outcome would be boring. In order to build a program that behaves randomly, you need some mechanism for representing a random process, such as flipping a coin or tossing a die, in the context of your programs. Programs that simulate such random events are called **nondeterministic** programs.

Partly because early computers were used primarily for numerical applications, the idea of generating randomness using a computer is often expressed in terms of being able to generate a **random number** in a particular range. From a theoretical perspective, a number is random if there is no way to determine in advance what value it will have among a set of equally probable outcomes. For example, rolling a die generates a random number between 1 and 6. If the die is fair, there is no way to predict which number will come up. The six possible values are equally likely.

Although the idea of a random number makes intuitive sense, it is a difficult notion to represent inside a computer. Computers operate by following a sequence of instructions in memory and therefore function in a deterministic mode. If a number is generated by a deterministic process, any user should be able to work through that same set of rules and anticipate the computer's response. This situation seems paradoxical. How is it possible for a computer to generate unpredictable results by following a deterministic set of rules?

Pseudorandom numbers

In almost all cases, computers today sidestep this paradox by giving up on the notion of true randomness. Java programs that use random numbers in fact compute those numbers

using a deterministic procedure. Although doing so means that a user could, in theory, follow the same set of rules and anticipate the computer's response, the strategy works in practice because no one actually bothers to perform those calculations. In most practical applications, it doesn't matter if the numbers *are* random; all that matters is that the numbers *appear* to be random. For numbers to appear random, they should (1) behave like random numbers from a statistical point of view and (2) be sufficiently difficult to predict in advance that no user would bother. "Random" numbers generated by an algorithmic process inside a computer are formally referred to as **pseudorandom numbers** to underscore the fact that no truly random activity is involved. Informally, however, it is easier to call them **random numbers,** despite the fact that this is not entirely accurate in some philosophical sense.

The techniques for generating random numbers tend to be simple in terms of their computational requirements but highly sophisticated in terms of their mathematical underpinnings. Stanford Professor Don Knuth, author of a landmark series of books entitled *The Art of Computer Programming,* devotes well over 100 pages to the question of how one might code a random number generator whose output appears as random as possible. The details of that mathematical analysis are well beyond the scope of an introductory text. Fortunately, almost no programmers ever have to write the code to generate random numbers. That code has already been developed by someone who has studied the necessary mathematics and waded through all the complex practical and theoretical issues necessary to make it work. Most programmers are interested only in *using* random numbers and have no interest in the details how they are calculated. For those programmers, it is sufficient to think of the random number generator as a black box. You ask it for a random value, and out pops the next one.

The idea of thinking of parts of your programs as black boxes is fundamental to the concept of object-oriented programming. To generate random numbers, you need a random number generator, which can be illustrated by the following black box:

> **RandomGenerator**
> instance

In the Java domain, the black box constitutes an object. Its class is **RandomGenerator**, which is defined in the **acm.util** package. As the label indicates, the black box is a particular instance of the **RandomGenerator** class. When you want to generate a random value, you do so by sending a message to the object, which can be represented pictorially as follows:

Give me the next random number ⟶ **RandomGenerator**
instance
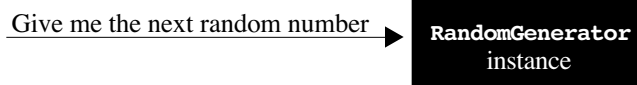
The random generator object then replies with the next random number, computed by some algorithmic process, the details of which are hidden inside the black box. For example, given a request for a random number, the random generator object might give the following reply:

Give me the next random number ⟶ **RandomGenerator**
instance
⟵ 5263172067
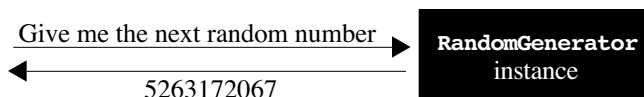
As is always the case in object-oriented programming, the message takes the form of a method call. The reply is simply the value that method returns. As you will learn in the next section, the random generator object responds to several different methods, depending on the type of random value your application needs.

Using the `RandomGenerator` class

The most important public methods in the `RandomGenerator` class are listed in Figure 6-1. For the most part, these methods are easy to understand. The only method that seems to cause confusion is the constructor, which creates a new instance of the `RandomGenerator` class. The important thing to remember is that a `RandomGenerator` is not itself a random value but rather an object that generates random values. Although you will typically want to generate many different random values in the course of a program, you only need to have one generator. Moreover, because you will probably want to use that generator in several methods within your program, it is easiest to declare it as an instance variable, as follows:

```
private RandomGenerator rgen = new RandomGenerator();
```

That line both declares and initializes the variable `rgen` to be a source of random values for the program as a whole.

Once you've declared and initialized the `rgen` variable, you can then use the other methods in `RandomGenerator` to generate the next random value of the appropriate type. Which method you select depends on the application. If you want to select a random integer in a specific range, for example, you use the two-argument version of `nextInt`. Thus, you can simulate rolling a standard six-sided die by writing

```
int die = rgen.nextInt(1, 6);
```

If you want to generate a random real number between –1.0 and +1.0, you would call

```
double r = rgen.nextDouble(-1.0, +1.0);
```

FIGURE 6-1   **Useful methods in the RandomGenerator class**

`RandomGenerator()`
Constructs a new random generator. *Your programs should create only one generator.*
`int nextInt(int n)`
Returns a random integer chosen from the **n** values in the range 0 to **n** - 1, inclusive.
`int nextInt(int low, int high)`
Returns a random integer in the range **low** to **high**, inclusive.
`double nextDouble()`
Returns a random double d in the range $0 \le d < 1$; a range excluding one end is called **half-open.**
`double nextDouble(double low, double high)`
Returns a random double d in the half-open range **low** $\le d <$ **high**.
`boolean nextBoolean()`
Returns a **boolean** that is **true** roughly 50 percent of the time.
`boolean nextBoolean(double p)`
Returns a **boolean** that is **true** with probability **p**, which must be between 0 and 1.
`Color nextColor()`
Returns a random Java color.
`void setSeed(long seed)`
Sets a "seed" to indicate a starting point for the pseudorandom sequence.

If you want to simulate the toss of a coin, you could use **nextBoolean** as follows:

```
String coinFlip = rgen.nextBoolean() ? "Heads" : "Tails";
```

The **Craps.java** program in Figure 6-2 illustrates the use of the **RandomGenerator** class in the context of a complete application. This program simulates the casino game of Craps, which is played as follows. You start by rolling two six-sided dice and looking at the total. The game then breaks down into the following cases based on that first roll:

- Your first roll is a 2, 3, or 12. Rolling these numbers on your first roll is called *craps* and means that you lose.
- Your first roll is a 7 or an 11. When either of these numbers comes up on your first roll, it is called a *natural,* and you win.
- Your first roll is one of the other numbers (4, 5, 6, 8, 9, or 10). In this case, the number you rolled is called your *point*, and you continue to roll the dice until either (a) you roll your point a second time, in which case you win, or (b) you roll a 7, in which case you lose. If you roll any other number—including 2, 3, 11, and 12, which are no longer treated specially—you just keep on rolling until your point or a 7 appears.

The program in Figure 6-2 is a straightforward translation of the English rules into Java code. The sample runs in Figure 6-3 illustrate each of the possible outcomes.

The **Craps.java** program in Figure 6-2 illustrates the standard programming pattern for working with random numbers. The random number generator is declared and initialized by the line

```
private RandomGenerator rgen = new RandomGenerator();
```

at the end of the program. That generator is used to create a succession of random values for the die rolls by the following lines in the method **rollDice**:

```
int d1 = rgen.nextInt(1, 6);
int d2 = rgen.nextInt(1, 6);
```

Each of these declarations uses the random generator **rgen** to obtain a new random integer between 1 and 6, inclusive.

The role of the random number seed

Most of the methods shown in Figure 6-1 for the **RandomGenerator** class return random values of a particular type, which might be integers, double-precision numbers, Boolean values chosen with a particular probability, or even random colors for use in graphical programs. The one method other than the constructor that does not fit into this framework is the **setSeed** method at the very end of the list. To understand what this method does, it is necessary to think a little more concretely about what goes on inside the black box that represents the random number generator.

In Chapter 2, an object was defined as "a conceptually integrated entity that encapsulates both state and behavior." The behavior of the random generator object is defined by the messages to which it responds. Although it is not immediately so clear, the random generator object must also maintain internal state so that it can generate a new random value each time one of its methods is called. The typical pseudorandom number

FIGURE 6-2   Program to play the casino game of Craps

```java
/*
 * File: Craps.java
 * ----------------
 * This program plays the casino game of Craps.  At the beginning of
 * the game, the player rolls a pair of dice and computes the total.
 * If the total is 2, 3, or 12 (called "craps"), the player loses.
 * If the total is 7 or 11 (called a "natural"), the player wins.
 * If the total is any other number, that number becomes the "point."
 * From here, the player keeps rolling the dice until (a) the point
 * comes up again, in which case the player wins or (b) a 7 appears,
 * in which case the player loses.  Note that the numbers 2, 3, 11,
 * and 12, no longer have special significance after the first roll.
 */

import acm.program.*;
import acm.util.*;

public class Craps extends ConsoleProgram {

/* Run the program */
   public void run() {
      int total = rollTwoDice();
      if (total == 7 || total == 11) {
         println("That's a natural.  You win.");
      } else if (total == 2 || total == 3 || total == 12) {
         println("That's craps.  You lose.");
      } else {
         int point = total;
         println("Your point is " + point + ".");
         while (true) {
            total = rollTwoDice();
            if (total == point) {
               println("You made your point.  You win.");
               break;
            } else if (total == 7) {
               println("That's a 7.  You lose.");
               break;
            }
         }
      }
   }

/* Roll two dice and returns their sum. */
   private int rollTwoDice() {
      int d1 = rgen.nextInt(1, 6);
      int d2 = rgen.nextInt(1, 6);
      int total = d1 + d2;
      println("Rolling dice: " + d1 + " + " + d2 + " = " + total);
      return total;
   }

/* Create an instance variable for the random number generator */
   private RandomGenerator rgen = new RandomGenerator();

}
```

FIGURE 6-3   Sample runs illustrating the possible outcomes of a craps game
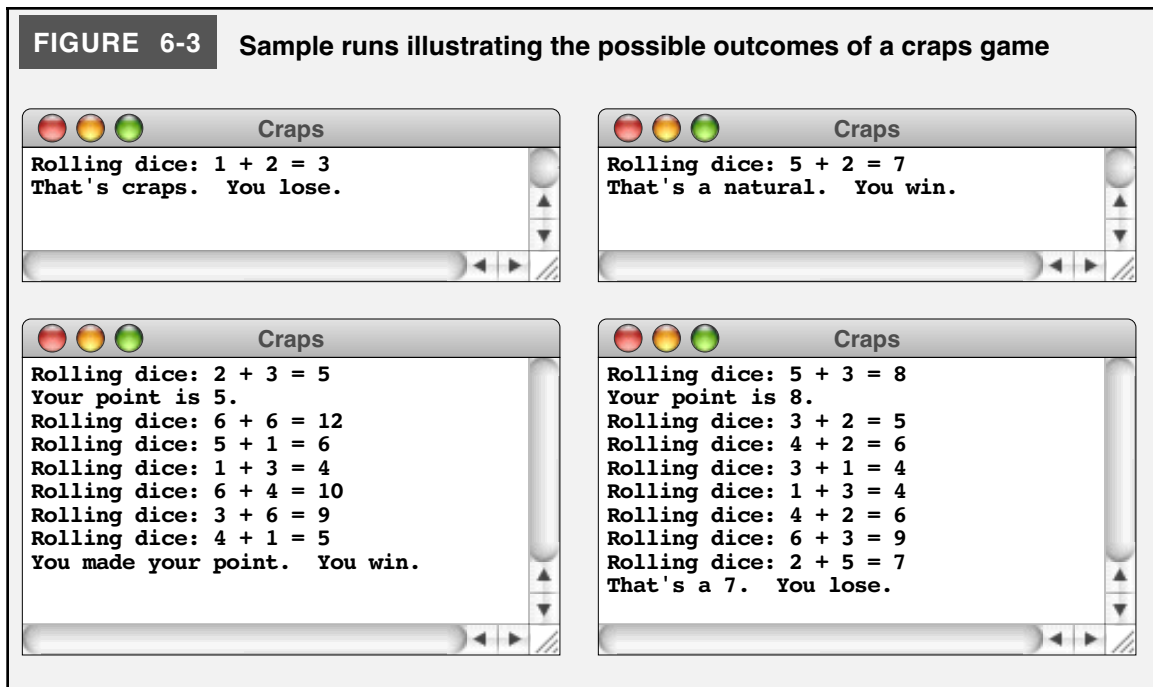
generator is implemented so that it produces a sequence of seemingly random values from a particular starting point. That starting point is called the **seed.** Each time you call one of the methods that generates a random value, the generator takes the current seed, applies a mathematical function to generate the next value in the random sequence, and then stores the result back into the memory reserved for the seed. The method then converts the new seed to the specific type and range of values appropriate to the method call. For example, if you call

```
rgen.nextInt(1, 6);
```

the method converts the current random seed from its internal value as a `long` into an `int` in the range between 1 and 6, inclusive.

Since each call to the random number generator updates the seed, any series of calls to the methods in that generator will produce a sequence of different random values. If you don't know the starting point for that sequence, the pattern of random numbers will be impossible to predict. By default, Java sets the initial seed to the time registered on the internal system clock. As a result, the sequence of values will seem to shift each time you run the program. You can, however, use the `setSeed` method to give the random sequence a specific starting point. If you do that at the beginning of your program, the random values will be the same each time.

On the face of it, the idea that it might be useful for a program to generate the same random sequence each time seems a bit crazy. If you're trying to play a computer game, the last thing you want is to have it behave in exactly the same way each time. Such a game would be entirely predictable and would quickly become boring. Even so, there is a very important reason why one might want to set a specific random seed even in the context of a computer game, which is simply that deterministic programs are generally much easier to debug.

To illustrate this principle, suppose you have just written a program to play an intricate game, such as Monopoly. As is always the case with newly written programs, the odds

are good your program has a few bugs. In a complex program, bugs can be relatively obscure, in the sense that they only occur in rare situations. Suppose you've been playing the game and find that the program starts behaving in a bizarre way, but that you weren't alert enough to pay attention to all the relevant symptoms. You would like to run the program again and watch more carefully this time.

If your program is running in a nondeterministic way, however, a second run of the program will behave differently from the first. Bugs that showed up the first time may not occur on the second pass. In general, it is extremely difficult to reproduce the conditions that cause a program to fail if the program is behaving in a truly random fashion. If, on the other hand, your program can be made to operate deterministically, it will do the same thing each time it is run. This behavior makes it possible for you to recreate a problem, which is just what you need during the debugging phase.

Making it possible to force deterministic behavior during debugging is the primary reason that the **RandomGenerator** class includes the **setSeed** method. For example, if during the debugging phase of your program development, you include a call

```
        rgen.setSeed(1);
```

at the beginning of your program, that program will replay the same set of values every time it is run. If you set the seed to something else, the program will sequence through a different set of random values but will still behave deterministically. Before you ship your Monopoly program to the world, you will of course want to remove this statement, but it can be a godsend when you're getting things to work.

Clients and implementors

The method summary of the **RandomGenerator** class presented in Figure 6-1 is sufficient for many applications but does not represent a complete description of the class. For one thing, the **RandomGenerator** class also includes a few specialized methods that novices are unlikely to use but that may be important for more sophisticated users. For another, the descriptions given in Figure 6-1 are limited to the method header and a one-sentence description. If you needed more detail about one of these methods, you would have to look elsewhere for that information. In the early years of computing before the development of modern object-oriented languages, the ultimate source for such additional information would have been the code that implemented the **RandomGenerator** class. Looking at that code, however, is not an attractive prospect. If you are interested only in using the facilities of a class, you are unlikely to care about how that class is actually implemented. The code brings that implementation to the fore where it can easily get in the way of discovering what you really need to know. There has to be a better way.

Before discussing the approach Java has adopted to separate the general documentation from the details of the underlying implementation, it is useful to introduce two terms that express the difference in perspective between programmers who implement a class and those who use it. Naturally enough, a programmer who implements a class is called an **implementor**. Because the word *user* refers to someone who runs a program rather than someone who writes part of one, computer scientists have adopted the term **client** to refer to a programmer who makes use of the facilities provided by an implementor.

Even though clients and implementors have different perspectives, both must understand certain aspects of the design of a class. As a client, you don't need to know the details of its operation, but you do need to know how to call the methods it contains. As an implementor, you are not concerned with how other programmers may use those methods, but you have to give your clients the information they need to call them. For each method in the class, the client must know the following:

- The name of the method
- The arguments it requires and the types of those arguments
- The type of result it returns
- What the method does in *abstract* terms, presumably expressed in English or some other human language rather than in code.

The first three items here are provided by the method header; the last is typically provided by the comments that are included with the method definition.

The javadoc documentation system

In the Java world, clients are not expected to look at the code to understand how a particular class behaves. What clients typically do instead is to consult the web-based documentation for a class, which is produced automatically by a tool called **javadoc.** The **javadoc** system, which is provided as part of the standard Java Development Kit from Sun, reads through the code for a package and creates documentation for each of the classes it contains.

Figure 6-4 shows the class overview and method summaries for the **RandomGenerator** class. Figure 6-5 then expands the method summaries to show more detail for each of the methods. As you can see from the figures, many of the words in the descriptions are **hyperlinks** that take you to other pages or to specific entries on this page. These hyperlinks are produced automatically by the **javadoc** system and make it much easier to negotiate the documentation.

Layered abstractions

If you look at the start of the **javadoc** description in Figure 6-4, you will discover that the **RandomGenerator** class in the **acm.util** package is a subclass of the **Random** class in **java.util**. As a subclass, **RandomGenerator** inherits the structure and methods of the **Random** class. The method summary in Figure 6-4 makes this inheritance explicit. Although some methods, such as **nextColor**, are defined by the **RandomGenerator** class itself, other methods, such as the simple version of **nextBoolean** that returns **true** 50 percent of the time, are inherited from **Random**. For the most part, it doesn't matter at all which of these two classes actually implements a particular method. As a client of **RandomGenerator**, all that matters to you is that you have ready access to the complete set of methods.

The situation illustrated by the **Random** and **RandomGenerator** classes is common in object-oriented programming. You will often encounter situations in which a class provides some but not all of the functionality you need. Defining a subclass allows you to extend those capabilities because the new class inherits the methods of the old one. The same process can then be repeated so that a class inherits behavior from an entire chain of superclasses. Because each new class builds on the framework provided by its predecessors, such hierarchies are referred to as **layered abstractions.**

6.2 Defining your own classes

At least to a certain extent, the prospect of defining your own classes in Java should not be a particularly daunting prospect. You have, after all, been defining your own classes ever since Chapter 2. Every program is a class definition that extends one of the **Program** subclasses, which means that you have seen enough examples to get a sense of the structure of a class definition.

FIGURE 6-4 Overview and summary for the RandomGenerator javadoc page

acm.util
Class RandomGenerator

java.lang.Object
  |
  +--java.util.Random
         |
         +--acm.util.RandomGenerator

---

public class RandomGenerator extends Random

This class implements a simple random number generator that allows clients to generate pseudorandom integers, doubles, booleans, and colors. To use it, the first step is to declare an instance variable to hold the random generator as follows:

    private RandomGenerator rgen = new RandomGenerator();

When you then need to generate a random value, you call the appropriate method on the rgen variable.

The RandomGenerator class is actually implemented as an extension to the Random class in java.util. The new version has the following advantages over the original:

- The name of the class emphasizes that the object is a random generator rather than a random value.
- The class includes overloaded versions of nextInt and nextDouble to simplify choosing numbers in a specific range.
- The method nextBoolean is overloaded to allow the specification of a probability.
- The class includes a method nextColor that generates a random opaque color.

Constructor Summary

RandomGenerator() Creates a new random generator initialized to an unpredictable starting point.	

Method Summary

boolean	**nextBoolean**(double p) Returns a random **boolean** value with the specified probability.
Color	**nextColor**() Returns a random opaque **Color** whose components are chosen uniformly in the 0-255 range.
double	**nextDouble**(double low, double high) Returns the next random real number in the specified range.
int	**nextInt**(int low, int high) Returns the next random integer in the specified range.

Inherited Method Summary

boolean	**nextBoolean**() Returns a random boolean that is true 50 percent of the time.
double	**nextDouble**() Returns a random double d in the range $0 \le d < 1$.
int	**nextInt**(int n) Returns a random integer k in the range $0 \le k < n$.
void	**setSeed**(long seed) Sets a new starting point for the random generator sequence.

FIGURE 6-5   Detail section for the RandomGenerator javadoc page

Constructor Detail

public RandomGenerator()

Creates a new random generator initialized to an unpredictable starting point. In almost all programs, you want to call this method <u>once</u> to produce a single generator object, which you then use each time you need to generate a random value. If you create several random generators in succession, they will typically generate the same sequence of values.

During debugging, it is often useful to set the internal seed for the random generator explicitly so that it always returns the same sequence. To do so, you need to invoke the **setSeed** method.

Usage: `RandomGenerator rgen = new RandomGenerator();`

Method Detail

public boolean nextBoolean(double p)

Returns a random **boolean** value with the specified probability. You can use this method to simulate an event that occurs with a particular probability. For example, you could simulate the result of tossing a coin like this:

```
String coinFlip = rgen.nextBoolean(0.5) ? "HEADS" : "TAILS";
```

Usage: `if (rgen.nextBoolean(p)) . . .`
Parameter: **p** A value between 0 (impossible) and 1 (certain) indicating the probability
Returns: The value **true** with probability **p**

---

public Color nextColor()

Returns a random opaque **Color** whose components are chosen uniformly in the 0-255 range.

Usage: `Color color = rgen.newColor()`
Returns: A random opaque **Color**

---

public double nextDouble(double low, double high)

Returns the next random real number in the specified range. The resulting value is always at least **low** but always strictly less than **high**. You can use this method to generate continuous random values. For example, you can set the variables **x** and **y** to specify a random point inside the unit square as follows:

```
double x = rgen.nextDouble(0.0, 1.0);
double y = rgen.nextDouble(0.0, 1.0);
```

Usage: `double d = rgen.nextDouble(low, high)`
Parameters: **low** The low end of the range
               high The high end of the range
Returns: A random **double** value d in the range $low \le d < high$

---

public int nextInt(int low, int high)

Returns the next random integer in the specified range. For example, you can generate the roll of a six-sided die by calling

```
rgen.nextInt(1, 6);
```

or a random decimal digit by calling

```
rgen.nextInt(0, 9);
```

Usage: `int k = rgen.nextInt(low, high)`
Parameters: **low** The low end of the range
               high The high end of the range
Returns: The next random **int** between **low** and **high**, inclusive

The basic structure of a class definition is shown in the syntax box on the right. In Java, every class that is available for use by clients—as opposed to being defined only within the context of a package—must be marked with the keyword **public**. Each public class definition, moreover, should be in a separate source file called *name*.**java** where *name* is the name of the new class. The **extends** specification indicates the superclass from which this class extends. If the **extends** specification is missing, the new class extends the **Object** class in **java.lang**, which is the root of Java object hierarchy. The body of the class is a sequence of definitions of various kinds. In the classes you have seen so far, these definitions have been limited to methods and

> Syntax for a public class definition:
>
> > **public class** *name* **extends** *super* **{**
> >   *class body*
> > **}**
>
> where:
> > *name* is the name of the new class
> > *super* is the name of the superclass
> > *class body* is a sequence of definitions that
> >   will typically include constructors,
> >   methods, constants, and instance
> >   variables

named constants. More generally, classes will also contain **constructors,** which specify how to create new instances of the class, and **instance variables,** which maintain the state of the objects. Constructors are defined the same way as methods, with two exceptions: (1) the name of the constructor is always the same as that of the class, and (2) a constructor does not specify a result type. Instance variable declarations look much the same as any other variable declarations except that they appear outside of any method definition at the top level of a class.

Defining a Stoplight class

The role of constructors and instance variables can be illustrated most easily by example. Suppose, for example, that you want to represent a simple stoplight as a Java class. Stoplights have internal state, because the light can be in any of three states: green, yellow, and red. Stoplights also have behavior. A stoplight that is green will at some point turn yellow, then red, and eventually back to green, as illustrated by the following diagram:
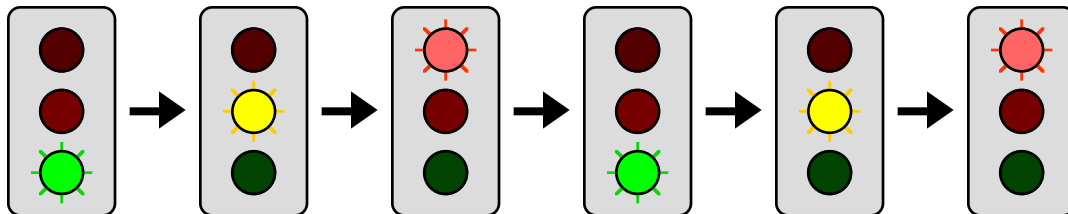
Stoplights, however, never change in the opposite order, so that it would be impossible, for example, to have a stoplight turn from yellow to green.

A simple definition for a **Stoplight** class appears in Figure 6-6. The elements that make up the class are:

• *The named constants* **GREEN***,* **YELLOW***, and* **RED***.* In order to determine what color is showing on the stoplight, the class must specify the possible states for the stoplight. The simplest strategy for doing so is to define named constants for the green, yellow, and red states. In the code in Figure 6-6, the values of these constants are set to the corresponding constant values in the **Color** class introduced briefly in Chapter 5. For now, the only advantage to using these values—as opposed, for example, to the

FIGURE 6-6   Implementation of a Stoplight class

```java
import java.awt.*;

/**
 * This class represents a simple implementation of a stoplight.
 * The client can determine the current state of the stoplight by
 * calling stoplight.getState() and change it to the next color
 * in the sequence (GREEN -> YELLOW -> RED -> GREEN) by calling
 * stoplight.advance().
 */

public class Stoplight {

/** Constant indicating the color GREEN */
   public static final Color GREEN = Color.GREEN;
/** Constant indicating the color YELLOW */
   public static final Color YELLOW = Color.YELLOW;
/** Constant indicating the color RED */
   public static final Color RED = Color.RED;

/**
 * Creates a new Stoplight object, which is initially GREEN.
 */
   public Stoplight() {
      state = GREEN;
   }

/**
 * Returns the current state of the stoplight.
 * @return The state of the stoplight (GREEN, YELLOW, or RED)
 */
   public Color getState() {
      return state;
   }

/**
 * Advances the stoplight to the next state.
 */
   public void advance() {
      if (state == RED) {
         state = GREEN;
      } else if (state == YELLOW) {
         state = RED;
      } else if (state == GREEN) {
         state = YELLOW;
      }
   }

/* Private instance variable */

   private Color state;

}
```

characters **'G'**, **'Y'**, and **'R'**—lies in the idea that using existing values reduces the number of new definitions and therefore preserves a certain economy. If, however, you wanted to use the **Stoplight** class in a graphical application, using the standard Java colors would offer the additional advantage of making it easier to draw the stoplight elements in the correct color.

- *A constructor to create a new **Stoplight** instance*. The first entry in the file after the named constants is the **Stoplight** constructor, which creates a new instance of the class. Constructors need not be specified unless a class needs to perform some initialization on new objects. In this case, the constructor sets the state of the stoplight to **GREEN** as specified in the constructor description.

- *A method to determine the state of the stoplight*. The **getState** method allows the client to determine the current state of the stoplight, which is always one of the constants **GREEN**, **YELLOW**, or **RED**. Internally, the class stores the current state in the instance variable **state**, but that variable is not directly accessible to the client for reasons that are outlined later in this section.

- *A method to advance the stoplight to the next state*. The **advance** method changes the state of the stoplight from **GREEN** to **YELLOW**, **YELLOW** to **RED**, or **RED** to **GREEN**, as appropriate.

- *A private instance variable to store the current state*. The instance variable **state** keeps track of the current color of the stoplight. Because the variable is declared to be **private**, only the methods in this class have direct access to it. Clients can determine the state of a stoplight indirectly by calling **getState** but cannot change the state other than by calling **advance**. Declaring this variable to be **private** therefore ensures that the stoplight object can never change directly from **GREEN** to **RED** without going through **YELLOW** in between.

If you look closely at the code in Figure 6-6, you may notice one other difference from the examples in the earlier chapters. Instead of beginning with **/*** as every comment has from the beginning of the book, the comments in the **Stoplight** class definition begin with **/****. Because Java ignores everything between **/*** and ***/**, the extra asterisk is simply part of the text of the comment and has no significance for the compiler. That extra asterisk, however, indicates that this comment contains information for use by the **javadoc** documentation tool described earlier in this chapter. For now, these comments include only a brief description of the declaration that follows. As you will discover later in this chapter, it is also possible to specify such additional information as details about the parameters accepted by a particular method or the type of result such a method returns.

The message-passing model

As discussed in Chapter 5, object-oriented programming uses the metaphor of sending messages from one object to another. In practice, message passing is implemented using method calls, but it is often useful to think in terms of messages when designing an object-oriented application. In the stoplight example, the basic model is that the client will create one or more instances of the **Stoplight** class and then manipulate those instances by sending messages.

The simplest example consists of creating a single stoplight, which you can do by invoking **new Stoplight**, as follows:

```
Stoplight myStoplight = new Stoplight();
```

This declaration creates a new **Stoplight** instance and assigns it to **myStoplight**, which you could diagram conceptually as follows:
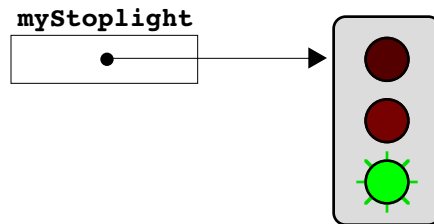
As you will discover in Chapter 7, the **myStoplight** variable is not large enough to hold all the information associated with the **Stoplight** object. That variable instead keeps track of an internal memory address that allows it to find the data associated with the actual **Stoplight** instance. At the more abstract level reflected in this diagram, that internal memory address is called a **reference.**

To advance the stoplight to the next phase in the cycle, all you need to do is send the "advance" message to **myStoplight** like this:

    myStoplight.advance();

This operation corresponds to the following diagram, which shows the state of the stoplight before the message takes effect:
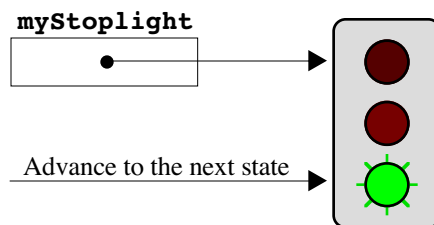
To verify that the light indeed changed, you could then call

    Color currentState = myStoplight.getState();

which you could diagram as follows:
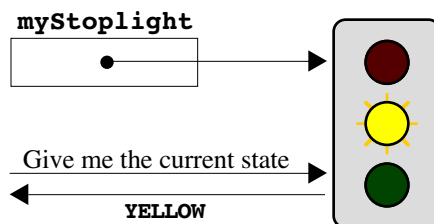
Maintaining conceptual separation between the client and the implementation

In the discussion of the various components of the **Stoplight** class, the reason offered for declaring **state** as **private** was that doing so ensured that the stoplight would always advance in the correct order, from **GREEN** to **YELLOW** to **RED** and back to **GREEN** again. Although that argument is powerful in its own right, there are many other reasons to make instance variables private to a class. One of the strongest justifications for using

private instance variables is that doing so gives the implementor greater freedom to change the implementation of a class. In the implementation shown in Figure 6-6, the state of each **Stoplight** object is stored in the variable **state**, which keeps track of the actual color. The implementation of the **Stoplight** class becomes slightly shorter if you change the underlying representation so that the state is maintained as an integer instead, as shown in Figure 6-7. The **advance** method then becomes simpler because it can use addition to move to the next state. The **getState** method, however, becomes more complex because it has to translate the integer used internally into the **Color** object that **getState** is defined to return.

The important point illustrated by the alternative implementation in Figure 6-7 is that the client has no way to tell the difference between the two implementations. The behavior of the two classes is exactly the same. If the implementor decided to replace one implementation with the other, clients of the **Stoplight** class shouldn't even notice. The same would not be true if the client had access to the details of the internal representation. Changing the underlying representation would almost certainly upset some group of clients who had decided to rely on specific characteristics of the original one.

In fact, the advantages of making instance variables private are so compelling that all instance variables in this book—and indeed all instance variables in the ACM libraries—are marked using the keyword **private**. Although public instance variables have occasional uses (as do the protected and package private designations that are beyond the scope of this book), you will develop better programming habits if you adhere to the convention of using only private instance variables.

6.3 Defining a class to represent rational numbers

Although the stoplight example from the preceding section explores the basic mechanics of classes, developing a solid understanding of the topic requires you to consider more sophisticated example. To get a sense of the importance of the specific example, imagine yourself back in elementary school facing the problem of adding the following fractions:

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{6} = 1$$

Basic arithmetic—or even a little intuition—makes it clear that the mathematically exact answer is 1, but that answer might be difficult to obtain using double-precision arithmetic on a computer. Consider, for example, the following **run** method:

```
public void run() {
    double sum = 1.0 / 2.0 + 1.0 / 3.0 + 1.0 / 6.0;
    println("1.0 / 2.0 + 1.0 / 3.0 + 1.0 / 6.0 = " + sum);
}
```

If you execute a Java program containing this **run** method, you get the following output:

```
RoundoffExample
1.0 / 2.0 + 1.0 / 3.0 + 1.0 / 6.0 = 0.9999999999999999
```
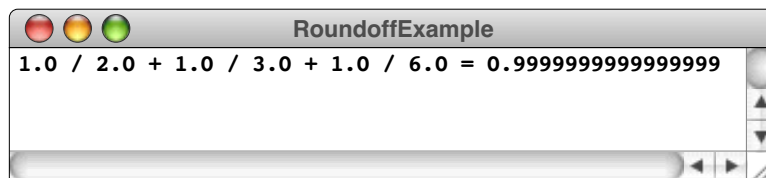
Although the result of 0.9999999999999999 is certainly close enough for practical purposes, your elementary school teacher would surely suspect the use of impermissible calculating devices if you suggested this answer in class.

FIGURE 6-7   **Revised implementation of Stoplight using integers to store the state**

```java
import java.awt.*;

/**
 * This class represents a simple implementation of a stoplight.
 * The client can determine the current state of the stoplight by
 * calling stoplight.getState() and change it to the next color
 * in the sequence (GREEN -> YELLOW -> RED -> GREEN) by calling
 * stoplight.advance().
 */

public class Stoplight {

/** Constant indicating the color GREEN */
   public static final Color GREEN = Color.GREEN;
/** Constant indicating the color YELLOW */
   public static final Color YELLOW = Color.YELLOW;
/** Constant indicating the color RED */
   public static final Color RED = Color.RED;

/**
 * Creates a new Stoplight object, which is initially GREEN.
 */
   public Stoplight() {
      state = 0;
   }

/**
 * Returns the current state of the stoplight.
 * @return The state of the stoplight (GREEN, YELLOW, or RED)
 */
   public Color getState() {
      switch (state) {
        case 0:  return GREEN;
        case 1:  return YELLOW;
        case 2:  return RED;
        default: return null;  /* Can't occur but required by Java */
      }
   }

/**
 * Advances the stoplight to the next state.
 */
   public void advance() {
      state = (state + 1) % 3;
   }

/* Private instance variable */

   private int state;

}
```

The problem is simply that the memory cells used to store numbers inside a computer have a limited storage capacity, which in turn restricts the precision they can offer. Within the limits of double-precision arithmetic, the sum of one-half plus one-third plus one-sixth is 0.9999999999999999, but such an answer is not mathematically satisfying. To get an exact answer you need to move out of the realm of double-precision numbers and into the domain of **rational numbers,** which are those values that can be represented as the quotient of two integers. Rational numbers are well understood as a mathematical concept and have their own arithmetic rules, which are summarized in Figure 6-8. Unfortunately, rational numbers are not understood by Java. If you wanted to use rational arithmetic in Java, you would have to define a new class to represent them.

If you were able to define a `Rational` class, you could use it in Java programs to perform mathematically precise calculations. For example, the following `run` method would perform the fraction calculation used in the earlier examples:

```
public void run() {
   Rational a = new Rational(1, 2);
   Rational b = new Rational(1, 3);
   Rational c = new Rational(1, 6);
   Rational sum = a.add(b).add(c);
   println(a + " + " + b + " + " + c + " = " + sum);
}
```

The code declares three variables—`a`, `b`, and `c`—and initializes them to the rational values corresponding to the fractions in the calculation. The next line then declares the variable `sum` and initializes it to the sum of `a`, `b`, and `c`. The only possible source of confusion in this line is that addition for the `Rational` class must be expressed using the `add` method because the `+` operator is defined only for the primitive types. The last line then prints out the three values and their sum. The output of this program—assuming that the `Rational` class operates as it should—would look like this:

```
TestRational
1/2 + 1/3 + 1/6 = 1
```
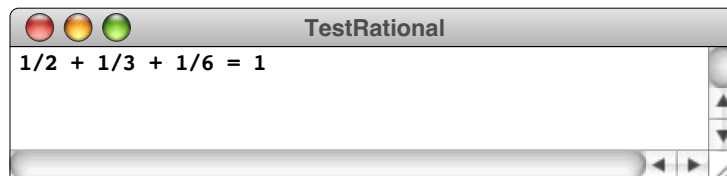
Figure 6-9 shows a scaled-down implementation of a `Rational` class that implements the four basic arithmetic operations: addition, subtraction, multiplication, and division.

FIGURE 6-8   **Rules for rational arithmetic**

Addition

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

Multiplication

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

Subtraction

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

Division

$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$$

FIGURE 6-9   Definition of a class to represent rational numbers

```java
import acm.util.*;

/**
 * The Rational class is used to represent rational numbers, which
 * are defined to be the quotient of two integers.
 */
public class Rational {

/**
 * Creates a new Rational initialized to zero.
 */
   public Rational() {
      this(0);
   }

/**
 * Creates a new Rational from the integer argument.
 * @param n The initial value
 */
   public Rational(int n) {
      this(n, 1);
   }

/**
 * Creates a new Rational with the value x / y.
 * @param x The numerator of the rational number
 * @param y The denominator of the rational number
 */
   public Rational(int x, int y) {
      if (y == 0) throw new ErrorException("Division by 0");
      int g = gcd(Math.abs(x), Math.abs(y));
      num = x / g;
      den = Math.abs(y) / g;
      if (y < 0) num = -num;
   }

/**
 * Adds the rational number r to this one and returns the sum.
 * @param r The rational number to be added
 * @return The sum of the current number and r
 */
   public Rational add(Rational r) {
      return new Rational(this.num * r.den + r.num * this.den,
                          this.den * r.den);
   }

/**
 * Subtracts the rational number r from this one.
 * @param r The rational number to be subtracted
 * @return The result of subtracting r from the current number
 */
   public Rational subtract(Rational r) {
      return new Rational(this.num * r.den - r.num * this.den,
                          this.den * r.den);
   }
```

FIGURE 6-9   **Definition of a class to represent rational numbers** (continued)

```java
/**
 * Multiplies this number by the rational number r.
 * @param r The rational number used as a multiplier
 * @return The result of multiplying the current number by r
 */
   public Rational multiply(Rational r) {
      return new Rational(this.num * r.num, this.den * r.den);
   }

/**
 * Divides this number by the rational number r.
 * @param r The rational number used as a divisor
 * @return The result of dividing the current number by r
 */
   public Rational divide(Rational r) {
      return new Rational(this.num * r.den, this.den * r.num);
   }

/**
 * Creates a string representation of this rational number.
 * @return The string representation of this rational number
 */
   public String toString() {
      if (den == 1) {
         return "" + num;
      } else {
         return num + "/" + den;
      }
   }

/**
 * Calculates the greatest common divisor using Euclid's algorithm.
 * @param x First integer
 * @param y Second integer
 * @return The greatest common divisor of x and y
 */
   private int gcd(int x, int y) {
      int r = x % y;
      while (r != 0) {
         x = y;
         y = r;
         r = x % y;
      }
      return y;
   }

/* Private instance variables */

   private int num;    /* The numerator of this Rational   */
   private int den;    /* The denominator of this Rational */

}
```

Much of the class definition is straightforward, but the code does illustrate several new features, as follows:

- *Multiple constructors*. The `Rational` class in Figure 6-9 defines three different constructors. Calling `new Rational()` creates a new rational number whose value is 0, which is represented internally as the fraction $0/1$. Calling `new Rational(`*n*`)` creates a new rational number equal to the integer *n*, which is simply the fraction $n/1$. Finally, calling `new Rational(`*x,* *y*`)` creates a new rational number equal to the fraction x/y. Java can tell which constructor to call by looking at the arguments provided by the client. Constructors and methods that have multiple argument forms under the same name are said to be **overloaded.**

- *The keyword* `this`. Within any class, Java defines the keyword `this` to mean the current object. This `Rational` class uses the keyword `this` in two contexts. The first example is in the first two versions of the constructor, which use `this` to invoke a more general version of the constructor. The second example comes up in the methods that perform arithmetic, where `this` is used to make it emphasize that the `num` and `den` instance variables are the ones in the current rational number. Consider, for example, the definition of the `multiply` method, which looks like this:

```
public Rational multiply(Rational r) {
    return new Rational(this.num * r.num, this.den * r.den);
}
```

This method generates a new `Rational` value whose numerator is the product of this object's numerator and `r`'s numerator and whose denominator is likewise the product of the denominators of these same two objects.

- *The* `toString` *method*. The `Rational` class includes a method called `toString` that returns a string showing how this object should be displayed. The `toString` method is integrated into the Java language and libraries and provides a standard approach for converting values to their string representation. By overriding the definition in this class with a `toString` method appropriate to rational numbers, statements like

```
println(a + " + " + b + " + " + c + " = " + sum);
```

in the test program can display the rational values in a reasonable way.

- *Public and private methods together in one class*. The public methods in the `Rational` class are those that clients of the class will need to call. These include the constructors, the methods that implement the arithmetic operators, and the `toString` method described in the preceding point. This class, however, also includes a private method used in its own implementation that does not need to be exported to clients. That method is the `gcd` function from Chapter 5 that implements Euclid's algorithm and is used here to reduce the fraction to lowest terms. As a general rule, methods should be marked as private whenever you can.

- *Error exceptions*. Because it is illegal for the denominator of a rational number to be zero, the constructor checks for this case and signals an error condition by invoking

```
throw new ErrorException("Division by zero");
```

The effect of this statement is to interrupt the execution of the program and report that an exceptional condition has occurred. This book uses the `ErrorException` class defined in the `acm.util` package to report error conditions. A more complete discussion of exceptions occurs toward the end of the book.

- *Parameter and result tags*. The comments before the constructors and methods in the `Rational` class include `@param` and `@result` tags that provide information to the `javadoc` system about the parameters and result of the method, respectively. It is good to get in the habit of including these tags as you write your code, both because doing so makes it easier to generate the automatic documentation and because these markers provide useful information to readers of your code.

The `Rational` class offers even more evidence in support of keeping instance variables private. While it might be useful to some clients to be able to obtain the numerator and denominator of a rational number `r` by selecting the `r.num` and `r.den` variables within the object, making these variables public compromises the integrity of the class. The `Rational` class makes sure that the following two properties are maintained:

1. The denominator of the rational number is always positive. If the denominator is zero, the constructor reports a division-by-zero error. If the denominator is negative, the constructor transfers the sign to the numerator by negating that value.

2. The fraction represented by the numerator and denominator is always reduced to lowest terms by dividing both parts of the fraction by their greatest common divisor. This design decision ensures that rational numbers will always be displayed in their simplest form.

Properties that are guaranteed to be true throughout a body of code are called **invariants.** Maintaining these invariants is only possible if the class can guard against clients that accidentally or maliciously undermine them. If the instance variables were public, any client could violate the first invariant of rational number `r` by setting `r.den` to 0 or violate the second invariant in any of a number of ways.

The implementation of `Rational` class in Figure 6-9 actually goes even further to preserve the invariants of the class. Declaring the instance variables as private ensures that clients cannot assign new values to them in an assignment statement. If you look carefully at the methods in the `Rational` class, you will discover that none of them ever assign values to these variables either. Once instance variables have been assigned values by the constructor, those values can never be changed during the lifetime of the object. Classes whose internal state cannot be changed by the client, even through method calls, are said to be **immutable.** Immutable types have several extremely useful properties and will come up quite often in subsequent chapters.

Summary

The purpose of this chapter is to introduce the ideas of classes and objects at a holistic, conceptual level. In Java, a *class* represents a template for a set of values that have a common structure and behavior; an *object* is a particular example of that class. Thus, given a single class definition, there can be arbitrarily many *instances* of that class.

In addition to offering a more complete definition of objects and classes, the important points introduced in this chapter include:

- For the most part, computers operate *deterministically* in the sense that the output produced by a program is determined by its inputs, with the same inputs always generating the same results. For applications such as computer games and simulations, however, it is important to make programs operate *nondeterministically* by introducing randomness into their behavior. Because computing hardware typically does not

support true randomness, most programming environments simulate random behavior by providing some method to produce *pseudorandom numbers*.

- The `acm.util` package includes a `RandomGenerator` class that offers several methods for simulating random events. The methods exported by the `RandomGenerator` class are listed in Figure 6-1.

- The `RandomGenerator` class defined in `acm.util` is actually a subclass of the `Random` class in `java.util` and therefore provides the full set of capabilities offered by the standard library class along with a few extensions. Because the new extensions of `RandomGenerator` are superimposed on top of the underlying capabilities of `Random`, these classes represent what is called a *layered abstraction*.

- Defining new classes in Java makes it possible for one programmer to provide useful services to other programmers who need to make use of those capabilities. The programmer who writes the code for the class is called the *implementor*. Programmers who use that class as a resource in programs of their own are called *clients*.

- Clients and implementors have different perspectives about a class. The client needs to know how to use the capabilities provided by a class but is generally unconcerned about the details of how those features actually work. The implementor, by contrast, must understand those details. Moreover, to ensure that the client can work at an appropriately high level of abstraction, the implementor of a class should try to hide as much of the underlying detail as possible.

- The usual strategy for giving clients the information they need about a Java class is to use the **javadoc** documentation system. The **javadoc** applications reads through the code for a class and automatically generates web pages that document the methods in that class.

- You can define your own classes by enclosing the definitions for that class in a block identified by the keyword `class`. Most class definitions specify the name of a specific superclass that the new class extends; in the absence of a specific superclass, the new class extends the `Object` class in `java.lang`, which is the root of Java hierarchy.

- Class definitions typically contain *constructors, methods, named constants,* and *instance variables*. Constructors specify how to create new objects of that class, methods define the behavior of that class, named constants provide a mechanism (already discussed in Chapter 3) to assign more readable names to constant values, and instance variables allow each object to maintain its internal state. Collectively, these definitions are called *entries*.

- Each entry in a class can be marked using the keyword `public` or the keyword `private` to indicate who has access to that entry. Constructors, methods, and constants should be public only if a client would need to use them. In this book, all instance variables are marked as private to ensure that the details of the internal state remain hidden from clients.

- A class can contain several constructors and methods with the same name as long as they have different argument structures that allows the compiler to determine which version is required. If more than one method definition exists for a particular name, that method name is said to be *overloaded*.

- Java defines the keyword `this` to mean the current object. You can use this keyword in expressions to refer to instance variables within the current object if there might otherwise be ambiguity. You can also use the keyword `this` in a constructor to invoke one of the other constructors for this object.

- Classes that do not allow clients to change any properties of an object once it has been created are called *immutable*. The `Rational` class in section 6-3 is an example of an immutable class, as are several of the classes introduced in later chapters.

Review questions

1. Why might it be useful for a program to behave nondeterministically?

2. What is meant by the term *pseudorandom number?*

3. This chapter encourages you to define an instance variable called `rgen` to hold the `RandomGenerator` object that generates each random value. What is the exact syntax of this line?

4. How would you use the methods in the `RandomGenerator` class to generate each of the following values:

    a) A random digit between 0 and 9
    b) A `double` between -1.0 and +1.0
    c) A `boolean` value that is `true` one-third of the time
    d) A random color value suitable for setting the color of a graphical object

5. Assuming that `d1` and `d2` have been declared as variables of type `int`, can you use the multiple assignment statement

        d1 = d2 = rgen.nextInt(1, 6);

    to simulate the process of rolling two dice?

6. In what circumstances is it useful to call the `setSeed` method?

7. Describe the differences in perspective between clients and implementors.

8. On what aspects of the design of a class do clients and implementors need to agree?

9. What is the role of the **javadoc** documentation system?

10. What is a *layered abstraction?*

11. Most class definitions include the keyword `extends` to specify the superclass. What happens if you omit this keyword?

12. What are the most common types of entries in a class definition?

13. How does the implementor of the `Stoplight` class described in this chapter ensure that the client cannot change the light from green to red without going through yellow on the way?

14. True or false: All instance variables in this book and in the ACM library packages are defined using the keyword `private`.

15. What is meant by the term *overloading?*

16. In Java, what does the keyword `this` signify? In what two contexts is it used in this chapter?

17. What property makes a class *immutable?*

Programming exercises

1. Write a program that displays the name of a card randomly chosen from a complete deck of 52 playing cards. Each card consists of a **rank** (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King) and a **suit** (Clubs, Diamonds, Hearts, Spades). Your program should display the complete name of the card, as shown in the following sample run:

```
This program displays a randomly chosen card.
Queen of Spades
```
RandomCard
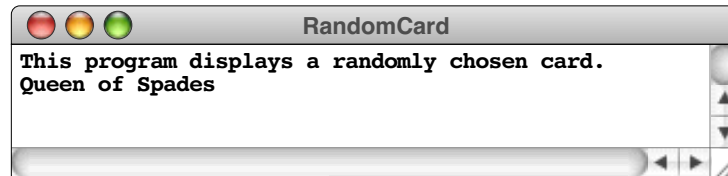
2.   Heads. . . .
                Heads. . . .
                Heads. . . .
                A weaker man might be moved to re-examine his faith, if in nothing
                else at least in the law of probability.
                                    —Tom Stoppard, *Rosencrantz and Guildenstern are Dead,* 1967

    Write a program that simulates flipping a coin repeatedly and continues until three *consecutive* heads are tossed. At that point, your program should display the total number of coin flips that were made. The following is one possible sample run of the program:

```
Heads
Tails
Heads
Heads
Tails
Tails
Tails
Tails
Heads
Tails
Tails
Heads
Heads
Heads
It took 14 flips to get 3 consecutive heads.
```
ConsecutiveHeads
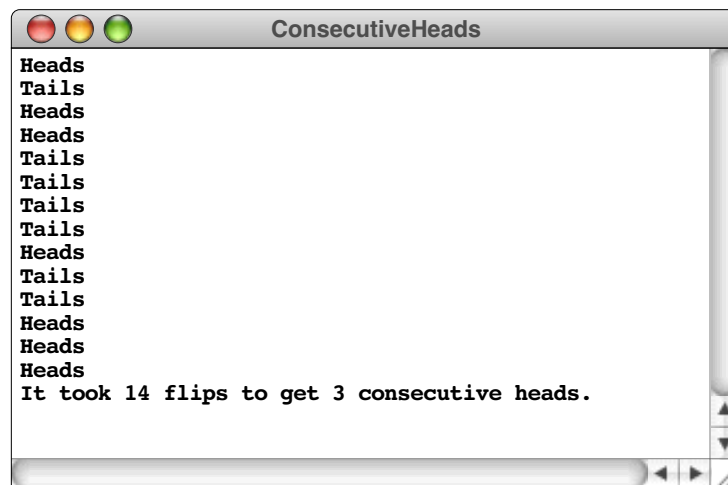
3. Although it is often easiest to think of random numbers in the context of games of chance, they have other, more practical uses in computer science and mathematics. For example, you can use random numbers to generate a rough approximation of the constant π by writing a simple program that simulates a dart board. Imagine that you have a dart board hanging on your wall. It consists of a circle painted on a square backdrop, as in the following diagram:
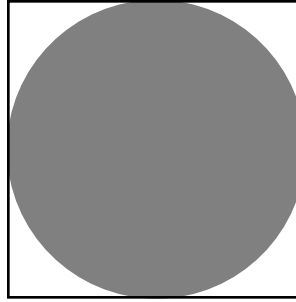
If you throw darts at this board in a random fashion, some will fall inside the circle, but some will fall outside. If the tosses are truly random, the ratio of the number of darts that land inside the circle to the total number of darts hitting the square should be roughly equal to the ratio between the two areas. The ratio of the areas is independent of the actual size of the dart board, as illustrated by the following formula:

$$\frac{\text{darts falling inside the circle}}{\text{darts falling inside the square}} \cong \frac{\text{area of the circle}}{\text{area of the square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

To simulate this process in a program, imagine that the dart board is drawn in the standard Cartesian coordinate plane you learned about in high school. The process of throwing a dart randomly at the square can be modeled by generating two random numbers, x and y, each of which lies between -1 and 1. This (x, y) point always lies somewhere inside the square. The point (x, y) lies inside the circle if

$$\sqrt{x^2 + y^2} < 1$$

This condition, however, can be simplified considerably by squaring each side of the inequality, which gives the following more efficient test:

$$x^2 + y^2 < 1$$

If you perform this simulation many times and compute the fraction of darts that fall within the circle, the result will be somewhere in the neighborhood of $\pi/4$.

Write a program that simulates throwing 10,000 darts and then uses the simulation technique described in this exercise to generate and display an approximate value of π. Don't worry if your answer is correct only in the first few digits. The strategy used in this problem is not particularly accurate, even though it occasionally proves useful as a technique for making rough approximations. In mathematics, this technique is called *Monte Carlo integration*, after the gambling center that is the capital city of Monaco.
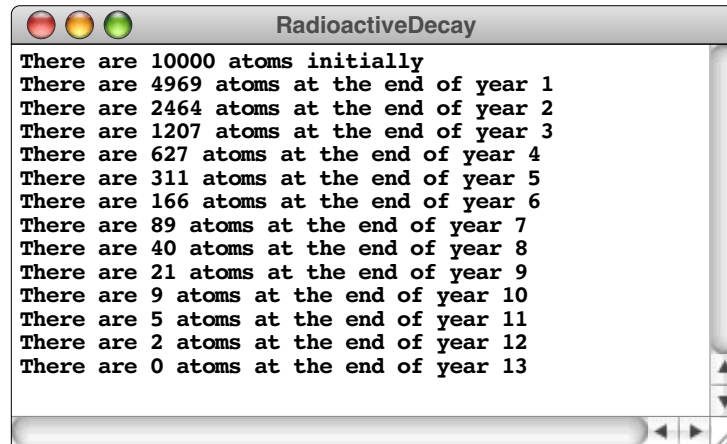
4.
> I shall never believe that God plays dice with the world.
>
> —Albert Einstein, 1947

Despite Einstein's metaphysical objections, the current models of physics, and particularly of quantum theory, are based on the idea that nature does indeed involve random processes. A radioactive atom, for example, does not decay for any specific reason that we mortals understand. Instead, that atom has a random probability of decaying within a period of time. Sometimes it does, sometimes it doesn't, and there is no way to know for sure.

Because physicists consider radioactive decay a random process, it is not surprising that random numbers can be used to simulate that process. Suppose you start with a collection of atoms, each of which has a certain probability of decaying in any unit of time. You can then approximate the decay process by taking each atom in turn and deciding randomly whether it decays, considering the probability.

Write a program that simulates the decay of a sample that contains 10,000 atoms of radioactive material, where each atom has a 50 percent chance of decaying in a year. The output of your program should show the number of atoms remaining at the end of each year, which might look something like this:
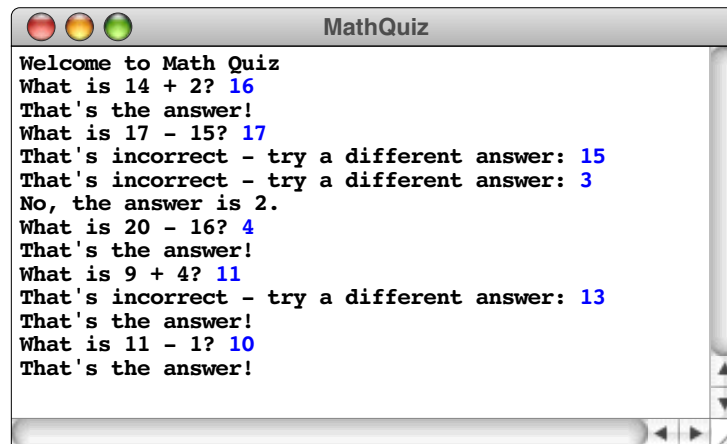
```
⬤ ⬤ ⬤                RadioactiveDecay
There are 10000 atoms initially
There are 4969 atoms at the end of year 1
There are 2464 atoms at the end of year 2
There are 1207 atoms at the end of year 3
There are 627 atoms at the end of year 4
There are 311 atoms at the end of year 5
There are 166 atoms at the end of year 6
There are 89 atoms at the end of year 7
There are 40 atoms at the end of year 8
There are 21 atoms at the end of year 9
There are 9 atoms at the end of year 10
There are 5 atoms at the end of year 11
There are 2 atoms at the end of year 12
There are 0 atoms at the end of year 13
```

As the numbers indicate, roughly half the atoms in the sample decay each year. In physics, the conventional way to express this observation is to say that the sample has a **half-life** of one year.

5. As computers become more common in schools, it is important to find ways to use the machines to aid in the teaching process. This need has led to the development of an educational software industry that has produced many programs that help teach concepts to children.

As an example of an educational application, write a program that poses a series of simple arithmetic problems for a student to answer, as illustrated by the following sample run:

```
⬤ ⬤ ⬤                    MathQuiz
Welcome to Math Quiz
What is 14 + 2? 16
That's the answer!
What is 17 – 15? 17
That's incorrect – try a different answer: 15
That's incorrect – try a different answer: 3
No, the answer is 2.
What is 20 – 16? 4
That's the answer!
What is 9 + 4? 11
That's incorrect – try a different answer: 13
That's the answer!
What is 11 – 1? 10
That's the answer!
```

Your program should meet these requirements:

- It should ask a series of five questions. As with any such limit, the number of questions should be coded as a named constant so that it can easily be changed.
- Each question should consist of a single addition or subtraction problem involving just two numbers, such as "What is 2 + 3?" or "What is 11 – 7?". The type of problem—addition or subtraction—should be chosen randomly for each question.
- To make sure the problems are appropriate for students in the first or second grade, none of the numbers involved, including the answer, should be less than 0 or greater than 20. This restriction means that your program should never ask questions like "What is 11 + 13?" or "What is 4 – 7?" because the answers are outside the legal range. Within these constraints, your program should choose the numbers randomly.
- The program should give the student three chances to answer each question. If the student gives the correct answer, your program should indicate that fact in some properly congratulatory way and go on to the next question. If the student does not get the answer in three tries, the program should give the answer and go on to another problem.

6. Even though the program in exercise 5 was designed to offer encouragement when the student responds correctly, the monotonous repetition of a sentence like "That's the answer!" has the opposite effect after a while. To add variety to the interaction, modify your solution to exercise 5 so that it randomly chooses among four or five different messages when the student gets the right answer, as illustrated in this sample run:
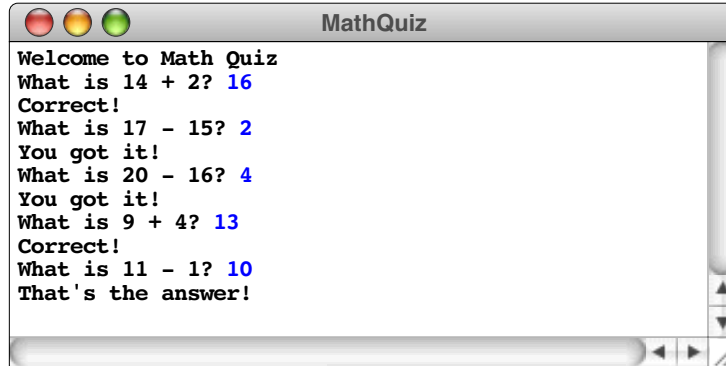
7. Implement a new class called **Card** that includes the following entries:

- Named constants for the four suits (**CLUBS**, **DIAMONDS**, **HEARTS**, **SPADES**) and the four ranks that are traditionally represented in words (**ACE**, **JACK**, **QUEEN**, **KING**). The values of the rank constants should be 1, 11, 12, and 13, respectively.
- A constructor that takes a rank and a suit and returns a **Card** with those values.
- Accessor methods **getRank** and **getSuit** to retrieve the rank and suit components of a card.
- An implementation of the **toString** method that returns the complete name of the card as in exercise 1. Remember that you can use the **+** operator to connect the different parts of the string together, as shown in the **toString** implementation for the **Rational** class in Figure 6-9.

8. Write the definition for a new class named **Employee** that stores the following data for a single employee:

    - The name of the employee (a **String**)
    - The name of the employee's supervisor (also a **String**)
    - The employee's annual salary (a **double**)

    As with the other classes defined in this book, you should make sure that the instance variables containing these values are private and instead provide **get** and **set** methods to retrieve or change any of the values. For example, if the variable **emp** is an employee, you should be able to retrieve the name of the employee by writing

    ```
    emp.getName()
    ```

    or double the employee's salary by writing

    ```
    emp.setSalary(emp.getSalary() * 2);
    ```

    You should also make sure that there is a constructor method that allows clients to initialize all these fields when a new **Employee** object is created, as in

    ```
    new Employee("Bob Cratchit", "Ebenezer Scrooge", 25.00)
    ```

9. The implementation of the **Rational** class given in this chapter is not particularly useful in practice because it doesn't allow the numerator and denominator to exceed the size of an integer, even though larger values tend to occur quite often in rational arithmetic. One way to avoid the problem is to use the **BigInteger** class in the **java.math** package, which defines an extended type of integer that can take on arbitrarily large values. Rewrite the implementation of **Rational** so that the private instance variables **num** and **den** are declared as **BigInteger**s instead of **int**s, but without changing the argument and result types of any of the public methods. To learn how **BigInteger** works, consult the **javadoc** page.

Chapter 7
The Object Memory Model

A teacher who can arouse a feeling for one single good action, for one single good poem, accomplishes more than he who fills our memory with rows on rows of natural objects, classified with name and form.

— Goethe, *Elective Affinities*, 1808

Jay Forrester

After growing up on a Midwestern cattle ranch without electricity, Jay Forrester studied electrical engineering at the University of Nebraska and MIT, where he became director of the Navy's Project Whirlwind in 1944. Along with the ENIAC system at the Moore School in Philadelphia and the MARK I system at Harvard, Whirlwind played a central role in the early history of computers as they evolved from earlier analog designs to the digital systems that are standard in the industry today. Forrester's most significant contribution to computer hardware design was the development of core memory, in which small ferrite disks could be magnetized in one direction or the other to represent a binary 0 or 1. Magnetic core memory revolutionized hardware designs and was used in essentially all computers until it was replaced by integrated-circuit memory in the late 1970s. In 1956, Forrester joined the faculty of the Sloan School of Management, where he founded the new discipline of system dynamics, which attempts to focus holistically on large-scale systems and their interactions rather than looking only at their individual parts.

One of the enormous advantages of high-level languages like Java is that they free you from having to worry about the details of low-level representation inside the hardware of the machine. Managing the details of the internal representation is typically time-consuming, tedious, and susceptible to error; having the language keep track of these details can therefore increase programmer productivity. The problem, however, is that it is much more difficult to understand the programming process without having some idea of how that internal representation works. The structures that programmers have historically used—and that continue to form the foundation of languages like Java—reflect the capabilities of the hardware on which programs run. Knowing something about those low-level machine capabilities provides a conceptual framework for understanding programming at a higher, more abstract level.

Developing an intuitive feel for how data representation works inside the machine is particularly important when you begin to work with objects instead of primitive data. In Java, passing an object as an argument to a method seems very different from the corresponding process of using primitive types as arguments. If you understand how objects are represented inside the computer, however, the reasons for that apparent difference become much clearer and in fact turn out to be far more consistent than it might initially appear.

7.1 The structure of memory

In Chapter 1, you had the opportunity to learn a little bit about the internal structure of a computer, but only at a very abstract level. Figure 1-1 identified the major hardware components of a typical computer system including the CPU, memory, secondary storage, and I/O devices. To develop a mental model of how objects are stored inside a computer, it is important to look at the structure of the memory system in more detail.

Bits, bytes, and words

At the most primitive level, all data values inside the computer are stored in the form of fundamental units of information called *bits*. A **bit** records the simplest possible value, which can be in one of two possible states. If you think of the circuitry inside the machine as if it were a tiny light switch, you might label those states as *on* and *off*. Because the word *bit* comes originally from a contraction of *binary digit*, it is more common to label those states using the values 0 and 1, which are the two digits used in the binary number system on which computer arithmetic is based.

Since a single bit holds so little information, the individual bits do not provide a convenient mechanism for storing data. To make it easier to store such traditional types of information as numbers or characters, bits are collected together into larger units that are then treated as integral units of storage. The smallest combined unit is called a **byte** and is composed of eight individual bits. On most machines, bytes are assembled into larger structures called **words**—defined to be four bytes in Java—that are large enough to hold an integer value.

The amount of memory available to a particular computer varies over a wide range. Early machines supported memories whose size was measured in kilobytes (KB); today's machines have memory sizes measured in megabytes (MB) or even gigabytes (G172es *kilo, mega,* and *giga* stand for one thousand, one million, and one billion, respectively. In the world of computers, however, those base-10 values do not fit well into the internal structure of the machine. By tradition, therefore, these prefixes are taken to represent the power of two closest to their traditional interpretations. Thus, in programming, the prefixes *kilo, mega,* and *giga* have the following meanings:

$$\begin{aligned} \text{kilo (K)} \ &= \ 2^{10} \ = & 1{,}024 \\ \text{mega (M)} &= \ 2^{20} \ = & 1{,}048{,}576 \\ \text{giga (G)} \ &= \ 2^{30} \ = & 1{,}037{,}741{,}824 \end{aligned}$$

A 64KB computer from the early 1970s would have had 64 × 1024 or 65,536 bytes of memory. Similarly, a 512MB machine would have 512 × 1,048,576 or 536,870,912 bytes of memory.
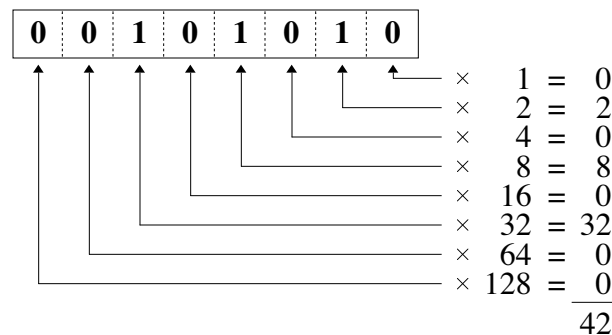
Binary and hexadecimal representations

Each of the bytes inside a machine holds data whose meaning depends on how the system interprets the individual bits. Depending on the hardware instructions that are used to manipulate it, a particular sequence of bits can represent an integer, a character, or a floating-point value, each of which requires some kind of encoding scheme. The easiest encoding scheme to describe is that for integers, in which the bits are used to represent an integer represented in **binary notation**, in which the only legal values are 0 and 1, just as is true for the underlying bits. Binary notation is similar in structure to our more familiar decimal notation, but uses 2 rather than 10 as its base. The contribution that a binary digit makes to the entire number depends on its position within the number as a whole. The rightmost digit represents the units field, and each of the other positions counts for twice as much as the digit to its right.

Consider, for example, the eight-bit byte containing the following binary digits:

0	0	1	0	1	0	1	0
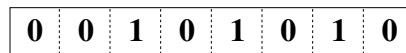
That sequence of bits represents the number forty-two, which you can verify by calculating the contribution for each of the individual bits, as follows:

0	0	1	0	1	0	1	0

$$\begin{aligned} \times \quad 1 \ &= \ 0 \\ \times \quad 2 \ &= \ 2 \\ \times \quad 4 \ &= \ 0 \\ \times \quad 8 \ &= \ 8 \\ \times \quad 16 \ &= \ 0 \\ \times \quad 32 \ &= \ 32 \\ \times \quad 64 \ &= \ 0 \\ \times \ 128 \ &= \ \underline{\ \ 0\ \ } \\ & \quad \ 42 \end{aligned}$$

This diagram illustrates how to map an integer into bits using binary notation, but is not a particularly convenient representation. Binary numbers are cumbersome, mostly because they tend to be so long. Decimal representations are intuitive and familiar, but make it difficult to understand how the number translates into bits. For those applications in which it is useful to understand how a number translates into its binary representation without having to work with binary numbers that stretch all the way across the page, computer scientists tend to use **hexadecimal** (base 16) representation instead.

In hexadecimal notation, there are sixteen digits, representing values from 0 to 15. The decimal digits 0 through 9 are perfectly adequate for the first ten digits, but classical arithmetic does not define the extra symbols you need to represent the remaining six digits. Computer science traditionally uses the letters **A** through **F** for this purpose, as follows:

Hex digit	Value
A	10
B	11
C	12
D	13
E	14
F	15

What makes hexadecimal notation so attractive is that you can quickly move back and forth between hexadecimal values and the underlying binary representation. All you need to do is combine the bits into groups of four. For example, the number forty-two can be converted from binary to hexadecimal like this:

$$0 \quad 0 \quad 1 \quad 0 \quad | \quad 1 \quad 0 \quad 1 \quad 0$$

$$2 \qquad\qquad A$$
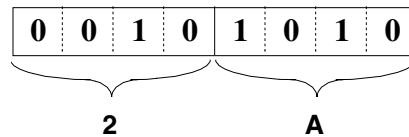
The first four bits represent the number 2, and the next four represent the number 10. Converting each of these to the corresponding hexadecimal digit gives **2A** as the base-16 form. You can then verify that this number still has the value 42 by adding up the independent digit values, as follows:

$$
\begin{aligned}
2 \quad A & \\
10 \times 1 &= 10 \\
2 \times 16 &= 32 \\
\hline
&\ 42
\end{aligned}
$$

For the most part, the numeric representations that appear in this book use decimal notation for readability. If the base is not clear from the context, the text follows the usual strategy of using a subscript to denote the base. Thus, the three most common representations for the number forty-two—decimal, binary, and hexadecimal—look like this:

$$42_{10} \ = \ 101010_2 \ = \ 2A_{16}$$

The key point is that the number itself is always the same; the numeric base affects only the representation. Forty-two has a real-world interpretation that is independent of the base. That real-world interpretation is perhaps easiest to see in the representation an elementary school student might use, which is after all just another way of writing it down:

$$\cancel{||||}\ \cancel{||||}\ \cancel{||||}\ \cancel{||||}\ \cancel{||||}\ \cancel{||||}\ \cancel{||||}\ \cancel{||||}\ ||$$

The number of line segments in this representation is forty-two. The fact that a number is written in binary, decimal, or any other base is a property of the representation and not of the number itself.

Memory addresses

Within the memory system of a typical computer, every byte is identified by a numeric **address.** The first byte in the computer is numbered 0, the second is numbered 1, and so on, up to the number of bytes in the machine. For example, you can diagram the memory bytes of a tiny 64KB computer as shown along the right margin of this page. The numbering scheme, however, may seem unfamiliar at first, but only because the addresses are written using hexadecimal rather than decimal values. Hexadecimal notation was introduced in the preceding section, along with techniques for converting a hexadecimal value to its binary or decimal equivalent. The important thing to remember is that the addresses are simply numbers and that the base is relevant only to how those numbers are written down. The final address in the diagram is the address **FFFF**, which corresponds to the decimal value 65535. It would have been easy to write the addresses in decimal notation, which would have made them a little easier to read as numbers. This text, however, uses hexadecimal for the following reasons:

1. Address numbers are conventionally written in hexadecimal, and Java debuggers and runtime environments will generally display addresses in this form.

2. Writing address numbers in their hexadecimal form and using a sans-serif font makes it easier to recognize that a particular number represents an address rather than some unidentified integer.

3. Using hexadecimal values makes it easier to see why particular limits are chosen. When you write it as a decimal value, the number 65535 seems like a rather random value. If you express that same number in hexadecimal as **FFFF**, it becomes easier to recognize that this value is the largest value that can be represented in 16 bits.

4. Making the numbers seem less familiar may discourage you from thinking about them arithmetically. Java does a wonderful job of hiding the underlying address calculations from the programmer. What is important to understand is that addresses are represented as numbers. It is completely unimportant—and indeed usually impossible to determine—what those numbers are.

In earlier days, byte values were useful in and of themselves because characters could be represented using a single byte. Today, bytes are too small to represent the large character sets found in a modern computer. The Unicode representation discussed in Chapter 8, for example, requires two bytes for each character. A value of type **char** therefore takes up two of these byte-sized units in memory, as illustrated by the shaded bytes in the following diagram:

	FFF0
	FFF1

Data values requiring multiple bytes are identified by the address of the first byte, so that the character represented by the shaded area is considered to be at address **FFF0**. As a second example, values of type **double** require eight bytes of memory, so that a variable of type **double** stored at address **1000** would take up all the bytes between addresses **1000** and **1007**, inclusive:

0000
0001
0002
0003
0004
0005
0006
0007
0008
0009
000A
000B
000C
000D
000E
000F
0010
0011
0012
0013
0014
0015
0016
0017
0018
0019

⋮

1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
100A
100B
100C
100D
100E
100F

⋮

FFF0
FFF1
FFF2
FFF3
FFF4
FFF5
FFF6
FFF7
FFF8
FFF9
FFFA
FFFB
FFFC
FFFD
FFFE
FFFF

Memory

	1000
	1001
	1002
	1003
	1004
	1005
	1006
	1007
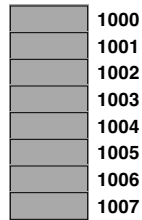
Although addresses in memory are usually specified in terms of bytes, the fact that bytes are too small to be of much use has led hardware designers to group consecutive bytes into larger units. Typical architectures today include instructions for manipulating memory units in the following sizes: one byte (8 bits), two bytes (16 bits), four bytes (32 bits), and eight bytes (64 bits). These units correspond to the built-in Java types **byte**, **short**, **int**, and **long**. The four-byte unit used to represent an integer is conventionally called a **word.** Because words tend to be much more useful, it is common to draw memory diagrams in units of words rather than the individual bytes. The **double** stored in the bytes numbered between **1000** and **1007** could therefore be diagrammed more compactly in the following word-based form:

	1000
	1004
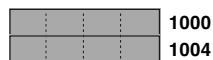
7.2 Allocation of memory to variables

Whenever you declare a variable in a program, the compiler must reserve memory space to hold its value. The process of reserving memory space is called **allocation**. In Java, memory is allocated from one of three different sources depending on how it is declared, each of which is illustrated in the diagram to the right. The three allocation strategies are as follows:

memory for program code and static data

1. *Static variables and constants*. Any variable whose declaration includes the **static** keyword applies to the class as a whole and not to an individual object. These variables are typically allocated at the beginning of the memory space allocated to the program alongside the memory cells used to store the instructions for the program. In the examples included in this book, the only static declarations are those for named constants.

pool of memory available for objects (the "heap")

2. *Dynamically allocated objects*. All objects created using the **new** keyword are assigned storage from a region of memory called the **heap.** Although there is no explicit rule that enforces this convention, most implementations of the Java Virtual Machine assign heap memory beginning immediately after the fixed region assigned to the static declarations in a class, as illustrated in the diagram. The advantage of doing so is that this allocation strategy ensures that the program can take advantage of as much memory as possible. Because the stack and the heap start at opposite ends of memory and grow toward each other, neither will run out until all available space is exhausted.

the heap grows toward higher addresses

the stack grows toward lower addresses

3. *Local variables*. All variables that are declared as local variables inside a method are allocated from a region of memory called the **stack,** which is typically implemented at least partially in hardware. In modern architectures, the stack begins at the highest legal address in memory and grows toward lower addresses as new methods are

memory for local variables (the "stack")

Memory regions
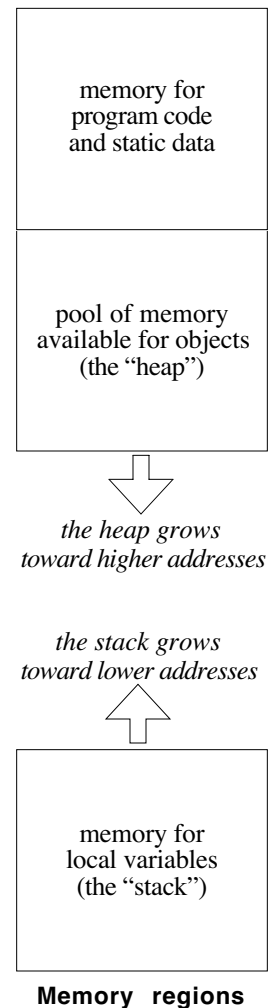
called. Calling a method increases the size of the stack by the amount needed to hold the local variables that method declares. Collectively, the memory assigned to the local variables for a method are called a **stack frame.** When a method returns, its stack frame is discarded, restoring the frame of its caller.

It is important to recognize that declaring an object variable is likely to reserve memory in both the stack and the heap. The local variable used to store the object appears on the stack just like any local variable. That variable, however, holds only the memory address of the object for which the actual data is stored in the heap. As noted in Chapter 6, that address is called a **reference.**

At this point, it is useful to consider a simple example to illustrate the relationship between heap storage and stack storage. Suppose that you have decided to use the `Rational` class introduced in section 6.3, which allows for a precise representation of fraction numbers. Internally, each `Rational` object declares two instance variables, `num` and `den`, to store the numerator and denominator of the fraction, respectively. The important question is then what happens inside the computer when you execute a declaration such as the following:

```
Rational half = new Rational(1, 2);
```

The variable `half` is a local variable and therefore appears in the stack frame for its caller. That variable, however, contains only enough storage to hold a memory address, which is typically four bytes on a machine with 32-bit addressing, although the growing number of 64-bit architectures means that the memory required will presumably grow to eight bytes over time. In any event, the amount of memory that appears on the stack is small. No matter how large the object itself turns out to be, a *reference* to that object can be stored in a single word.

The complete picture of what happens during the execution of this Java statement requires looking at what happens in both the heap and the stack. The evaluation of the initial-value expression on the right-hand side of the equal sign creates a new object in the heap. The heap storage for that object contains space for the integer values `num` and `den` along with some additional information common to all objects, which is used by the Java runtime system to manage objects in memory. From your perspective as a programmer, the details of that additional information are of no consequence, but it sometimes helps to know that objects take up more memory than you might expect from looking at the memory requirements of their instance variables. In this book, this extra management information is called the **object overhead** and is represented as a shaded area in memory diagrams. For example, the object created by the call to `new Rational(1, 2)` might look like this:

	overhead	**1000**
num	1	
den	2	

Once the new `Rational` object has been allocated in the heap, the next step in the declaration process is to store the address of that object in the variable `half`. That variable is allocated on the stack and therefore appears in a stack frame somewhere toward the high end of memory on most machines. Although there is no way to know what other values might exist in the stack frame without seeing the code for the entire method, the word assigned to `half` might look like this:
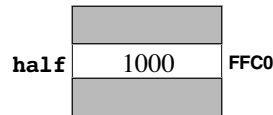
The value 1000 shown in the variable `half` is not intended as an integer but rather to signify the address at which the actual object is located. The choice of 1000 for that address—along with the addresses of every other heap and stack variable used in this book—is entirely arbitrary. In general, there is no way for you as a Java programmer to know the address of an object, nor is there any reason for you to do so. It is, however, important for you to remember that every object has some address and that Java programs can gain access to the data in the object by keeping track of the address at which that object lives.

Even after you become more familiar with objects, you will discover that it often helps to create diagrams that make it easier to follow what's happening inside the memory. To illustrate this process, it is useful to try a somewhat more elaborate example. Consider the following `run` method, which first appeared in Chapter 6:

```
public void run() {
    Rational a = new Rational(1, 2);
    Rational b = new Rational(1, 3);
    Rational c = new Rational(1, 6);
    Rational sum = a.add(b).add(c);
    println(a + " + " + b + " + " + c + " = " + sum);
}
```

The idea of this program, of course, is to add three `Rational` variables and display their sum. Our concern here, however, is simply to understand how these values are represented inside the machine.

The first three declarations are easy enough. As in the simple example of the variable `half`, each of these declarations constructs a new `Rational` object and assigns its address to a variable in the stack frame for the `run` method, giving rise to a memory diagram that looks like this:
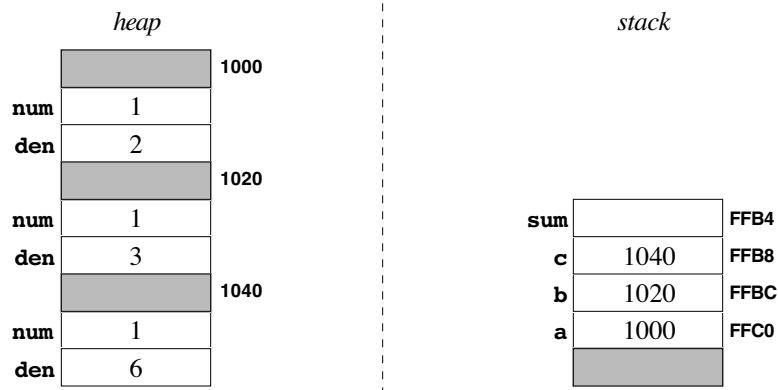
As in most languages, Java creates the complete stack frame for a method at the time the method is called, so that all four local variables—`a`, `b`, `c`, and `sum`—have been allocated on the stack, but only the first three have been initialized at this point in the execution. The value used to initialize `sum` is the result of the expression

```
a.add(b).add(c)
```

It is worth tracing through this operation step by step.

   The first step in the process is to invoke the **add** method on the **Rational** object **a**, passing it the value of **b** as a parameter. Because this operation is a method call, a new stack frame must be created for it, binding the formal parameters of the method to the values of the actual arguments included in the call. As you can see from Figure 6-9, the code for the **add** method appears as follows:

```
public Rational add(Rational r) {
    return new Rational(this.num * r.den + r.num * this.den,
                        this.den * r.den);
}
```

The formal parameter is named **r** inside the **add** method, which means that **r** must be assigned the value of the actual argument, which is named **b** in the calling frame. The value currently sitting in the variable **b** is the number 1020, which is the address of the corresponding object in the heap. Java initializes the variable **r** in the new frame simply by copying the address, which means that the variable **r** in the new frame will also contain 1020.

   For this example, it is important to note that the stack frame for any method invoked on a receiving object also contains a reference to that object, which Java identifies using the keyword **this**. Intuitively, it is best to think of **this** as simply a local variable that refers to the receiving object, which in this case is the value stored in the variable **a**. As with the argument value, the identity of the receiver is also copied into the new frame as a simple address. Thus, the state of the memory once the new frame has been created looks like this:
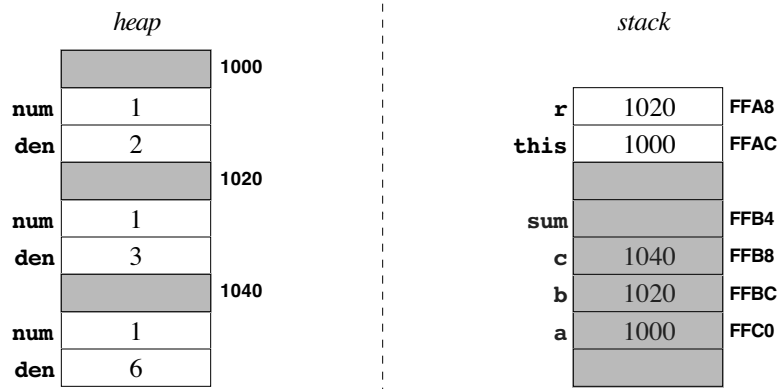
In this diagram, the stack region below the current frame is shown in gray to emphasize that the shaded region is inaccessible to the **add** method. The only variables accessible to **add** are the references stored in the variables **r** and **this** in the current frame. These reference variables are exactly what the **add** method needs to perform its computation. The **add** method returns a new **Rational** value whose numerator is given by the expression

        this.num * r.den + r.num * this.den

and whose denominator has the value

        this.den * r.den

Each term in these expressions is a selection operation that extracts one of the two fields, **num** or **den**, from the appropriate **Rational** value. For example, the term **r.den** specifies the selection of the **den** field of the value whose reference is stored in **r**. Since **r** contains

the address 1020, the term **r.den** indicates the denominator component of the object stored at address 1020, which is the integer 3. Expanding the value for each term in a similar fashion reveals that the result of the **add** method is given by the expression

```
new Rational(1 * 3 + 1 * 2, 2 * 3)
```

which can be further simplified to

```
new Rational(5, 6)
```

This expression constructs a new **Rational** object with the indicated components in the heap and returns the address of that object as the value of the call **a.add(b)**, clearing away the stack frame for **add** as it returns.

Unlike the result of most of the computations you have seen so far, the new value computed by this method does not get stored in a variable. The calculation **a.add(b)** is only a subexpression of the longer expression

```
a.add(b).add(c)
```

The value of **a.add(b)** therefore becomes the receiver of a new **add** call with the argument **c**. Setting up the frame for this new method call leaves memory in the following state, with the new **Rational** object shown at address 1060 in the heap:

	heap			*stack*	
		1000			
num	1				
den	2				
		1020			
num	1		r	1040	FFA8
den	3		this	1060	FFAC
		1040			
num	1		sum		FFB4
den	6		c	1040	FFB8
		1060	b	1020	FFBC
num	5		a	1000	FFC0
den	6				
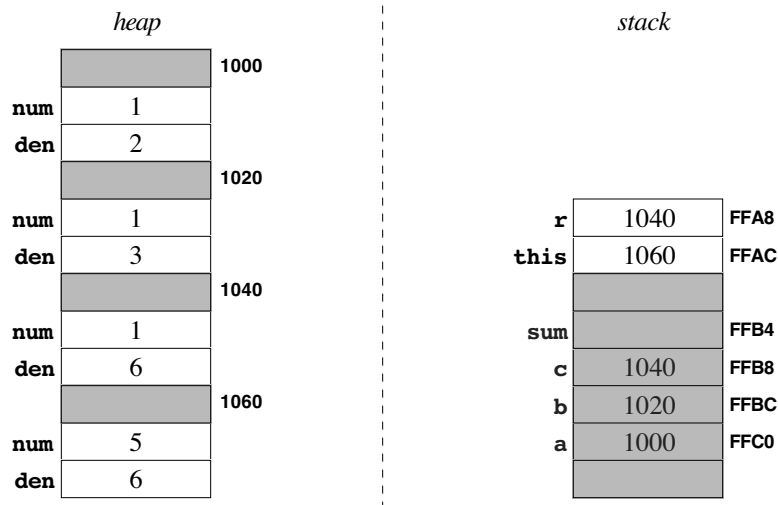
Once again, the evaluation of the **add** method requires determining the values of the fields in the two **Rational** values accessible from the current frame. This time, the straightforward expansion of the result expression gives

```
new Rational(5 * 6 + 1 * 6, 6 * 6)
```

which works out to

```
new Rational(36, 36)
```

Fortunately, the constructor for **Rational** provides additional simplification of the result by using Euclid's algorithm to reduce the fraction to lowest terms. Eliminating the common factor of 36 from both the numerator and the denominator gives rise to a new **Rational** object with 1 as the value of both its **num** and **den** field. When that value is assigned to the variable **sum**, the heap and stack look like this:

heap

stack

		1000
num	1	
den	2	
		1020
num	1	
den	3	
		1040
num	1	
den	6	
		1060
num	5	
den	6	
		1080
num	1	
den	1	

sum	1080	**FFB4**
c	1040	**FFB8**
b	1020	**FFBC**
a	1000	**FFC0**
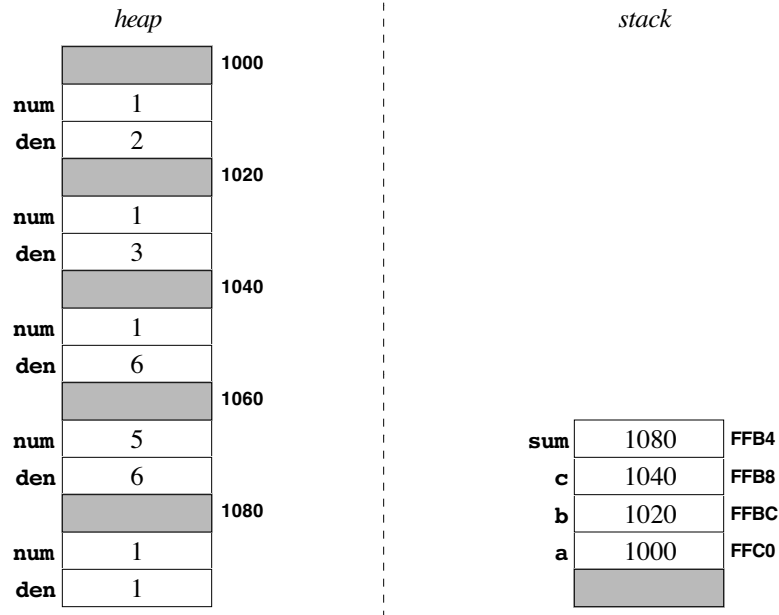
Garbage collection

The example from the preceding section illustrates an interesting wrinkle that often arises when objects are stored in memory. If you look carefully at the most recent diagram, you will see that the heap contains five different **Rational** objects even though only four of those objects have references on the stack. The fraction 5/6 stored at address 1060 was created during the calculation as the intermediate result from the subexpression

```
        a.add(b)
```

While that value is necessary to complete the calculation, it becomes irrelevant once the final addition is performed. In the traditional parlance of computer science, that value is now **garbage.** Calculations in object-oriented languages often generate quite a bit of garbage along the way as intermediate results are computed as part of complex computations. Unfortunately, those values occupy space in the heap. If that garbage is allowed to persist, the heap will eventually fill up, even if there is plenty of room for the active data.

To get around the problem of having the heap fill up with objects that are no longer useful, the Java runtime system adopts a strategy called **garbage collection,** which does pretty much exactly what it sounds like it does. Whenever the memory available in the heap seems to be running short, Java defers what it's doing to collect any garbage in the heap and return that memory to the pool of assignable storage. To do so, Java uses a two-phase process that operates in more or less the following way, although the strategy presented here has been simplified to ensure that the basic idea remains clear:

1. Go through every variable reference on the stack and every variable reference in static storage and mark the corresponding object as being "in use" by setting a flag that lives in the object's overhead space. In addition, whenever you set the "in use" flag for an object, determine whether that object contains references to other objects and mark those objects as being "in use" as well.
2. Go through every object in the heap and delete any objects for which the "in use" flag has not been set, returning the storage used for that object to the heap for subsequent reallocation. It must be safe to delete these objects, because there is no way for the

program to gain access to them, given that no references to those objects still exist in the program variables.

Because the garbage-collection algorithm operates in two phases—one to mark the objects that are in use and one to sweep through memory collecting inaccessible objects—this strategy is traditionally called a **mark-and-sweep** collector.

7.3 Primitive types vs. objects

From a high-level conceptual perspective, you can often think about Java objects in much the same way that you think about the primitive types like **int** and **double**. If you declare an integer variable by writing

```
int three = 3;
```

you can diagram that variable by drawing a box named **three** and writing a 3 in the box, as follows:

three

3
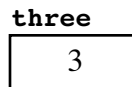
To a certain extent, you can treat objects in the same way. For example, if you create a new **Rational** variable using the declaration

```
Rational third = new Rational(1, 3);
```

you can think about this variable conceptually as if it held the actual value as shown in the following diagram:

third

$\frac{1}{3}$
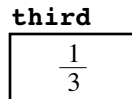
It is, however, important to understand at what point this symmetry breaks down. As you know from the discussion of internal representation earlier in this chapter, the variable **third** does not in fact contain the actual object but instead contains the address of that object. This mode of representing objects has several implications that it is important for you, as a budding programmer, to understand.

Parameter passing

One of the most important consequences of the representation scheme that Java uses for objects arises when you pass an object as a parameter to a method. At first glance, the rules for passing objects as parameters seem different from the corresponding rules for passing primitive values. Suppose, for example, that you have written the following pair of methods:

```
public void run() {
   int x = 17;
   increment(x);
   println("x = " + x);
}

private void increment(int n) {
   n++;
   println("n = " + n);
}
```

Running this program produces the following output:

As you can see from the output, the **++** operator in the **increment** method affects only the value of the local variable **n** and not the value of **x** in the calling frame. The behavior simply reflects the way these values are stored on the stack. When **increment** is called, the value from the stack entry for the local variable **x** is copied into a new stack entry for the variable **n**, so that the stack looks something like this:
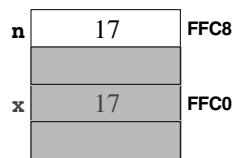
Changing the value of **n** inside the new frame has no effect on the original value.

But what would happen if these values were objects instead of simple integers? One way to answer that question is to design a simple class that contains an embedded integer value, as shown in Figure 7-1. The definition of the **EmbeddedInteger** class includes a constructor, a **setValue** method for setting the internal value, a **getValue** method for getting that value back, and a **toString** method to make it easier to display the object using **println**. Given these methods, you can write a similar test program that looks like this:

```
public void run() {
   EmbeddedInteger x = new EmbeddedInteger(17);
   increment(x);
   println("x = " + x);
}

private void increment(EmbeddedInteger n) {
   n.setValue(n.getValue() + 1);
   println("n = " + n);
}
```

If you run this version of the program, the output is different:
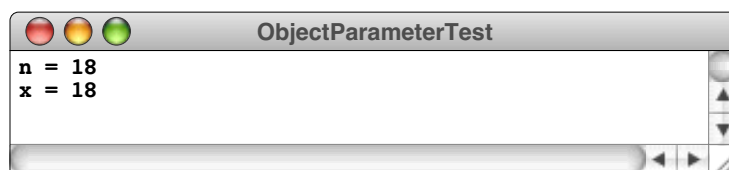
In the object-based implementation, changing the value of a field in the underlying object continues to be reflected after the method returns. This change in behavior follows directly from the memory structure. At the beginning of the call to **increment** in the new implementation, the heap and stack are in the following state:

FIGURE 7-1   Implementation of the EmbeddedInteger class

```
/* Class: EmbeddedInteger */
/**
 * This class allows its clients to treat an integer as an object.
 * The underlying integer value is set using setValue and returned
 * using getValue.
 */
public class EmbeddedInteger {

/* Constructor: EmbeddedInteger(n) */
/**
 * Creates an embedded integer with the value n.
 */
   public EmbeddedInteger(int n) {
      value = n;
   }

/* Method: setValue(n) */
/**
 * Sets the internal value of this EmbeddedInteger to n.
 */
   public void setValue(int n) {
      value = n;
   }

/* Method: getValue() */
/**
 * Returns the internal value of this EmbeddedInteger.
 */
   public int getValue() {
      return value;
   }

/* Method: toString() */
/**
 * Overrides the toString method to make it return the string
 * corresponding to the internal value.
 */
   public String toString() {
      return "" + value;
   }

/* Private instance variable */

   private int value;       /* The internal value */

}
```

heap   *stack*
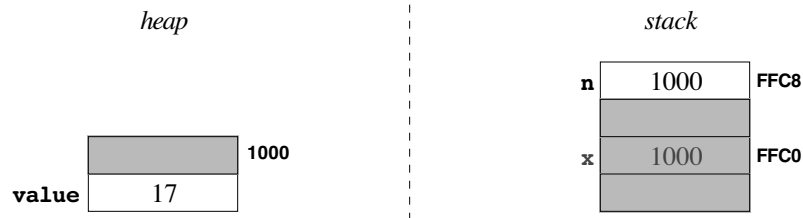
In this case, changing the **value** field inside the object affects the conceptual value of both **n** and **x** because those two variables contain the same reference.

The intuitive effect of this difference in representation is that objects—in contrast to primitive values—are shared between the calling method and the method being called. The underlying mechanism, however, is exactly the same. Whenever a new local variable is initialized, the old value is copied into the stack location for the new variable. If that value is an object, what gets copied is the reference and not the underlying value.

Wrapper classes

The **EmbeddedInteger** class shown in Figure 7-1 is actually somewhat more than a simple illustration of parameter-passing in the object world. As you will discover in subsequent chapters, the Java library packages contain quite a few classes that are designed to work with any type of object. For example, the **ArrayList** class that you will learn about in Chapter 10 makes it possible to maintain an ordered list of objects. Given an **ArrayList**, you can add new values, remove existing ones, and perform a number of other useful operations that make sense in the abstract context of a list. The nature of the operations that manipulate the list turn out to be independent of the type of value the list contains. Thus, to make it possible to store any kind of object at all, the **ArrayList** class uses the universal class **Object** from which all other classes in the Java class hierarchy descend.

Unfortunately, being able to store any kind of object in an **ArrayList** does not provide quite as much flexibility as you might like. The Java primitive types are not objects and therefore cannot be used in conjunction with this marvelously convenient class. To get around this problem, Java defines a class, usually called a **wrapper class,** to go along with each of the eight primitive types, as follows:

Primitive type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Each of the wrapper classes has a constructor that creates a new object from the corresponding primitive type. For example, if the variable **n** is an **int**, the declaration

```
Integer nAsInteger = new Integer(n)
```

creates a new **Integer** object whose internal value is **n** and assigns it to **nAsInteger**. Each of the wrapper classes also defines an accessor method to retrieve the underlying value. The name of that method is always the name of the primitive type followed by the suffix **Value**. Thus, once you had initialized the variable **nAsInteger**, you could get back the value stored inside it by writing

```
nAsInteger.intValue()
```

Because **nAsInteger** is a legitimate object, you can store that value in an **ArrayList** or any other compound structure.

Unlike the **EmbeddedInteger** class from the preceding section, however, the wrapper classes provide no method to set the value of the underlying variable in an existing object. The wrapper classes are in fact immutable, as defined in Chapter 6. Immutable classes have the wonderful property that it is always possible to think about them as if they represent pure values in the way that Java's primitive types do. With an immutable class, you don't have to worry whether a value is shared or copied when you pass it from one method to another. Because neither side can change that value, it turns out not to matter.

The wrapper classes also include several static methods that you are likely to find useful, particularly those in the **Integer** class that support numeric conversion in arbitrary bases. Figure 7-2 lists a few of the most important static methods for the classes **Integer** and **Double**; the even more useful static methods in the **Character** class are described in Chapter 9. Because these methods are static, you need to include the name of the class. Thus, to convert the integer 50 into a hexadecimal string, you would need to write

```
Integer.toString(50, 16)
```

7.4 Linking objects together

The fact that objects are represented internally as references has one more implication that seems important to consider in the context of this chapter, even though a full discussion of the topic must wait until much later in the book. Because references are small, an object can easily contain references to other objects. If objects were stored only in the complete form in which they appear in the heap, it would obviously be impossible for a small object to contain a larger one. Moreover, given that every object requires some amount of overhead, it would even be impossible for one heap object to contain another object of the same size, because there would no longer be any room in which to store the bookkeeping information that Java needs. Those restrictions, however, disappear

---

FIGURE 7-2   **Numeric conversion methods in the Integer and Double classes**

Static methods in the **Integer** class

static int parseInt(String str)   *or*   **parseInt(String str, int radix)**
Converts a string of digits into an **int** using the specified radix, which defaults to 10.
static String toString(int i)   *or*   **toString(int i, int radix)**
Converts an **int** to its string representation for the specified radix, which defaults to 10.

Static methods in the **Double** class

static double parseDouble(String str)
Converts the string representation of a number into the corresponding **double**.
static String toString(double d)
Converts a **double** to its string representation.

completely if objects are stored as references. Although it is true that a small object cannot physically contain a larger object, there is nothing to prevent one object from containing a reference to another object no matter how much heap storage those objects might consume. A reference, after all, is simply the address of an object in memory and can therefore be represented using just a few bytes. Thus, there is no problem in having one object contain a reference to another object of a different class, or even an object of the same class.

Creating an object that contains references to other objects is an extremely powerful programming technique. Such objects are said to be **linked.** Although an in-depth discussion of linked structures lies beyond the scope of this chapter, it is useful to present a simple example, both because that example will reinforce your understanding of how linked structures are represented in memory and because it offers a powerful illustration of the fundamental concept of **message-passing,** which is the metaphor Java uses to describe communication among the objects in a system. When an object needs to interact with some other object, it does so by invoking a method in the receiving object. In the context of the object-oriented paradigm, however, the act of calling that method is usually described as one of "sending a message" to the receiver. That message may in turn trigger a response in the receiver, such as a change in its state or the generation of additional messages that propagate that message to other objects. Even though the implementation of message-passing depends on method calls, it is important to keep the underlying metaphor in mind.

Message passing in linked structures: The beacons of Gondor

> *For answer Gandalf cried aloud to his horse. "On, Shadowfax! We must hasten. Time is short. See! The beacons of Gondor are alight, calling for aid. War is kindled. See, there is the fire on Amon Dîn, and flame on Eilenach; and there they go speeding west: Nardol, Erelas, Min-Rimmon, Calenhad, and the Halifirien on the borders of Rohan."*
>
> — J. R. R. Tolkien, *The Return of the King,* 1955

In adapting this scene for the concluding episode in his trilogy, Peter Jackson created what may be the most evocative and dramatic example of message-passing ever recorded on film. After the first beacon is lit in the towers of Minas Tirith, we see the signal pass from mountaintop to mountaintop as the keepers of each signal tower, ever vigilant, light their own fires as they see the triggering fire at the preceding station. The message of Gondor's danger thus passes quickly over the miles that separate it from Rohan, as illustrated by the following schematic diagram:

Minas Tirith   Amon Dîn   Eilenach   Nardol   Erelas   Min-Rimmon   Calenhad   Halifirien   Rohan

How would you go about simulating the lighting of the beacons of Gondor using the method-passing paradigm? Each signal tower is presumably a separate object that contains such information as the name of the tower. Each tower, however, must also be able to identify the next tower in the chain so that it knows where to send its message. The most straightforward strategy is to have each tower contain a reference to its successor. Thus, the object that represents Minas Tirith contains a reference to the object used to model Amon Dîn, which in turn contains a reference to the object that represents Eilenach, and so on. If you adopt this approach, the private data for each object must include—possibly along with other information required by the application as a whole— the following instance variables:

```
private String towerName;
private SignalTower nextTower;
```

The first variable contains the name of the tower, and the second is a reference to the next tower in sequence.

While this model makes sense for Minas Tirith and the intermediate towers in the chain, using this strategy to represent Rohan raises a minor issue by virtue of the fact that Rohan appears at the end of the chain. Like the other towers, Rohan has a **nextTower** field that is supposed to contain a reference to the next tower. Rohan doesn't have one, and the question is simply how to represent this fact. Fortunately, Java defines a special value called **null** for precisely this situation. The constant **null** represents a reference to a nonexistent value and can be assigned to any variable that holds an object reference. Thus, you can record the fact that Rohan is the last tower in the chain simply by making sure that its **nextTower** field is **null**.

Figure 7-3 contains a simple definition of a **SignalTower** class that implements the idea of passing a message along a chain of towers. The constructor takes the name of the tower and a reference to the next tower in sequence. The only other public method is **signal**, which is the action associated with the "Light your signal fire" message that propagates through the towers. When a tower receives the **signal** message, it lights its own signal fire and then passes the message along by sending the **signal** message to its successor, if any.

In the design of the **SignalTower** class shown in Figure 7-3, the process of lighting the current signal tower is the responsibility of the **lightCurrentTower** method. In this implementation, **lightCurrentTower** consists of the code

```
public void lightCurrentTower() {
    System.out.println("Lighting " + towerName);
}
```

which prints a message including the name of the tower on **System.out**, which is a standard output stream that is always available to Java programs. Unless your signal tower application is acting only as a test program, simply displaying the name of the tower is probably not what you want to do at this point. For example, if you were writing a graphical implementation of the "Beacons of Gondor" program—as you will have a chance to do in the exercises for Chapter 8—the implementation of this method would have to take whatever actions were necessary to display the signal fire on the screen. Given that the point here is to illustrate how including references as links inside each tower makes it possible to propagate the signal from one tower to the next, the additional detail required for a graphical implementation would only get in the way.

Even though the **lightCurrentTower** implementation shown has an effect that is useful only in test programs, it still makes sense, particularly in the object-oriented paradigm, to include it as a separate method in the class. A simple implementation that you intend to replace later is usually called a **stub.** If you got around to writing a more elaborate application, you could simply replace this stub with the appropriate code. But you could probably do better still to create a new subclass that overrides the definition of **lightCurrentTower**. The extended class can substitute its own version of this method but still rely on all the other features provided by the base class for linking the towers together.

FIGURE 7-3 Implementation of the SignalTower class

```java
/* Class: SignalTower */
/**
 * This class defines a signal tower object that passes a message
 * to the next tower in a line.
 */

public class SignalTower {

/* Constructor: SignalTower(name, link) */
/**
 * Constructs a new signal tower with the following parameters:
 *
 * @param name The name of the tower
 * @param link A link to the next tower, or null if none exists
 */
   public SignalTower(String name, SignalTower link) {
      towerName = name;
      nextTower = link;
   }

/* Method: signal() */
/**
 * This method represents sending a signal to this tower. The effect
 * is to light the signal fire here and to send an additional signal
 * message to the next tower in the chain, if any.
 */
   public void signal() {
      lightCurrentTower();
      if (nextTower != null) nextTower.signal();
   }

/* Method: lightCurrentTower() */
/**
 * This method lights the signal fire for this tower. This version
 * supplies a temporary implementation (typically called a "stub")
 * that simply prints the name of the tower to the standard output
 * channel. If you wanted to redesign this class to be part of a
 * graphical application, for example, you could override this
 * method to draw an indication of the signal fire on the display.
 */
   public void lightCurrentTower() {
      System.out.println("Lighting " + towerName);
   }

/* Private instance variables */

   private String towerName;        /* The name of this tower    */
   private SignalTower nextTower;   /* A link to the next tower  */
}
```

The constructor for the `SignalTower` class is simple and does nothing more than copy its arguments into the corresponding instance variables. The code looks like this:

```
public SignalTower(String name, SignalTower link) {
    towerName = name;
    nextTower = link;
}
```

Because each invocation of the constructor for `SignalTower` requires you to specify the link to the next tower in the chain, it is easiest to create the data structure representing the chain of beacons in Gondor if you create it starting at the end of the chain with Rohan and working backwards toward the front of the chain at Minas Tirith. If each of the towers is declared as an instance variable, the following method initializes each of those variables to reflect the structure of the signal tower chain:

```
private void createSignalTowers() {
    rohan = new SignalTower("Rohan", null);
    halifirien = new SignalTower("Halifirien", rohan);
    calenhad = new SignalTower("Calenhad", halifirien);
    minRimmon = new SignalTower("Min-Rimmon", calenhad);
    erelas = new SignalTower("Erelas", minRimmon);
    nardol = new SignalTower("Nardol", erelas);
    eilenach = new SignalTower("Eilenach", nardol);
    amonDin = new SignalTower("Amon Din", eilenach);
    minasTirith = new SignalTower("Minas Tirith", amonDin);
}
```

The internal representation of linked structures

After you call `createSignalTowers` to initialize the towers, the internal representation for the objects in the heap would look something like the diagram in Figure 7-4. Note that the order in which the signal towers appear is indicated only by the chain of references in

FIGURE 7-4   Memory layout for the objects in the SignalTower chain
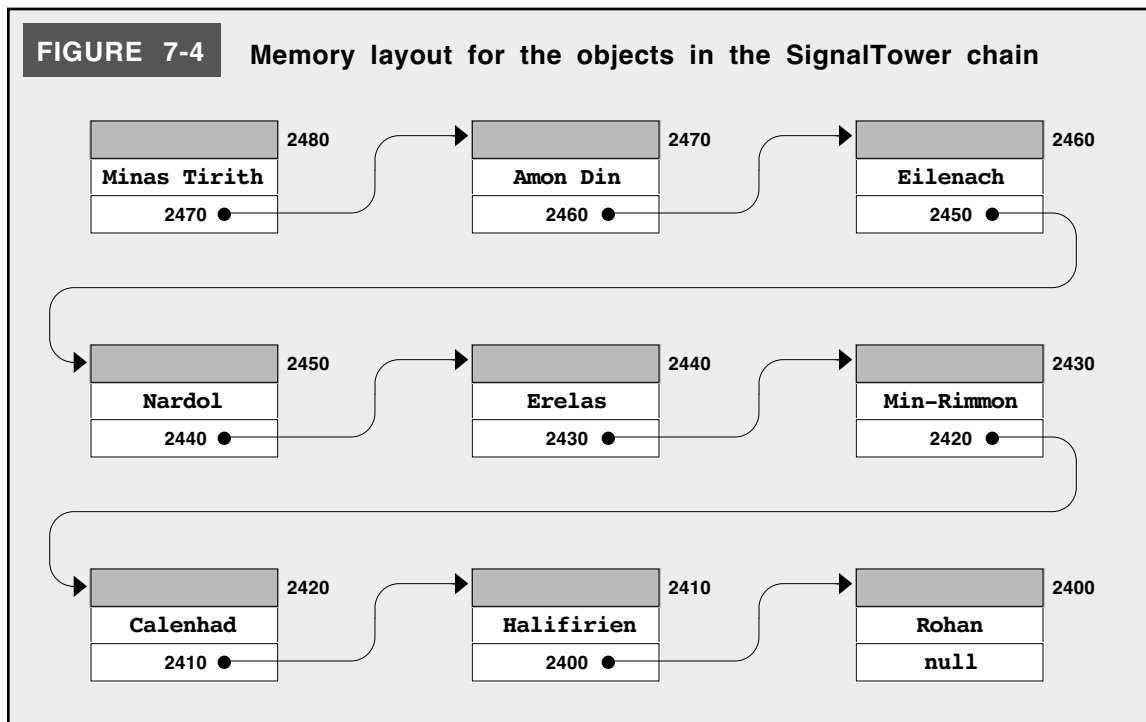
the final memory cells within each structure and not by the order in which the cells appear in memory. To emphasize the internal connections within the chain, the diagram in Figure 7-4 includes arrows that link each cell to its successor. The only value stored in memory is the numeric address, which is sufficient to find the next object in the chain.

Once you have established the data structure shown in Figure 7-4, all you need to do to get the process started is send a `signal` message to the first tower by invoking

```
minasTirith.signal();
```

The implementation of the `signal` method takes whatever action is necessary to simulate lighting the beacon in Minas Tirith and then passes the `signal` message to the object representing Amon Dîn. That object in turn passes the `signal` message to its successor, and the process continues until it reaches the Rohan tower, at which point the `null` in the `nextTower` field indicates that the signal has reached the end of the line.

Summary

Even though the object model is designed to promote a more abstract view of data, using objects effectively requires you to have a mental model of how objects are represented in memory. In this chapter, you have had a chance to see how those objects are stored and to get a sense of what goes on "under the hood" as you create and use objects in your programs.

The important points introduced in this chapter include:

- The fundamental unit of information in a modern computer is a *bit,* which can be in one of two possible states. The state of a bit is usually represented in memory diagrams using the binary digits 0 and 1, but it is equally appropriate to think of these values as *off* and *on* or *false* and *true,* depending on the application.

- Sequences of bits are combined inside the hardware to form larger structures, including *bytes,* which are eight bits long, and *words,* which usually contain four bytes or 32 bits.

- The internal memory of a computer is arranged into a sequence of bytes in which each byte is identified by its index position in that sequence, which is called its *address*.

- Computer scientists tend to write address values and the contents of memory locations in *hexadecimal* notation (base 16) because doing so makes it easy to identify the individual bits.

- The primitive types in Java require different amounts of memory. A value of type `char` requires two bytes, a value of type `int` requires four, and a value of type `double` requires eight. The address of a multibyte value is the address of the first byte it contains.

- Data values that you create in a Java program are allocated in three different regions of memory. Static variables and constants that apply to an entire class are allocated in a region of memory devoted to the program code and static data. Local variables are allocated in a region called the *stack,* which is apportioned into structures called *frames* that contain all of the local variables for a method. All objects are allocated in a region called the *heap,* which is simply a pool of available memory. In most systems, the stack and the heap start at opposite ends of memory and grow toward each other to make available as much memory as possible.

- When you declare a local variable of some object class and assign it an initial value by calling its constructor, memory is allocated in both the heap and the stack. The heap

contains the storage for the object data itself. The stack entry for the variable contains only enough memory to hold the address of the object, which is called a *reference*.

- The stack frame for a method call includes an entry identified by the keyword `this` that identifies the *receiver,* which is the object on which that method is applied.

- As a computation proceeds, objects on the heap are often used as temporary values that are no longer needed at the end of the computation. Such values, however, continue to occupy memory in the heap and are referred to as *garbage*. The Java runtime system periodically undertakes a process called *garbage collection* to search through the heap and reclaim objects that are no longer needed.

- When an object is passed from one method to another, only the reference to that object is copied into the stack frame for the new method. Because this reference identifies exactly the same object as the identical reference in the caller, object values are effectively shared between the calling and called methods.

- The primitive types in Java are not in fact objects and therefore cannot be used in contexts in which an object is required. To get around this problem, Java defines a set of *wrapper classes* that encapsulate each of the primitive types in a full-fledged object.

- Objects in Java can contain references to other objects. Such objects are said to be *linked*. Linked structures are used quite often in programming and will be covered in more detail in later chapters in this book.

Review questions

1. Define the following terms: *bit, byte,* and *word*.

2. What is the etymology of the word *bit?*

3. How many bytes of memory are there in a 384MB machine?

4. Convert each of the following decimal numbers to its hexadecimal equivalent:

   a) 17
   b) 256
   c) 1729
   d) 2766

5. Convert each of the following hexadecimal numbers to decimal:

   a) **17**
   b) **64**
   c) **CC**
   d) **FAD**

6. What is an *address?*

7. How many bytes does Java assign to a value of type `int`? How many bytes are required for a `double`?

8. What are the three memory regions in which values can be stored in a Java program?

9. Using the example in section 7.2 as a model, trace the heap and stack operations that occur in the execution of the following method:

```
public void run() {
   Rational x = new Rational(4, 5);
   Rational y = new Rational(5, 2);
   Rational z = x.multiply(y).subtract(z);
   println(x + " x " + y + " - " + y + " = " + z);
}
```

Which objects on the heap are garbage when the **println** statement is reached.

10. True or false: When you pass a primitive data value from one method to another, Java always copies that value into the frame of the method being called.

11. True or false: When you pass an object from one method to another, Java copies the data in that object into the new frame.

12. Describe the two phases in a simple mark-and-sweep garbage collector.

13. What is meant by the term *wrapper class?* What purpose do wrapper classes serve?

14. What methods can you use to convert between integers and the string representations of those integers in a particular base?

15. What property identifies a *linked structure?*

16. Given that objects of a particular class require a certain amount of space in memory, it is clear that an object in the heap could never physically contain another object of that same class and still have room for additional data. What can a Java programmer do to achieve the same effect?

17. What is a *stub?*

18. Why is important for Java to include the special value **null?** What does this value represent?

Programming exercises

1. Use the static methods **Integer.parseInt** and **Integer.toString** to write a program that converts hexadecimal values into their decimal equivalents. Your program should continue to read hexadecimal values until the user enters a 0. A sample run of this program might look like this:

```
○ ○ ○               HexToDecimalConverter
This program converts hexadecimal to decimal.
Enter 0 to stop.
Enter a hexadecimal number: 10
10 hex = 16 decimal
Enter a hexadecimal number: 2A
2A hex = 42 decimal
Enter a hexadecimal number: FFFF
FFFF hex = 65535 decimal
Enter a hexadecimal number: 0
```
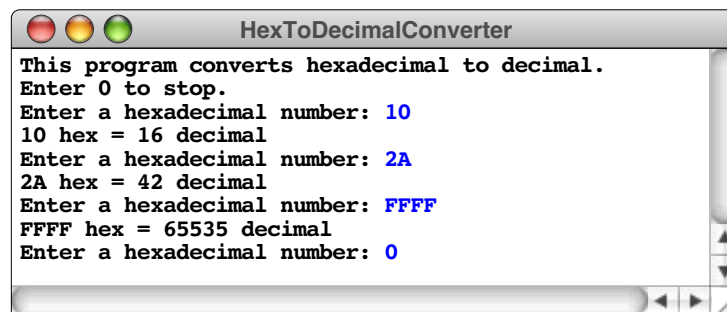
2. The fact that the **Integer.parseInt** method makes it possible for a program to read user input as a string and then convert it to an integer makes it possible to write programs that use something other than an integer—such as a blank line—as a

sentinel to signal the end of the input. Rewrite the **AverageList** program from exercise 4-6 so that it uses a blank line to mark the end of the input.

3.   But don't panic. Base 8 is just like base 10 really—if you're missing two fingers.

—Tom Lehrer, "The New Math," 1965

Rewrite the Math Quiz program from exercise 6-5 so that it poses its questions in base 8 instead of base 10, as shown in the following sample run:

```
●●●                        MathQuiz
Welcome to the octal Math Quiz
What is 3 + 5 in base 8? 10
That's the answer!
What is 17 – 15 in base 8? 2
That's the answer!
What is 20 – 16 in base 8? 4
That's incorrect – try a different answer: 2
That's the answer!
What is 6 + 7 in base 8? 13
That's incorrect – try a different answer: 15
That's the answer!
What is 21 – 6 in base 8? 13
That's the answer!
```
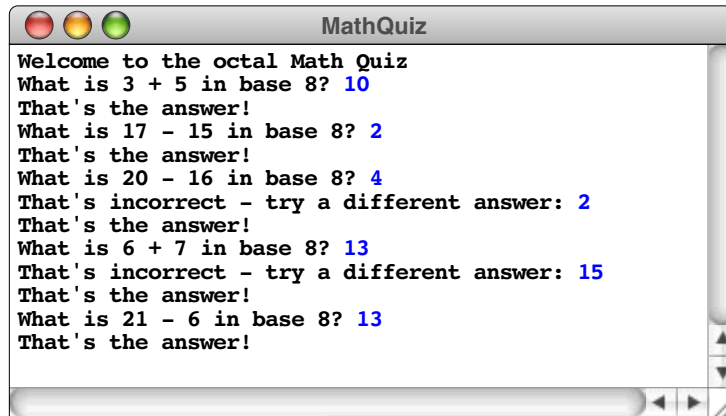
4.   The **Runtime** class in the **java.lang** package includes a few simple methods that may help you get a better sense of what Java's garbage collector does. A **Runtime** object maintains information about the state of the Java Virtual Machine. If you want to look at that information, you can get the current runtime environment by calling the static method **getRuntime()** and storing the result in a variable like this:

```
Runtime myRuntime = Runtime.getRuntime();
```

Once you have this variable, you can find out how much free memory is available by calling

```
myRuntime.freeMemory();
```

Because memory sizes can be large, the value returned by **freeMemory** is a **long** rather than an **int** and indicates the number of bytes available. You can also explicitly trigger the garbage collector by calling

```
myRuntime.gc();
```

Write a program that allocates 10000 **Rational** objects without saving any of them in variables so that they all become garbage. Once you've done so, measure the amount of free memory before and after garbage collection and use the difference to report how many bytes were freed, as shown in the following sample run:

```
●●●                        GCTest
Allocating 10000 Rational objects
Garbage collection freed 94140 bytes
```
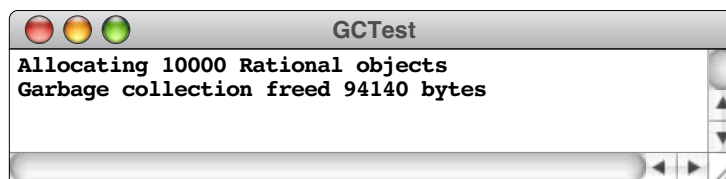
Chapter 8
Object-Oriented Graphics

Art, it seems to me, should simplify. That, indeed, is very nearly the whole of the higher artistic process; finding what conventions of form and what details one can do without and yet preserve the spirit of the whole.

— Willa Cather, *On the Art of Fiction*, 1920

Brenda Laurel

Brenda Laurel directs the graduate program in Media Design at the Art Center College of Design in Pasadena, California and has long been one of the leading researchers in the field of Human-Computer Interaction (HCI). In the mid-1990s, Dr. Laurel was part of the founding team at the Interval Research Corporation, a visionary start-up company focusing on making it easier for individual users to interact with computers. In 1997, she co-founded Purple Moon Software, which pioneered the development of interactive applications for girls. Dr. Laurel described her experience with that project in a book entitled *Utopian Entrepreneur* (MIT Press, 2001).

Although you have already seen several of the classes and methods in the **acm.graphics** package, you have not yet had the chance to consider that collection of classes as a whole. Rather than continuing to scratch the surface, this chapter examines the entire package as an integrated set of tools. Doing so has two main purposes. First, you will learn a lot more about the graphical capabilities that the package provides. That understanding, in turn, will help you to write more interesting graphics programs, which ought to be an exciting prospect. But there is a subtler purpose as well. The object-oriented approach to programming has less to do with styles of coding than it does with strategies of design. By focusing on the **acm.graphics** package, you can get a sense of what considerations go into the design of a package. By studying existing packages, you will gain a more detailed sense of how to approach the design of your own classes and packages that will make it easier to write them on your own.

8.1 The acm.graphics model

Before you can appreciate the classes and methods available in **acm.graphics**, you need to understand the assumptions, conventions, and metaphors on which the package is based. Collectively, these ideas that define the appropriate mental picture for a package are called a **model.** The model for a package allows you to answer various questions about how you should think about working in that domain. Before using a graphics package, for example, you need to be able to answer the following sorts of questions:

- What are the real-world analogies and metaphors that underlie the package design?
- How do you specify positions on the screen?
- What units do you use to specify lengths?

The collage metaphor

In the case of the **acm.graphics**, the first question—what analogies and metaphors are appropriate for the package—is perhaps the most important to resolve. There are many real-world analogies for computer graphics, because there are many different ways to create visual art. One possible metaphor is that of painting, in which the artist selects a paintbrush and a color and then draws images by moving the brush across a screen that represents a virtual canvas. If you instead imagined yourself drawing with a pencil, you might develop a different style of graphics in which line drawings predominated and in which it was possible to erase things you had previously drawn. You might conceivably design a graphics package around an engraving metaphor in which the artist uses a stylus to etch out a drawing; erasing would presumably not make sense in that metaphor. There are many other possible metaphors, just as there are many styles of art.

To support the notion of object-oriented design, the **acm.graphics** package uses the metaphor of a **collage.** A collage artist works by taking various objects and assembling them on a background canvas. Those objects might be geometrical shapes, words clipped from newspapers, lines formed from bits of string, or images taken from magazines. The **acm.graphics** package offers counterparts for all of these. The notion that the image is a collage has important implications for the way you describe the process of creating a design. If you are painting, you might talk about making a brush stroke in a particular position or filling an area with paint. In the collage model, the key operations are adding and removing objects, along with repositioning them on the background. Collages also have the property that some objects can be positioned on top of other objects, where they obscure whatever is behind them. Removing those objects reveals whatever used to be underneath.

The coordinate system

You already know the answers to the questions about specifying positions on the screen and defining units for length from the discussion of the various **GraphicsProgram** examples presented in the preceding chapters. The fundamental rule is that the **acm.graphics** package uses precisely the coordinate model that traditional Java programs do, so that you will encounter less of a cognitive shift when you learn the more complex Java model later in this text. In the Java model, all lengths are expressed in terms of **pixels,** which are the individual dots that appear on the screen. Points on the screen are identified by specifying their coordinates in both the *x* and *y* directions. Those coordinates are also measured in pixels, with *x* values increasing as you move rightward across the canvas and *y* values increasing as you move down from the top. The interpretation of the *x* coordinate is therefore the same as in traditional Cartesian geometry, but the *y* coordinate is inverted. That inversion follows from the fact that Java uses a different starting point, or **origin,** for its coordinate system. Instead of placing the origin in the lower left corner as in the Cartesian world, Java puts it in the upper left corner. This reversal of the coordinate system will probably cause some confusion at first, but it doesn't take too long to become familiar with the new model.

8.2 The graphics class hierarchy

Once you understand the coordinate system, the next question you might reasonably ask is what sorts of objects are displayable in the collage. Again, you have some intuition for that answer from your holistic explorations of the **acm.graphics** package in the earlier chapters. In fact, you were introduced to most of the important classes in the package back in Chapter 2, in which Figure 2-7 offered a diagram of the shape classes in the **acm.graphics** package as an illustration of class hierarchies. That figure is reproduced as Figure 8-1, which shows the shape classes in the hierarchy along with two other classes—**GCanvas** and **GCompound**—that will be important in understanding the detailed operation of the package.

FIGURE 8-1 Important classes in the acm.graphics package
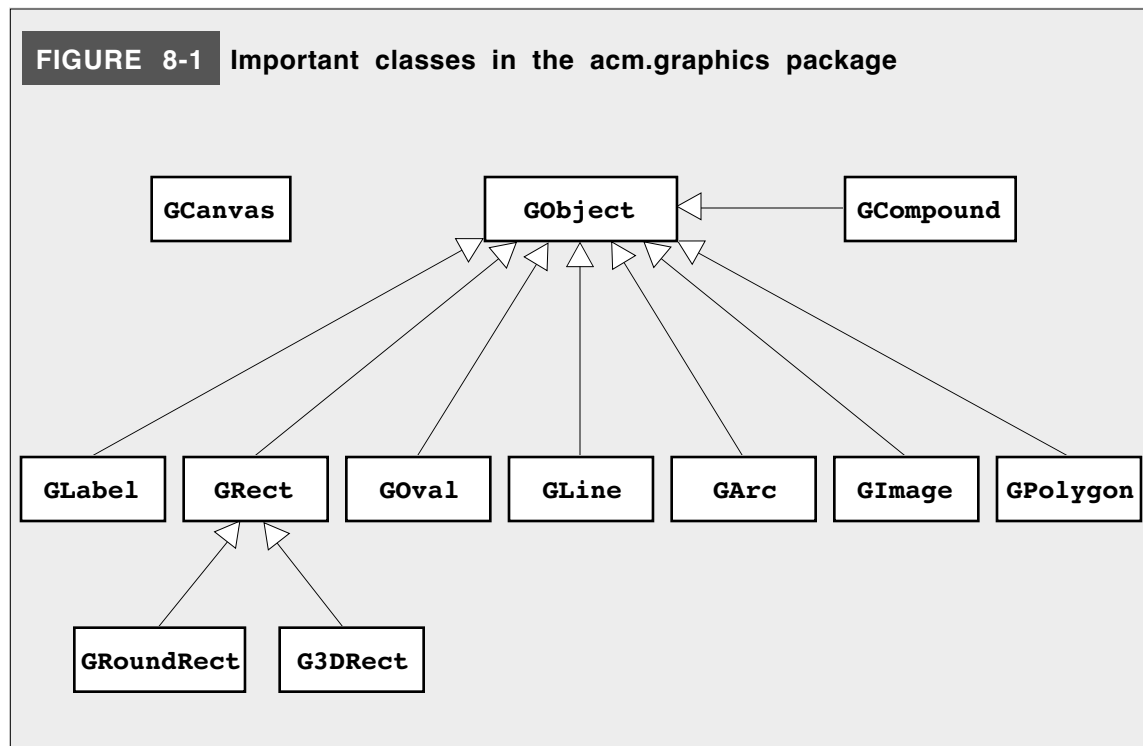
The `GObject` class

As the arrows in the Figure 8-1 emphasize, the central class in the `acm.graphics` package is `GObject`, which represents the universe of all graphical objects. You may have noticed that Figure 8-1 specifies the name of this class in italic type. In class diagrams, italics are used to identify **abstract classes,** which are classes that have no objects of their own but serve as a common template for a variety of concrete subclasses. Thus, there are no objects that have `GObject` as their primary class. Instead, every object you put on the screen is a `GLabel`, `GRect`, `GOval`, or one of the other subclasses that extends `GObject`. The rules of class formation, however, dictate that any of those objects is also a `GObject`, but you will never see a declaration of the form

```
GObject gobj = new GObject();
```

because you cannot construct a `GObject` itself. At the same time, it is perfectly legal to construct an instance of a specific `GObject` subclass and then assign it to a `GObject` variable, as in

```
GObject gobj = new GLabel("hello, world");
```

The label is a `GObject`, so the assignment works.

The `GObject` class itself defines the set of methods shown in Figure 8-2. As it happens, you've already seen most of these. Here are a few of the new ones:

- The `movePolar(r, theta)` method allows you to move an object using **polar coordinates,** which are expressed as a distance (`r`) and an angle (`theta`). The angle is measured in degrees counterclockwise from the +*x* axis, just as it is in classical geometry. Thus, the call

```
gobj.movePolar(100, 45);
```

moves the object stored in `gobj` 100 pixels along a line in the 45° direction, which is northeast.
- The `contains(x, y)` method allows you to determine whether an object contains a particular point. This is a predicate method and therefore returns `true` or `false`.
- The `setVisible()` method makes it possible to hide an object on the screen. If you call `setVisible(false)`, the object disappears until you call `setVisible(true)`. The predicate method `isVisible()` allows you to determine whether an object is visible.
- The various `send` methods allow you to change the layering arrangement in the collage. When you add a new object, it goes on top of the other objects and can therefore obscure the objects behind it. If you call `sendToBack()`, the object goes to the very back of the list. Conversely, `sendToFront()` brings it to the foreground. The `sendForward()` and `sendBackward()` methods move an object one step forward or backward in the stack. Because the layering axis corresponds to a third dimension at right angles to the *x-y* grid, this dimension is typically called the *z* **axis.**
- The mouse-handling methods at the bottom of Figure 8-2 make it possible for objects to respond to mouse events. Those capabilities are described in section 8.4.

FIGURE 8-2   Methods supported by all GObject subclasses
`void setLocation(double x, double y)`   Sets the location of this object to the specified point.
`void move(double dx, double dy)`   Moves the object using the displacements `dx` and `dy`.
`void movePolar(double r, double theta)`   Moves the object `r` units in direction `theta`, measured in degrees.
`double getX()`   Returns the x-coordinate of the object.
`double getY()`   Returns the y-coordinate of the object.
`double getWidth()`   Returns the width of the object.
`double getHeight()`   Returns the height of the object.
`boolean contains(double x, double y)`   Checks to see whether a point is inside the object.
`void setColor(Color c)`   Sets the color of the object.
`Color getColor()`   Returns the object color.
`void setVisible(boolean visible)`   Sets whether this object is visible.
`boolean isVisible()`   Returns `true` if this object is visible.
`void sendToFront()`   Sends this object to the front of the canvas, where it may obscure objects further back.
`void sendToBack()`   Sends this object to the back of the canvas, where it may be obscured by objects in front.
`void sendForward()`   Sends this object forward one position in the *z* ordering.
`void sendBackward()`   Sends this object backward one position in the *z* ordering.
`void addMouseListener(MouseListener listener)`   Specifies a listener to process mouse events for this graphical object.
`void removeMouseListener(MouseListener listener)`   Removes the specified mouse listener from this graphical object.
`void addMouseMotionListener(MouseMotionListener listener)`   Specifies a listener to process mouse motion events for this graphical object.
`void removeMouseMotionListener(MouseMotionListener listener)`   Removes the specified mouse motion listener from this graphical object.

Although it is nice to know about these new methods, it may be momentarily disturbing to discover that some of the old methods you've been using don't appear in Figure 8-2. There is no **setFilled** method, even though you've been filling **GOval**s and **GRect**s all along. Similarly, there is no **setFont** method, but you've been using that in the context of **GLabel**s. As the caption on the figure makes clear, the methods defined at the **GObject** level are the ones that apply to all the subclasses, and not just some of them. The **setFont** method, for example, only makes sense for the **GLabel** class and is therefore defined there. The **setFilled** method is more interesting. It makes sense to fill an oval, a rectangle, a polygon, and an arc, so there are four classes (plus the extended **GRoundRect** and **G3DRect** subclasses) for which **setFilled** would be appropriate. It is not at all clear, however, what one might mean by filling a line, an image, or a label. Since there are classes that cannot give a meaningful interpretation to **setFilled**, it is not

defined at the **GObject** level. At the same time, it doesn't seem ideal to define **setFilled** independently in each of the subclasses for which it is defined. You certainly would like **setFilled** to work the same way for each of the fillable classes, for consistency if nothing else.

In Java, the best way to define a suite of methods that are shared by some but not all subclasses in a particular hierarchy is to define what Java calls an **interface,** which is really just a listing of the method headers that all classes implementing that interface share. In the **acm.graphics** package, the classes **GOval**, **GRect**, **GPolygon**, and **GArc** all implement an interface called **GFillable**, which specifies the behavior of any fillable object. In addition to **GFillable**, there is an interface called **GResizable** that allows you to reset the bounds of an object and a somewhat broader interface called **GScalable** that allows you to scale an object by a scaling factor.

The methods specified by these three interfaces appear in Figure 8-3. The most interesting new method in that list is **setFillColor(c)**, which makes it possible to give the interior of a fillable shape a different color than its boundary. For example, if you execute the code

```
GRect r = new GRect(70, 20);
r.setColor(Color.RED);
r.setFillColor(Color.GREEN);
r.setFilled(true);
```

you get a filled rectangle whose interior is green and whose border is red.

The **GLabel** class

The **GLabel** class is the first class you encountered in this text, even though it is in some respects the most idiosyncratic entry in the hierarchy of shape classes. The most important difference between **GLabel** and everything else is that the position of a **GLabel** is not defined by the upper left corner, but rather by the starting point of the **baseline,**

FIGURE 8-3   **Methods specified by interfaces**

GFillable (implemented by **GArc**, **GOval**, **GPolygon**, and **GRect**)

void setFilled(boolean fill)   Sets whether this object is filled (**true** means filled, **false** means outlined).
boolean isFilled()   Returns **true** if the object is filled.
void setFillColor(Color c)   Sets the color used to fill this object. If the color is **null**, filling uses the color of the object.
Color getFillColor()   Returns the color used to fill this object.

GResizable (implemented by **GImage**, **GOval**, and **GRect**)

void setSize(double width, double height)   Changes the size of this object to the specified width and height.
void setBounds(double x, double y, double width, double height)   Changes the bounds of this object as specified by the individual parameters.

GScalable (implemented by **GArc**, **GCompound**, **GImage**, **GLine**, **GOval**, **GPolygon**, and **GRect**)

void scale(double sf)   Resizes the object by applying the scale factor in each dimension, leaving the location fixed.
void scale(double sx, double sy)   Scales the object independently in the *x* and *y* dimensions by the specified scale factors.

which is an imaginary line on which characters sit. The origin and baseline properties of the **GLabel** class are illustrated in the following diagram:

The quick brown fox jumped over the lazy dog ....*baseline*

origin

To set the size, style, and general appearance of a **GLabel**, you need to specify its font, using the **setFont** method as described in Chapter 5. The structure of this method and the other methods specific to the **GLabel** class appear in Figure 8-4.

The **GRect** class and its subclasses (**GRoundRect** and **G3DRect**)

If **GLabel** is the most idiosyncratic class in the **acm.graphics** package, **GRect** is the most conventional. It implements all three of the special interfaces—**GFillable**, **GResizable**, and **GScalable**—but otherwise defines nothing that is not in **GObject**.

The two specialized forms of rectangles, **GRoundRect** and **G3DRect**, differ only in their appearance on the display. The **GRoundRect** class has rounded corners, and the **G3DRect** class supports a raised appearance. Both of these display styles are included in the standard Java graphics package, and are therefore included in **acm.graphics** as well.

The **GOval** class

The **GOval** class is almost as straightforward as the **GRect** and operates in much the same way. The only thing that causes confusion about **GOval** is that its origin is the upper left corner and not the center. This convention makes sense when you see that ovals are specified in terms of their bounding rectangle, but can seem somewhat unnatural, particularly when the oval is a circle. Circles in mathematics are conventionally defined in terms of their center and their radius. In Java, you need to think of them in terms of their upper left corner and their diameter.

The **GLine** class

The **GLine** class makes it possible to construct line drawings in the **acm.graphics** package. The **GLine** class implements **GScalable** (which is performed relative to the starting point of the line), but not **GFillable** or **GResizable**. It is, moreover, important to modify the notion of containment for lines, since the idea of being within the boundary of the line is not well defined. In the abstract, of course, a line is infinitely thin and therefore contains no points in its interior. In practice, however, it makes sense to define

---

FIGURE 8-4   **Additional methods in the GLabel class**

void setFont(Font f) *or* **setFont(String description)**   Sets the font to a Java **Font** object or a string in the form **"Family-style-size"**
Font getFont()   Returns the current font.
double getAscent()   Returns the distance the characters in the current font extend above the baseline.
double getDescent()   Returns the distance the characters in the current font extend below the baseline.
void setLabel(String str)   Changes the string used to display the label.
String getLabel()   Returns the string that this label displays.

a point as being contained within a line if it is "close enough" to be considered as part of that line. In the **acm.graphics** package, that distance is specified by the constant **LINE_TOLERANCE** in the **GLine** class, which is defined to be a pixel and a half.

The **GLine** class also contains several methods for determining and changing the endpoints of the line. The **setStartPoint** method allows clients to change the first endpoint of the line without changing the second; conversely, **setEndPoint** gives clients access to the second endpoint without affecting the first. These methods are therefore different in their operation from **setLocation**, which moves the entire line without changing its length and orientation.

The **GArc** class

Arcs are implemented in many different ways in different graphics systems. The model used in **acm.graphics** is chosen—as always—to be consistent with the standard Java interpretation of arcs. The bad news is that this interpretation takes some getting used to. The good news is that you won't have to use **GArc**s very often.

In Java, an arc is defined in more or less the same way that an oval is. The usual form for calling the constructor is

> **new GArc(***width***,** *height***,** *start***,** *sweep***)**

The *width* and *height* parameters are exactly the same as those for a **GOval** and specify the dimensions of the bounding rectangle. The *start* parameter specifies the angle at which the arc starts, and *sweep* parameter indicates the number of degrees through which it extends. As is true throughout Java, angles are measured in degrees counterclockwise from the +*x* axis. The effect of these parameters is illustrated by the following diagrams, which show where various arcs appear inside their bounding rectangle, which is indicated here by the dotted line but would not appear on the screen:

new GArc(60, 60, 0, 45)   new GArc(60, 60, 90, 180)   new GArc(60, 60, -90, 270)
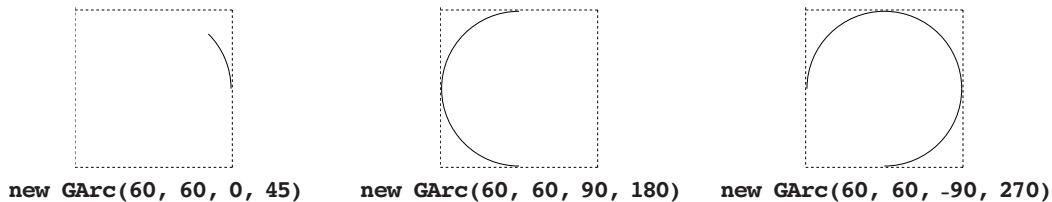
Java defines filling for arcs in a way that at first seems a little odd. If you create an unfilled **GArc**, only the arc is shown. If you then fill it by calling **setFilled(true)**, Java fills the wedge that extends to the center of the circle as shown in the diagram on the right. Adopting the standard Java interpretation of arc filling has implications for the design. Most notably, the **contains** method for the **GArc** class returns a result that depends on whether the arc is filled. For an unfilled arc, containment implies that the arc point is actually on the arc, subject to the same interpretation of "closeness" as described for lines in the preceding section. For a filled arc, containment implies inclusion in the wedge.

Unfilled arc

The **GArc** class includes methods that enable clients to manipulate the angles defining the arc (**setStartAngle**, **getStartAngle**, **setSweepAngle**, and **getSweepAngle**) as well as methods to return the points at the beginning and end of the arc (**getStartPoint** and **getEndPoint**).

Filled arc
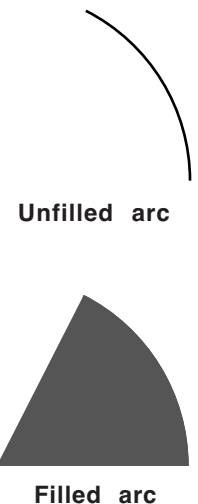
The `GImage` class

The `GImage` class is used to display an image stored in a recognized format for image data such as GIF (Graphics Interchange Format) or JPEG (Joint Photographic Experts Group). The way you use it is simply to put an image file in your workspace and then call

```
new GImage("filename")
```

where *filename* is the name of that file. This creates for you a `GImage` object that you can position on the screen just like any other `GObject`. A `GImage` object is both `GResizable` and `GScalable`, but not `GFillable`. Resizing or scaling an image has the effect of stretching or compressing the pixels in the image and is implemented by the standard graphics tools in Java.

The `GPolygon` class

The `GPolygon` class is the most complex of the shape classes, but it is not that hard to use. The primary difference in the design is that the constructor for a `GPolygon` does not create the whole thing. Rather, what you get is an empty `GPolygon` to which you can then add vertices or edges using the methods `addVertex`, `addEdge`, and `addPolarEdge`.

These methods are easiest to illustrate by example. The simplest method to explain is `addVertex(x, y)`, which adds a vertex at the point (`x`, `y`) relative to the location of the polygon. For example, the following code defines a diamond-shaped polygon in terms of its vertices, as shown in the diagram at the right:

```
GPolygon diamond = new GPolygon();
diamond.addVertex(-22, 0);
diamond.addVertex(0, 36);
diamond.addVertex(22, 0);
diamond.addVertex(0, -36);
```
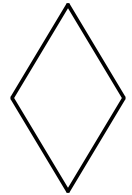
The diamond is drawn so that its center is at the point $(0, 0)$ in the coordinate space of the polygon. Thus, if you were to add `diamond` to a `GCanvas` and set its location to the point $(300, 200)$, the diamond would be centered at that location.

The `addEdge(dx, dy)` method is similar to `addVertex`, except that the parameters specify the displacement from the previous vertex to the current one. One could therefore draw the same diamond by making the following sequence of calls:

```
GPolygon diamond = new GPolygon();
diamond.addVertex(-22, 0);
diamond.addEdge(22, 36);
diamond.addEdge(22, -36);
diamond.addEdge(-22, -36);
diamond.addEdge(-22, 36);
```

Note that the first vertex must still be added using `addVertex`, but that subsequent ones can be defined by specifying the edge displacements. Moreover, the final edge is not explicitly necessary because the polygon is automatically closed before it is drawn.

Some polygons are easier to define by specifying vertices; others are more easily represented by edges. For many polygonal figures, however, it is even more convenient to express edges in polar coordinates. This mode of specification is supported in the `GPolygon` class by the method `addPolarEdge`, which is identical to `addEdge` except that its arguments are the length of the edge and its direction
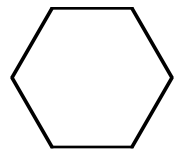
expressed in degrees counterclockwise from the +x axis. This method makes it easy to create figures with more sophisticated structure, such as the centered hexagon generated by the following method (as shown on the right using 36 pixels as the value of **side**):

```
GPolygon createHexagon(double side) {
   GPolygon hex = new GPolygon();
   hex.addVertex(-side, 0);
   int angle = 60;
   for (int i = 0; i < 6; i++) {
      hex.addPolarEdge(side, angle);
      angle -= 60;
   }
   return hex;
}
```

   The **GPolygon** class implements the **GFillable** and **GScalable** interfaces, but not **GResizable**. It also supports the method **rotate(theta)**, which rotates the polygon theta degrees counterclockwise around its origin.

8.3 Facilities available in the **GraphicsProgram** class

The preceding sections described the various shape classes available in the **GObject** hierarchy but did not discuss what you can do with those objects once you have them. Internally, graphical objects are added to a graphical canvas, which is implemented as the **GCanvas** class. Most of the time, however, you will not need to use **GCanvas** explicitly because the **GraphicsProgram** class you have already seen creates a **GCanvas** and fills the program window with it. It then intercepts all the messages you might send to a **GCanvas** and forwards them along. Those methods are listed in Figure 8-5.

   Most of these methods are completely straightforward. The only one that deserves special mention is the method

```
GObject getElementAt(double x, double y)
```

which returns the object, if any, at the coordinates (**x**, **y**). If there is more than one such

---

FIGURE 8-5 **Methods supported by the GraphicsProgram class (and by GCanvas)**

void add(GObject gobj)
   Adds a graphical object to the canvas at its internally stored location.

void add(GObject gobj, double x, double y)
   Adds a graphical object to the canvas and sets its location as indicated.

void remove(GObject gobj)
   Removes the specified graphical object from the canvas.

void removeAll()
   Removes all graphical objects from the canvas.

int getWidth()
   Returns the width of the canvas in pixels.

int getHeight()
   Returns the height of the canvas in pixels.

int getElementCount()
   Returns the number of graphical objects contained on the canvas.

GObject getElement(int i)
   Returns the graphical object at the specified index, numbering from back to front.

GObject getElementAt(double x, double y)
   Returns the frontmost object containing the specified point, or **null** if no such object exists.

object, **getElementAt** returns the one that is in front according to the *z* ordering established by the **GObject** methods like **sendToBack** and **sendToFront**. But what if there is no object there? In that case, **getElementAt** returns the special constant **null**, which is a reserved word in Java that indicates that there is in fact no object corresponding to that value.

The **getWidth** and **getHeight** methods make it possible to determine the size of the graphics canvas. For example, the constructor

```
new GLine(0, getHeight(), getWidth(), 0)
```

creates a **GLine** object that extends from the lower left corner of the canvas to the upper right corner. These methods also make it possible to center graphical objects in the window because the *x*- and *y*-coordinates of the center of the window can be expressed as **getWidth() / 2** and **getHeight() / 2**, respectively.

8.4 Animation and interactivity

So far, the graphical programs you have seen in this text produce a fixed picture on the screen that does not change after you have put it together. Although the facilities in the graphics library allow you to create wonderful pictures in this way, the programs that you write won't be nearly as exciting until you learn how to change those pictures as the program runs. You might, for example, want to have particular graphical objects move across the screen or have them change their size and color. In computer graphics, the process of updating a displayed image so that it changes over time is called **animation.**

Time-based animation

The easiest way to animate graphical programs is to write a loop in your program that makes a small change to the image and then suspends the program for a very short time interval. A very simple example of this style of graphics is shown in Figure 8-6, which moves a **GLabel** across the screen from right to left, just the way the headline displays in New York's Times Square do.

The first part of the **run** method is very much like the "hello, world" program from Chapter 2. The first three lines are

```
GLabel label = new GLabel(HEADLINE);
label.setFont("Times-72");
add(label, getWidth(), (getHeight() + label.getAscent()) / 2);
```

which create a new **GLabel**, set its font to something appropriately large for a headline, and then add the label to the window. The coordinates, however, might seem a little surprising at first. The expression for the *y*-coordinate of the label—cumbersome as it seems to be—is actually the idiomatic expression for centering a label vertically on the screen. The baseline of the label needs to be at the vertical center of the window, but shifted further towards the bottom by half the vertical ascent of the characters. It is the expression for the *x*-coordinate that is more surprising, because positioning the label so that its *x*-coordinate is the width of the window means that the entire label is outside the visible part of the window.

If the display stayed that way, the **TimesSquare** program would simply display an empty canvas. The rest of the **run** method, however, has the effect of changing the display so that the label gradually moves across the window. The code looks like this:

FIGURE 8-6   **Program to animate a text label**

```
/*
 * File: TimesSquare.java
 * ---------------------
 * This program displays the text of the string HEADLINE
 * on the screen in an animated way that moves it across
 * the display from left-to-right.
 */

import acm.graphics.*;
import acm.program.*;

public class TimesSquare extends GraphicsProgram {

/** Runs the program */
   public void run() {
      GLabel label = new GLabel(HEADLINE);
      label.setFont("Times-72");
      add(label, getWidth(), (getHeight() + label.getAscent()) / 2);
      while (label.getX() + label.getWidth() > 0) {
         label.move(-DELTA_X, 0);
         pause(PAUSE_TIME);
      }
   }

/** The number of pixels to shift the label on each cycle */
   private static final double DELTA_X = 2.0;

/** The number of milliseconds to pause on each cycle */
   private static final int PAUSE_TIME = 20;

/** The string to use as the value of the label */
   private static final String HEADLINE =
      "When in the course of human events it becomes necessary " +
      "for one people to dissolve the political bands which " +
      "connected them with another . . .";
}
```

```
while (label.getX() + label.getWidth() > 0) {
   label.move(-DELTA_X, 0);
   pause(PAUSE_TIME);
}
```

For the moment, forget about the condition in the while loop and focus on the two statements that make up the body. The first is

```
label.move(-DELTA_X, 0);
```

which adjusts the *x*-coordinate of the label to move it **DELTA_X** pixels to the left, where **DELTA_X** is defined to have the value 2. Thus, on each iteration of the **while** loop, the label moves two pixels leftward. The second line is

```
pause(PAUSE_TIME);
```

which causes the program to suspend its operation for the specified number of milliseconds. Sleeping for 20 milliseconds means that the display will be updated 50 times a second, which is well below the threshold at which the eye perceives motion as continuous.

The call to **pause** in the **TimesSquare** program is necessary to achieve the effect of animation. Computers today run so quickly that the label would instantly zip off the left side of the window if you didn't slow things down enough to bring the operation back to human speed. The fact that the label will eventually—even at this slower speed—move off the left edge makes it easier to understand the condition in the **while** loop. The expression **label.getX() + label.getWidth()** indicates the *x*-coordinate of the rightmost part of the label. As long as that is still greater than 0, some part of the label will be visible on the screen. When the last character shifts off the left edge, the **while** loop stops.

You can change the speed of an animation in either of two ways. First, you can change the distance that the position of each object is updated on each time step. For example, setting **DELTA_X** to 4 would double the apparent speed of the motion, although it would probably also begin to appear more jerky. The second approach is that you can change the delay time. Intuitively, you could double the speed of the display by halving the delay time on each cycle, although that strategy has limits in practice. The delay that a program experiences when it calls **pause** is not guaranteed to be precise. Moreover, there is always some overhead associated with pausing a program, and it is impossible to run the animation with a cycle time that is less than the overhead. Thus, there is always some point at which reducing the value of **PAUSE_TIME** in this example will no longer have any noticeable effect.

The general pattern for programs that animate the display can be expressed in the following pseudocode form:

```
public void run() {
      Initialize the graphical objects in the display.
      while (as long as you want to run the animation) {
            Update the properties of the objects that you wish to animate.
            pause(delay time);
}
```

The **TimesSquare** program from Figure 8-6 fits this paradigm, as does the code in Figure 8-7, which displays a square whose color changes randomly once a second.

Responding to mouse events

In the preceding section, you learned a simple strategy through which you can animate a graphical program so that the display changes with time. If you want to write programs that operate like the computing applications you use every day, you will also need to learn how to make those programs interactive by having them respond to actions taken by the user. In some ways, the **ConsoleProgram** examples you have been writing all along are interactive. Most of these programs display messages on the console and wait for a response, which is certainly one style of interaction. In those examples, however, the user was asked for input only at certain well-defined points in the program's execution history, such as when the program called a console method like **readInt** and waited for a response. This style of interaction is called **synchronous,** because it occurs in sync with the program operation. Modern user interfaces, however, are **asynchronous** in that they allow the user to intercede at any point, typically by using the mouse or the keyboard to trigger a particular action.

FIGURE 8-7   **Program to display a square whose color changes once a second**

```java
/*
 * File: ColorChangingSquare.java
 * ----------------------------
 * This program puts up a square in the center of the window
 * and randomly changes its color every second.
 */

import acm.graphics.*;
import acm.program.*;
import acm.util.*;

public class ColorChangingSquare extends GraphicsProgram {

/** Runs the program */
   public void run() {
      GRect square = new GRect(SQUARE_SIZE, SQUARE_SIZE);
      square.setFilled(true);
      add(square, (getWidth() - SQUARE_SIZE) / 2,
                  (getHeight() - SQUARE_SIZE) / 2);
      while (true) {
         square.setColor(rgen.nextColor());
         pause(PAUSE_TIME);
      }
   }

/** Size of the square in pixels */
   private static final double SQUARE_SIZE = 100;

/** Pause time in milliseconds */
   private static final double PAUSE_TIME = 1000;

/** Random number generator */
   private RandomGenerator rgen = new RandomGenerator();
}
```

Events that occur asynchronously with respect to the program operation—mouse clicks, key strokes, and the like—are represented using a structure called an **event.** When an event occurs, the response is always the invocation of a method in some object that is waiting to hear about that event. Such an object is called a **listener.** You'll have the opportunity to see several more examples of listeners later in the text, but it is wonderful to be able to use the mouse, even at this early stage in programming.

In Java, objects that listen for user-interface events do so by implementing the methods in a specific listener interface defined in the package **java.awt.event**. This package contains several interfaces that allow clients to respond to mouse clicks, button presses, keystrokes, changes in component sizes, and many more. For the moment, this text will concentrate only on mouse events, which involve two types of listeners. The first such interface is **MouseListener**, which makes it possible to listen for user actions that primarily involve clicking the mouse; the second is **MouseMotionListener**, which is used to track the mouse as it moves. The reason for separating these two types of listeners is that mouse motions generate many more events than click-type actions do. If your application is driven only by mouse clicks and never has to track the mouse itself,

that application will run more efficiently if it does not have to respond to frequent motion events in which it has no interest.

The listener methods that can be called in response to mouse events are outlined in Figure 8-8. The **GraphicsProgram** class declares itself to be both a **MouseListener** and a **MouseMotionListener** by defining implementations for each of these listener methods; those implementations, however, do nothing at all. For example, the standard definition of **mouseClicked** is simply

```
public void mouseClicked(MouseEvent e) {
    /* Empty */
}
```

Thus, unless you take some action to the contrary, a **GraphicsProgram** will simply ignore mouse clicks, along with all the other mouse events. If you, however, want to change the behavior for a particular event, all you need to do is add a new definition for that method. This new definition supersedes the original definition and will be called instead of the empty one. Any methods that you don't override continue to do what they did by default, which was nothing. Thus, you only have to override the methods you need.

Each of the methods listed in Figure 8-8 takes as its argument an object of type **MouseEvent**, which is a class defined as part of Java's standard window system toolkit. Like the listener interfaces themselves, the **MouseEvent** class lives in the package **java.awt.event**, which means that you need to add the **import** statement

```
import java.awt.event.*;
```

to the beginning of any program that uses mouse events.

The **MouseEvent** class includes a rich set of methods for designing sophisticated user interfaces. This text, however, uses only two of those methods. Given a **MouseEvent** stored in a variable named **e**, you can determine the location of the mouse by calling **e.getX()** and **e.getY()**. Being able to detect the location at which a mouse event

---

FIGURE 8-8   **Standard listener methods for responding to mouse events**

MouseListener interface

| **void mousePressed(MouseEvent e)** |
| Called whenever the mouse button is pressed. |
| **void mouseReleased(MouseEvent e)** |
| Called whenever the mouse button is released. |
| **void mouseClicked(MouseEvent e)** |
| Called when the mouse button is "clicked" (pressed and released within a short span of time). |
| **void mouseEntered(MouseEvent e)** |
| Called whenever the mouse enters the canvas. |
| **void mouseExited(MouseEvent e)** |
| Called whenever the mouse exits the canvas. |

MouseMotionListener interface

| **void mouseMoved(MouseEvent e)** |
| Called whenever the mouse is moved with the button up. |
| **void mouseDragged(MouseEvent e)** |
| Called whenever the mouse is moved with the button down. |

occurred enables you to write many interesting mouse-driven applications, as illustrated by the examples in the two subsections that follow.

Dragging objects on the canvas

The first example of using the mouse is a program that puts up two objects on the screen and lets the user drag them around, which appears as Figure 8-9.

FIGURE 8-9   **Program to drag objects on the canvas**

```java
import java.awt.*;
import java.awt.event.*;
import acm.graphics.*;
import acm.program.*;

/** This class displays a mouse-draggable rectangle and oval */
public class DragObjects extends GraphicsProgram {

/** Runs the program */
   public void run() {
      GRect rect = new GRect(100, 100, 150, 100);
      rect.setFilled(true);
      rect.setColor(Color.RED);
      add(rect);
      GOval oval = new GOval(300, 115, 100, 70);
      oval.setFilled(true);
      oval.setColor(Color.GREEN);
      add(oval);
      addMouseListeners();
   }

/** Called on mouse press to record the coordinates of the click */
   public void mousePressed(MouseEvent e) {
      lastX = e.getX();
      lastY = e.getY();
      gobj = getElementAt(lastX, lastY);
   }

/** Called on mouse drag to reposition the object */
   public void mouseDragged(MouseEvent e) {
      if (gobj != null) {
         gobj.move(e.getX() - lastX, e.getY() - lastY);
         lastX = e.getX();
         lastY = e.getY();
      }
   }

/** Called on mouse click to move this object to the front */
   public void mouseClicked(MouseEvent e) {
      if (gobj != null) gobj.sendToFront();
   }

/* Instance variables */
   private GObject gobj;          /* The object being dragged  */
   private double lastX;          /* The last mouse X position */
   private double lastY;          /* The last mouse Y position */
}
```

The first part of the **run** method in Figure 8-9 should be entirely familiar. It simply creates two graphical objects—a red rectangle and a green oval—and then adds them to the canvas. It's the line

```
addMouseListeners();
```

that introduces something new. The **addMouseListeners** method in **GraphicsProgram** hides a modest amount of internal complexity, but has the effect of enabling the program to watch the mouse and respond to its events. As described in the preceding section, you specify the actions to take when those events occur by overriding listener methods in your program. These listener methods are then called automatically when those events occur.

The first listener method defined in Figure 8-9 is **mousePressed**, which is called when the mouse button first goes down. That method looks like this:

```
public void mousePressed(MouseEvent e) {
    lastX = e.getX();
    lastY = e.getY();
    gobj = getElementAt(lastX, lastY);
}
```

The first two statements simply record the *x* and *y* coordinates of the mouse in the variables **lastX** and **lastY**. As you can see from the program, these variables are declared as instance variables for the object and not as local variables of the sort that you have seen in most methods. It turns out that you will need these values later when you try to drag the object. Because local variables disappear when a method returns, you have to hold onto these in the object itself.

The last statement in **mousePressed** checks to see what object on the canvas contains the current mouse position. Here, it is important to recognize that there are two possibilities. First, you could be pressing the mouse button on top of an object, which means that you want to start dragging it. Second, you could be pressing the mouse button somewhere else on the canvas at which there is no object to drag. The **getElementAt** method looks at the specified position and returns the object it finds there. If there is more than one object covering that space, it chooses the one that is in front of the others in the *z*-axis ordering. If there are no objects at that location, **getElementAt** returns the special value **null**, which signifies an object that does not exist. The other methods will check for this value to determine whether there is an object to drag.

The **mouseDragged** method consists of the following code:

```
public void mouseDragged(MouseEvent e) {
    if (gobj != null) {
        gobj.move(e.getX() – lastX, e.getY() – lastY);
        lastX = e.getX();
        lastY = e.getY();
    }
}
```

The **if** statement simply checks to see whether there is an object to drag. If the value of **gobj** is **null**, there is nothing to drag, and the rest of the method is simply skipped. If there is an object, you need to move it by some distance in each direction. That distance does not depend on where the mouse is in an absolute sense but rather in how far it has moved from where you last took stock of its position. Thus, the arguments to the **move**

method are—for both the *x* and *y* components—the location where the mouse is now minus where it used to be. Once you have moved it, you then have to record the mouse coordinates again so that the location will update correctly on the next `mouseDragged` call.

The final listener method specified in Figure 8-9 is `mouseClicked`, which looks like this:

```
public void mouseClicked(MouseEvent e) {
    if (gobj != null) gobj.sendToFront();
}
```

The intent of this method is to allow the user to move an object to the front by clicking on it, thereby bringing it out from under the other objects on the canvas. The code is almost readable as an English sentence

If there is a current object, send it to the front of the canvas display.

The only question you might have is how the variable `gobj`, which holds the current object, is initialized in this case. The answer depends on the fact that the `mouseClicked` event is always generated in conjunction with a `mousePressed` and a `mouseReleased` event, both of which precede the `mouseClicked` event. The `gobj` variable is therefore set by `mousePressed`, just as if you were going to drag it.

A simple drawing program

Most of you have used drawing programs that allow you to draw shapes on a canvas. In a typical drawing program, you create a rectangle by pressing the mouse at one corner and then dragging it to the opposite corner. For example, if the user pressed the mouse at the location in the left diagram and then dragged it to the position in which you see it in the right diagram, the program would create the rectangle shown:
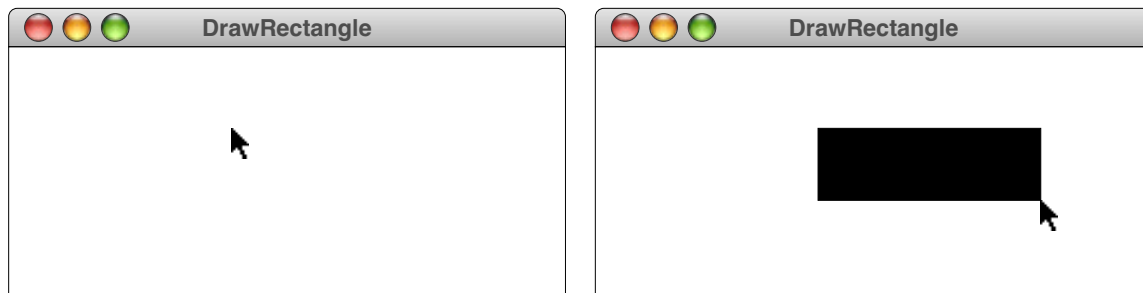
To make it easy for the user to see the shape as it is drawn, drawing programs typically update the coordinates of the rectangle on each call to `mouseDragged`. When the user releases the mouse button, the rectangle is complete and stays where it is. The user can then go back and add more rectangles to the screen by clicking and dragging in the same way.

Figure 8-10 shows a simple `GraphicsProgram` that allows the user to create rectangles by clicking and dragging as illustrated in the example. The only part of the program that might not be immediately obvious is in the calculation of the coordinates for the rectangle in the `mouseDragged` method. The example above illustrates only one possible dragging operation, in which the initial mouse click is in the upper left corner of the `GRect`. The program, however, has to work just as well if the user drags the mouse in some other direction besides to the right and down. For example, it should also be possible to draw a

FIGURE 8-10 **Program to draw rectangles on the canvas**

```java
/*
 * File: DrawRectangle.java
 * -----------------------
 * This program allows users to create rectangles on the canvas
 * by clicking and dragging with the mouse.
 */

import java.awt.event.*;
import acm.graphics.*;
import acm.program.*;

/** This class allows users to drag rectangles on the canvas */
public class DrawRectangle extends GraphicsProgram {

/** Runs the program */
   public void run() {
      addMouseListeners();
   }

/** Called on mouse press to record the starting coordinates */
   public void mousePressed(MouseEvent e) {
      startX = e.getX();
      startY = e.getY();
      currentRect = new GRect(startX, startY, 0, 0);
      currentRect.setFilled(true);
      add(currentRect);
   }

/** Called on mouse drag to reshape the current rectangle */
   public void mouseDragged(MouseEvent e) {
      double x = Math.min(e.getX(), startX);
      double y = Math.min(e.getY(), startY);
      double width = Math.abs(e.getX() - startX);
      double height = Math.abs(e.getY() - startY);
      currentRect.setBounds(x, y, width, height);
   }

/* Private state */
   private GRect currentRect;      /* The current rectangle        */
   private double startX;          /* The initial mouse X position */
   private double startY;          /* The initial mouse Y position */
}
```

rectangle by dragging to the left, as shown in the following illustration:
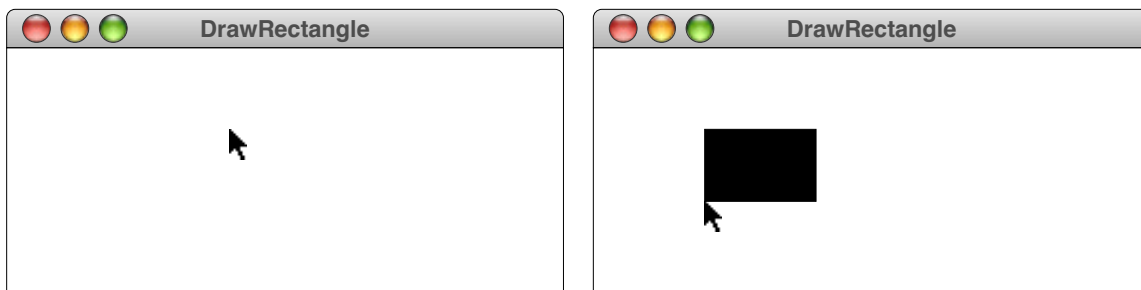
In this example, the origin of the **GRect** is no longer at the original position of the mouse. The width and height of a **GRect** cannot be negative, which means that the program needs to change the origin of the **GRect** whenever the mouse is moved to the left or upward.

8.5 Creating compound objects

The class from the **acm.graphics** hierarchy that has not yet been discussed is the **GCompound** class, which turns out to be extraordinarily handy. The **GCompound** class makes it possible to collect several **GObject**s together into a single unit, which is itself a **GObject**. This ability extends the notion of abstraction as discussed for methods into the domain of graphical objects. In much the same way that methods allow you to assemble many statements into a single unit, the **GCompound** class allows you to put together graphical objects into a single unit that has its own integrity as an graphical object.

To understand how the **GCompound** class works, it is easiest to start with a simple example. Imagine that you wanted to assemble the following face on the canvas:
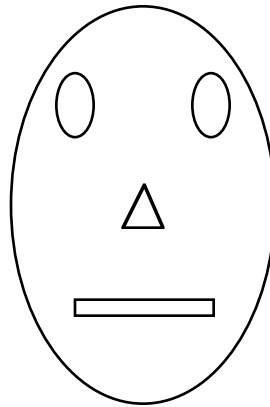
For the most part, this figure would be easy to create. All you need to do is create a new **GOval** for the head, two **GOval**s for the eyes, a **GRect** for the mouth, and a **GPolygon** for the nose. If you put each of these objects on the canvas individually, however, it will be hard to manipulate it as a unit. Suppose, for example, that you wanted to move the face around as a unit. Doing so would require moving every piece independently. It would be better simply to tell the entire face to move.

The code in Figure 8-11 uses the **GCompound** class to do just that. Figure 8-11 contains the code for a **GFace** class that extends **GCompound** to create a face object that contains the necessary components. These components are created and then added in the appropriate places by the first entry in the **GFace** class. This entry is not quite the same as a traditional method in the following respects:

• Its name matches the name of the class.
• It does not specify a return type

In Java, such methods are called **constructors** and are used to create new instances of a particular class. Here, the **GFace** constructor makes it possible to create a new **GFace** object by specifying the width and height in a statement like

```
GFace face = new GFace(200, 300);
```

FIGURE 8-11   A "graphical face" class defined using GCompound

```java
/*
 * File: GFace.java
 * ---------------
 * This file defines a compound GFace class.
 */

import acm.graphics.*;

public class GFace extends GCompound {

/** Construct a new GFace object with the specified dimensions. */
   public GFace(double width, double height) {
      head = new GOval(width, height);
      leftEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      rightEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      nose = createNose(NOSE_WIDTH * width, NOSE_HEIGHT * height);
      mouth = new GRect(MOUTH_WIDTH * width, MOUTH_HEIGHT * height);
      add(head, 0, 0);
      add(leftEye, 0.25 * width - EYE_WIDTH * width / 2,
                   0.25 * height - EYE_HEIGHT * height / 2);
      add(rightEye, 0.75 * width - EYE_WIDTH * width / 2,
                    0.25 * height - EYE_HEIGHT * height / 2);
      add(nose, 0.50 * width, 0.50 * height);
      add(mouth, 0.50 * width - MOUTH_WIDTH * width / 2,
                 0.75 * height - MOUTH_HEIGHT * height / 2);
   }

/** Creates a triangle for the nose */
   private GPolygon createNose(double width, double height) {
      GPolygon poly = new GPolygon();
      poly.addVertex(0, -height / 2);
      poly.addVertex(width / 2, height / 2);
      poly.addVertex(-width / 2, height / 2);
      return poly;
   }

/* Constants specifying feature size as a fraction of the head size */

   private static final double EYE_WIDTH    = 0.15;
   private static final double EYE_HEIGHT   = 0.15;
   private static final double NOSE_WIDTH   = 0.15;
   private static final double NOSE_HEIGHT  = 0.10;
   private static final double MOUTH_WIDTH  = 0.50;
   private static final double MOUTH_HEIGHT = 0.03;

/* Instance variables */

   private GOval head;
   private GOval leftEye, rightEye;
   private GPolygon nose;
   private GRect mouth;
}
```

which constructs a new **GFace** whose width is 200 and whose height is 300 pixels. Inside the constructor, the sizes of each component are expressed in terms of the width and height of the face as a whole, so that small faces have, for example, eyes of the appropriately small size. Once the constructor has created each of the features, it then adds each of them in turn, using calls like

```
add(nose, 0.50 * width, 0.50 * height);
```

which adds the nose object half the width and half the height from the upper left corner of the face, otherwise known as the center.

The first thing to recognize, however, is that this **add** call is not adding the objects to the canvas, but rather to the compound. The **add** method you've been using up to now is the one defined in **GraphicsProgram**, which does add objects to the **GCanvas** that it installs in the window. This version of the **add** method is defined in **GCompound**; the **GFace** class extends **GCompound** and therefore uses its add method. Adding things to a **GCompound** is intuitively analogous to the same operation at the canvas level except that the objects stay together as a unit. You can move everything in the **GCompound** simply by moving the **GCompound** itself.

Another important observation to make is that each **GCompound** has its own coordinate system. In that coordinate system, the point (0, 0) is the upper left corner of the compound and not of the entire canvas. This makes it possible for you to define a **GCompound** without having to know where it is going to appear on the canvas. When the components of a **GCompound** actually get drawn, they are shifted to the appropriate position.

Figure 8-12 contains the code necessary to create a **GFace** and allow the user to drag it as a unit, just as in the ObjectDrag example from the preceding section.

8.6 Principles of good object-oriented design

In contrast to the individual classes described in Chapter 6, the graphical facilities described in this chapter are part of a package—**acm.graphics**—that includes a large set of classes and interfaces. The contents of the package as a whole are summarized in the **javadoc** documentation, which also includes detailed descriptions of each class and method in the package. The package-level documentation appears in Figure 8-13; clicking on the various links shown on that page will take you to the more detailed documentation for the classes that comprise the **acm.graphics** package.

No matter at what level you are working—individual methods, classes that offer a suite of those methods, or packages that provide a set of classes—you need to think carefully about design. If you were programming entirely for your own amusement, design issues might seem relatively unimportant. But that situation rarely applies in the computing industry. Programs are developed cooperatively, with programmers on different parts of a large project agreeing to share a common set of stylistic conventions and to coordinate the ways in which their independent pieces fit together. In the absence of cooperative agreements, the programming process would descend—as it all too often does—into chaos. The important question, therefore, is how should one design methods, classes, and packages for *other* programmers to use. What are the principles that underlie well-chosen designs?

FIGURE 8-12   **A program to drag a GFace object**

```java
/*
 * File: DragFace.java
 * ------------------
 * This program creates a GFace object and allows the user to drag
 * it around the canvas.
 */

import java.awt.event.*;

import acm.graphics.*;
import acm.program.*;

public class DragFace extends GraphicsProgram {

/** Width of the face */
   private static final double FACE_WIDTH = 200;

/** Height of the face */
   private static final double FACE_HEIGHT = 300;

/** Runs the program */
   public void run() {
      GFace face = new GFace(FACE_WIDTH, FACE_HEIGHT);
      double x = (getWidth() - FACE_WIDTH) / 2;
      double y = (getHeight() - FACE_HEIGHT) / 2;
      add(face, x, y);
      addMouseListeners();
   }

/** Called on mouse press to record the coordinates of the click */
   public void mousePressed(MouseEvent e) {
      lastX = e.getX();
      lastY = e.getY();
      gobj = getElementAt(lastX, lastY);
   }

/** Called on mouse drag to reposition the object */
   public void mouseDragged(MouseEvent e) {
      if (gobj != null) {
         gobj.move(e.getX() - lastX, e.getY() - lastY);
         lastX = e.getX();
         lastY = e.getY();
      }
   }

/** Called on mouse click to move this object to the front */
   public void mouseClicked(MouseEvent e) {
      if (gobj != null) gobj.sendToFront();
   }

/* Private state */
   private GObject gobj;            /* The object being dragged   */
   private double lastX;            /* The last mouse X position  */
   private double lastY;            /* The last mouse Y position  */
}
```

FIGURE 8-13   **Documentation page for the acm.graphics package**

Overview **Package** Class Tree Index Help
PREV PACKAGE NEXT PACKAGE   FRAMES   NO FRAMES

Package acm.graphics

This package provides a set of classes that support the creation of simple, object-oriented graphical displays.

See:
   Description

Interface Summary

GContainer	Defines the functionality of an object that can serve as the parent of a `GObject`.
GFillable	Specifies the characteristics of a graphical object that supports filling.
GResizable	Specifies the characteristics of a graphical object that supports the `setSize` and `setBounds` methods.
GScalable	Specifies the characteristics of a graphical object that supports the `scale` method.

Class Summary

Gue3DRect	The `G3DRect` class is a graphical object whose appearance consists of a three-dimensional rectangle.
GArc	The `GArc` class is a graphical object whose appearance consists of an arc.
GCanvas	The `GCanvas` class is a lightweight component that also serves as a container for graphical objects.
GCanvasMenuBar	The `ConsoleMenuBar` class provides a standard menu bar for frames containing a console.
GCompound	This class defines a graphical object that consists of a collection of other graphical objects.
GDimension	This class is a double-precision version of the `Dimension` class in `java.awt`.
GImage	The `GImage` class is a graphical object whose appearance is defined by an image.
GLabel	The `GLabel` class is a graphical object whose appearance consists of a text string.
GLine	The `GLine` class is a graphical object whose appearance consists of a line segment.
GObject	This class is the common superclass of all graphical objects that can be displayed on a `GCanvas`.
GOval	The `GOval` class is a graphical object whose appearance consists of an oval.
GPen	The `GPen` class simulates a pen drawing on a canvas.
GPoint	This class is a double-precision version of the `Point` class in `java.awt`.
GPolygon	The `GPolygon` class is a graphical object whose appearance consists of a polygon.
GRect	The `GRect` class is a graphical object whose appearance consists of a rectangular box.
GRectangle	This class is a double-precision version of the `Rectangle` class in `java.awt`.
GRoundRect	The `GRoundRect` class is a graphical object whose appearance consists of a rounded rectangle.
GTurtle	The `GTurtle` class simulates a turtle moving on a canvas.

Developing a solid understanding of those principles requires you to understand that clients and implementors look at the facilities provided by a method, class, or package from different perspectives. Clients want to know what operations are available and are unconcerned about the details of the implementation. For implementors, the details of how those operations work are the fundamental issue.

It is, however, essential for implementors to keep the needs of clients in mind. If you are trying to design effective resources for others, you need to balance several criteria. Those criteria are described here at the level of an individual class, but these same considerations also apply at the lower level of individual methods and the higher level of complete packages:

- *Unified.* A class should define a consistent abstraction with a clear unifying theme. If a class does not fit within that theme, it should not be part of the class.
- *Simple.* The class design should try to simplify things for the client. To the extent that the underlying implementation is itself complex, the class must seek to hide that complexity.
- *Sufficient.* For clients to use a class, it must provide sufficient functionality to meet their needs. If some critical operation is missing from a class, clients may decide to abandon it and develop their own tools. As important as simplicity is, the designer must avoid simplifying a class to the point that it becomes useless.
- *General.* A well-designed class should be flexible enough to meet the needs of many different clients. A class that performs a narrowly defined set of operations for one client is not nearly as useful as one that can be used in many different situations.
- *Stable.* The methods defined in a class should continue to have precisely the same structure and effect, even as the package that includes it evolves. Making changes in the behavior of a class forces clients to change their programs, which reduces the utility of that class.

The sections that follow discuss each of these criteria in detail.

The importance of a unifying theme

Unity gives strength.

— Aesop, *The Bundle of Sticks,* 6th century BCE

A central feature of a well-designed class is that it presents a unified and consistent abstraction. In part, this criterion implies that the methods within a class should be chosen so that they reflect a coherent theme. For example, the `Math` class consists of mathematical methods, the `ConsoleProgram` class provides methods that make it easy to converse with a user typing at a console, and the various classes in the `acm.graphics` package provide methods for arranging graphical objects on a canvas. Each method exported by these classes fits the purpose of that class. For example, you would not expect to find `sqrt` in the `GObject` class, even though graphical applications will often call `sqrt` to compute the length of a diagonal line. The `sqrt` method fits much more naturally into the framework of the `Math` class.

The principle of a unifying theme also influences the design of the methods within a class. The methods within a class should behave in as consistent a way as possible. Differences in the ways its methods operate make using a class much harder for the client. For example, all the methods in the `acm.graphics` package use coordinates specified in pixels and angles specified in degrees. If the implementor of the class had decided to toss in a method that required a different unit of measurement, clients would have to remember what units to use for each method.

Simplicity and the principle of information hiding

Embrace simplicity.

— Lao-tzu, *The Way of Lao-tzu,* ca. 550 BCE

Because a primary goal of using classes is to reduce the complexity of the programming process, it makes sense that simplicity is a desirable criterion in the design of a class. In general, a class should be as easy to use as possible. The underlying implementation may perform extremely intricate operations, but the client should nonetheless be able to think about those operations in a simple, more abstract way.

As noted earlier in this chapter, the documentation that you need to use Java classes and packages is usually presented on the web. If, for example, you want to know how to use the **Math** class, you go to the web page that contains its javadoc description. If that documentation is well-designed, it tells you precisely the information that you need to know as a client, but no more. For clients, getting too much information can be as bad as getting too little, because additional detail is likely to make the class more difficult to understand. Often, the real value of a class lies not in the information it *reveals* but rather in the information it *hides*.

When you design a class, you should try to protect the client from as many of the complicating details of the implementation as possible. In that respect, it is perhaps best to think of a class not primarily as a communication channel between the client and the implementation, but instead as a wall that divides them.

<div align="center">
client   *implementation*
</div>
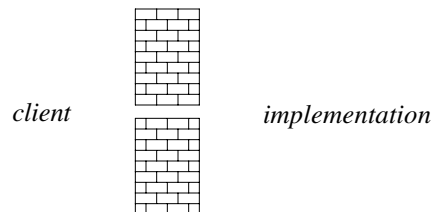
Like the wall that divided the lovers Pyramus and Thisbe in Greek mythology, the wall representing a class has a small chink that allows the client and the implementation to communicate. The main purpose of the wall, however, is to keep the two sides apart. Because it forms the border between the abstraction perspectives on each side, that wall is often called an **abstraction boundary**. Ideally, all the complexity involved in the realization of a class lies on the implementation side of the wall. The design is successful if it keeps that complexity away from the client side. Keeping details confined to the implementation domain is called **information hiding**.

The principle of information hiding has important practical implications for class design. When you write a class, you should be sure you don't reveal details of the implementation, even in the commentary. Especially if you are writing a class and an implementation at the same time, you may be tempted to document in your class all the clever ideas you used to write the implementation. Try to resist that temptation. The class is written for the benefit of the client and should contain only what the client needs to know.

Similarly, you should design the methods in a class so that they are as simple as possible. If you can reduce the number of arguments or find a way to eliminate confusing special cases, it will be easier for the client to understand how to use those methods. Moreover, it is usually good practice to limit the total number of methods exported by class, so that the client does not become lost in a mass of methods, unable to make sense of the whole.

Meeting the needs of your clients

<div align="center">
Everything should be as simple as possible, but no simpler.
</div>

<div align="right">
— attributed to Albert Einstein
</div>

Simplicity is only part of the story. You can easily make a class simple just by throwing away any parts of it that are hard or complicated. There is a good chance you will also make the class useless. Sometimes clients need to perform tasks that have some inherent complexity. Denying your clients the tools they require just to make the class simpler is

not an effective strategy. Your class must provide sufficient functionality to serve the clients' needs. Learning to strike the right balance between simplicity and completeness in class design is one of the fundamental challenges in programming.

In many cases, the clients of a class are concerned not only with whether a particular method is available but also with the efficiency of the underlying implementation. For example, if a programmer is developing a system for air-traffic control and needs to call methods provided by a class, those methods must return the correct answer quickly. Late answers may be just as devastating as wrong answers.

For the most part, efficiency is a concern for the implementation rather than the abstract design. Even so, you will often find it valuable to think about implementation strategies while you are designing the class itself. Suppose, for example, that you are faced with a choice of two designs. If you determine that one of them would be much easier to implement efficiently, it makes sense—assuming there are no compelling reasons to the contrary—to choose that design.

The advantages of general tools

> *Give us the tools and we will finish the job.*
>
> — Winston Churchill, radio address, 1941

A class that is perfectly adapted to a particular client's needs may not be useful to others. A good class abstraction serves the needs of many different clients. To do so, it must be general enough to solve a wide range of problems and not be limited to one highly specific purpose. By choosing a design that offers your clients flexibility in how they use the abstraction, you can create classes that are widely used.

The desire to ensure that a class remains general has an important practical implication. When you are writing a program, you will often discover that you need a particular tool. If you decide that the tool is important enough to go into a class, you then need to change your mode of thought. When you design the class for that class, you have to forget about the application that caused you to want the tool in the first place and instead design such a tool for the most general possible audience.

The value of stability

> *People change and forget to tell each other. Too bad—causes so many mistakes.*
>
> — Lillian Hellman, *Toys in the Attic*, 1959

Class and package designs have another property that makes them critically important to programming: they tend to be stable over long periods of time. Stable classes can dramatically simplify the problem of maintaining large programming systems by establishing clear boundaries of responsibility. As long as the client perspective on a class does not change, both implementors and clients are free to make changes on their own side of the abstraction boundary.

For example, suppose that you are the implementor of the `Math` class. In the course of your work, you discover a clever new algorithm for calculating the `sqrt` method that cuts in half the time required to calculate a square root. If you can say to your clients that you have a new implementation of `sqrt` that works just as it did before, only faster, they will probably be pleased. If, on the other hand, you were to say that the name of the method had changed or that its use involved certain new restrictions, your clients would be justifiably annoyed. To use your "improved" implementation of square root, they would

be forced to change their programs. Changing programs is a time-consuming, error-prone activity, and many clients would happily give up the extra efficiency for the convenience of being able to leave their programs alone.

Summary

In this chapter, you have had the chance to explore the **acm.graphics** package in more detail and to develop an appreciation of the entire package as an integrated collection of tools. Along the way, you have also had the opportunity to think holistically about the design of the graphics package and the assumptions, conventions, and metaphors on which the package is based.

Important points introduced in this chapter include:

- The set of assumptions, conventions, and metaphors that underlie the design of a package represent its conceptual *model*. Before you can use a package effectively, you must take the time to understand the model on which it is based.

- An essential part of the model for the **acm.graphics** package is the coordinate system that it uses. The **acm.graphics** package follows the conventions of the standard graphics packages in Java by specifying coordinates in *pixels* and placing the *origin* in the upper left corner of a canvas. This coordinate system is different from the Cartesian plane used in high-school geometry classes, which has its origin in the lower left.

- The foundation of the **acm.graphics** package is the **GObject** class, which is the common superclass of all objects that can be displayed on a canvas. The **GObject** class itself is an *abstract class,* which means that there are no objects whose primary class is **GObject**. When you create a graphical image on a canvas, the classes that you actually use are the subclasses of **GObject** called *shape classes:* **GArc**, **GImage**, **GLabel**, **GLine**, **GOval**, **GPolygon**, **GRect**, **GRoundRect**, and **G3DRect**.

- Each of the shape classes inherits a set of methods from **GObject** that all graphical objects share. In addition, each shape class includes additional methods that define its particular behavior. Several of the shape classes also share common behavior by virtue of implementing one or more of the interfaces **GFillable**, **GResizable**, and **GScalable**.

- To display a graphical object, you need to add it to a **GCanvas**, which serves as the background for a "collage" of **GObject**s. In most cases, that **GCanvas** will be provided automatically as part of a **GraphicsProgram**, although you can also create your own **GCanvas** objects and use them independently of the **acm.program** package.

- You can make your program respond to mouse events by implementing one or more of the following *listener methods:* **mousePressed**, **mouseReleased**, **mouseClicked**, **mouseMoved**, **mouseDragged**, **mouseEntered**, and **mouseExited**. In most cases, it makes sense to use the **GCanvas** as the source of these events; you can enable the event listeners in the **GCanvas** by calling **addMouseListeners** as part of the **run** method.

- You can use the **GCompound** type to assemble individual objects into larger structures that you can then manipulate as a unit.

- A well-designed package must be *unified*, *simple*, *sufficient*, *general*, and *stable*. Since these criteria sometimes conflict with each other, you must learn to strike an appropriate balance as you design your programs.

Review questions

1. Why does the text describe the graphical framework used in the **acm.graphics** package as a "collage" model?

2. What unit of measurement is used to specify coordinates in **acm.graphics**?

3. Where is the origin located in the **acm.graphics** coordinate system?

4. The text of the chapter emphasizes that **GObject** is an *abstract class*. What is the significance of that designation?

5. Without looking back at the figure in this chapter, draw a diagram that shows the class hierarchy formed by the **GObject** class and the shape classes **GArc**, **GImage**, **GLabel**, **GLine**, **GOval**, **GPolygon**, **GRect**, **GRoundRect**, and **G3DRect**.

6. What methods are defined by the **GFillable** interface? Which of the shape classes implement **GFillable**?

7. What is the difference between the **GResizable** and **GScalable** interfaces? Which of the shape classes are scalable but not resizable?

8. In terms of the geometric characteristics of **GLabel**s displayed on a canvas, explain the significance of the values returned by each of the following methods: **getWidth**, **getHeight**, **getAscent**, and **getDescent**.

9. Describe the significance of the **start** and **sweep** parameters in the constructor for the **GArc** class?

10. How does the **acm.graphics** package interpret filling in the case of the **GArc** class?

11. Describe the arcs produced by each of the following calls to the **GArc** constructor:

    a) **new GArc(2.0, 2.0, 0, 270);**
    b) **new GArc(2.0, 2.0, 135, -90);**
    c) **new GArc(2.0, 2.0, 180, -45);**
    d) **new GArc(3.0, 1.0, -90, 180);**

12. What does it mean if the *sweep* argument to the **GArc** constructor is negative?

13. For the **GLine** class, how do the methods **setLocation** and **setStartPoint** differ?

14. Write the Java statements necessary to create each of the following polygonal shapes as a single **GPolygon**:

    a)   b)   c)
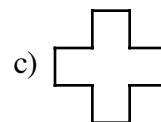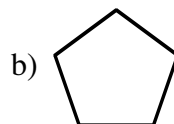
15. How can you obtain the coordinates of the center of the drawing canvas used by a **GraphicsProgram**?

16. What is an *event listener?*

17. Describe the purpose of each of the seven mouse listener methods: **mousePressed**, **mouseReleased**, **mouseClicked**, **mouseMoved**, **mouseDragged**, **mouseEntered**, and **mouseExited**.

18. In what package is the **MouseEvent** class defined?

19. In your own words, describe the purpose of the **GCompound** class.

20. What are the five criteria for good package design listed in the last section of this chapter?

Programming exercises

1. Write a **GraphicsProgram** subclass that draws a pyramid consisting of bricks arranged in horizontal rows, so that the number of bricks in each row decreases by one as you move up the pyramid, as shown in the following sample run:
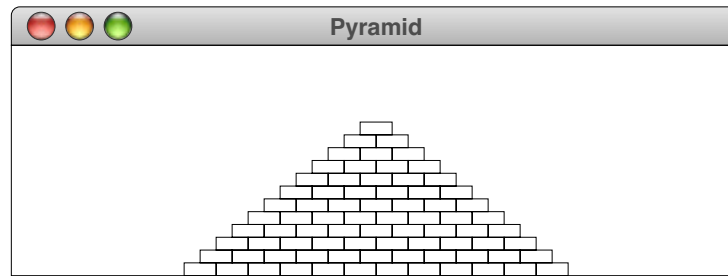
The pyramid should be centered at the bottom of the window and should use constants for the following parameters:

BRICK_WIDTH	The width of each brick (30 pixels)
BRICK_HEIGHT	The height of each brick (12 pixels)
BRICKS_IN_BASE	The number of bricks in the base (12)

The numbers in parentheses show the values for this diagram, but you must be able to change those values in your program.

2. One way to draw a heart-shaped figure is by drawing two semicircles on top of a square that is positioned so that its sides run diagonally, as illustrated by the following diagram:
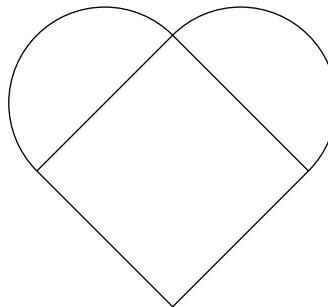
Write a **GraphicsProgram** that uses this construction to draw a heart on the screen using the classes **GArc** and **GLine**. Your program should display the heart without drawing the interior lines that form the top of the square, so the output looks like this:
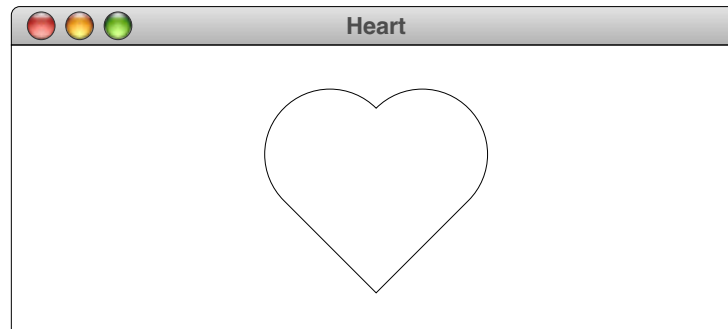
3. Change your definition of the heart from the preceding exercise so that you can generate a filled red heart as a single **GCompound** unit whose origin in the center of the square. The pieces of the **GCompound** are now two **GArc**s (or **GOval**s) and a **GPolygon**, as shown in the following sample run:
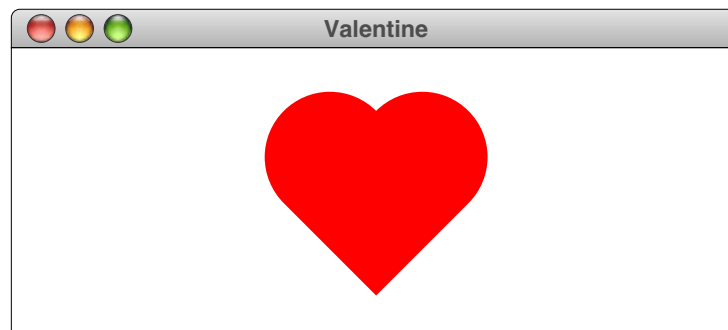
4. In the 1960s, this symbol

became universally identified as the *peace symbol,* and it still shows up from time to time as a motif for T-shirts or jewelry. The peace symbol took its form from the letters N and D—the initial letters in *nuclear disarmament*—as expressed in the international semaphore code:

N

D
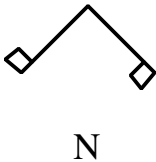
The peace symbol is formed by superimposing the lines in these two diagrams (without the flags) and enclosing them in a circle.

Implement a method **drawPeaceSymbol** with the header line

```
void drawPeaceSymbol(double x, double y, double r)
```

that draws a peace symbol centered at the point (x, y) with a circle of radius r. Write a subclass of **GraphicsProgram** to test your method.

5. Write a **GraphicsProgram** to draw your initials on the graphics window using only the **GArc** and **GLine** classes rather than **GLabel**. For example, if I wrote this program, I would want the output to be
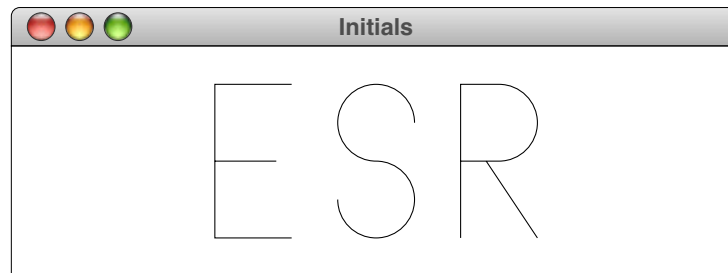
Think about the best decomposition to use in writing the program. Imagine that you've been asked to design a more general letter-drawing library. How would you want the methods in that library to behave in order to make using them as simple as possible for your clients?

6. Write a **GraphicsProgram** that draws a picture of the Halloween pumpkin shown in the following diagram:

The head is an orange circle, and the eyes and mouth are filled polygons. The stem is presumably a **GRect**. Use named constants in your program to define the sizes of the various features.

7. Write a **GraphicsProgram** that draws a line drawing of the house shown in the following diagram:
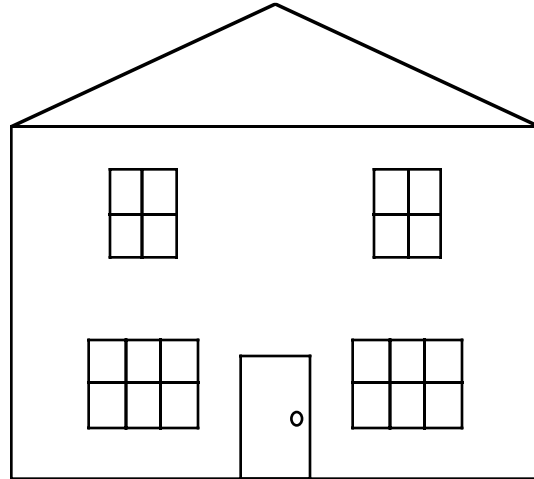
Make sure that you use stepwise refinement to decompose this figure into useful pieces.

8. If you wanted a house to go along with the Halloween pumpkin you designed in exercise 5, you might want to draw a diagram of the House of Usher, which Edgar Allen Poe describes as follows:

> I looked upon the scene before me . . . upon the bleak walls—upon the vacant eye-like windows . . . with an utter desperation of soul . . .

From Poe's description, you might imagine a house that looks something like this:
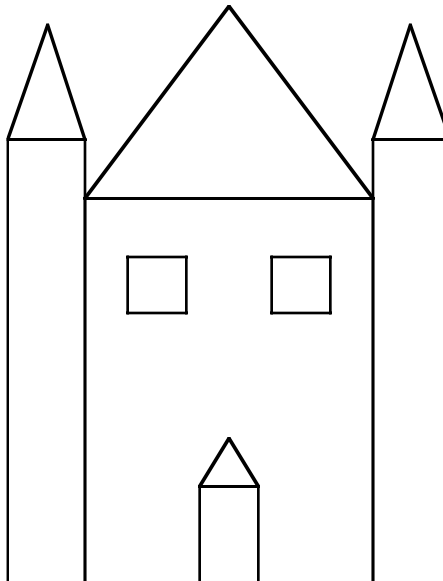
Write a **GraphicsProgram** that draws the house illustrated in the diagram, using named constants to specify the various dimensions.

9. Write a **GraphicsProgram** that draws the following stylized picture of the Lincoln Memorial in Washington, D.C.:
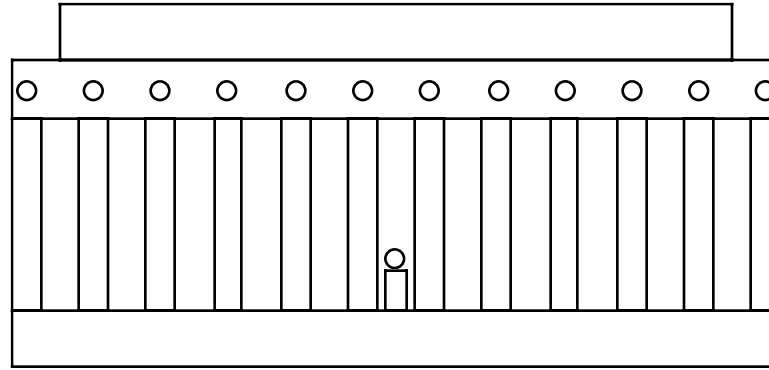
10. Write a **GraphicsProgram** that creates **GLabel**s for each of the color names **RED**, **ORANGE**, **YELLOW**, **GREEN**, **CYAN**, **BLUE**, and **MAGENTA**, and then puts those labels up on the screen in a random position and in a random color. It turns out to be difficult to identify the color of such a label if the name says one thing, but its color is different.

11. Modify the program you wrote in exercise 10 so that pressing the mouse button on top of one of the **GLabel**s temporarily resets its color to the one that matches its name. Releasing the mouse button should choose a new random color for the label.

12. Extend the **DrawRectangle** program from Figure 8-10 so that clicking the mouse inside an existing rectangle allows you to drag that rectangle to a new position on the canvas.

13. Write a program that animates the "Beacons of Gondor" simulation from section 7.4. Your program should redefine the **SignalTower** class so that it extends **GCompound** and includes a constructor that fills in all the components of the desired tower shape. For example, if your signal tower is just a box with the name of the tower underneath it, the chain of towers might look like this:
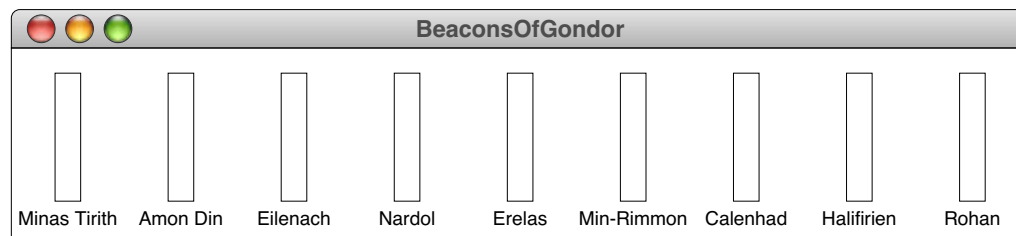
Redefine the **lightCurrentTower** method so that it fills in the tower rectangle, changes its color to red, and then pauses for half a second or so. Finally, define a mouse listener for the **SignalTower** objects so that clicking on a tower sends a **signal** message to that tower, which in turn propagates to the towers in the chain to the right. Thus, if you were to click on the tower at Minas Tirith, the beacons should turn red one at a time as the signal moves down the line.

14. Extend the **DrawRectangle** program so that the left side of the canvas includes a palette of the five following shapes: a filled rectangle, an outlined rectangle, a filled oval, an outlined oval, and a straight line, each of which are enclosed in a small square as shown in the following diagram:
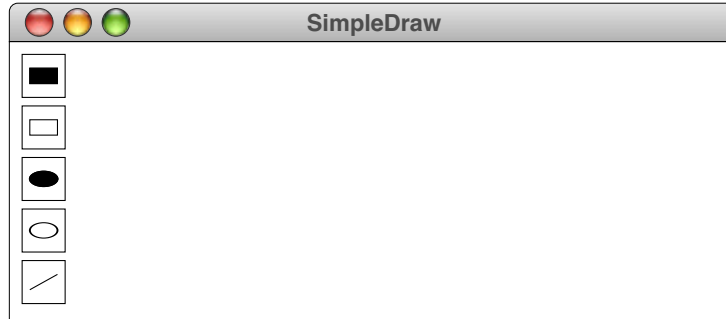
Clicking on one of the squares in the palette chooses that shape. Thus, if the user clicks on the filled oval in the middle of the palette, the program should draw filled ovals. Clicking and dragging outside of the palette area should draw the currently selected shape.

15. Write a **GraphicsProgram** that draws the message

𝕴 𝕷𝖔𝖛𝖊 𝕵𝖆𝖛𝖆

in 18-point London type (or any font you have that's close) and bounces it around the canvas. Whenever any edge of the **GLabel** would cross the boundary, reverse its speed in the appropriate direction.

Chapter 9
Strings and Characters

Surely you don't think numbers are as important as words.

— King Azaz to the Mathemagician
Norton Juster, *The Phantom Tollbooth*, 1961
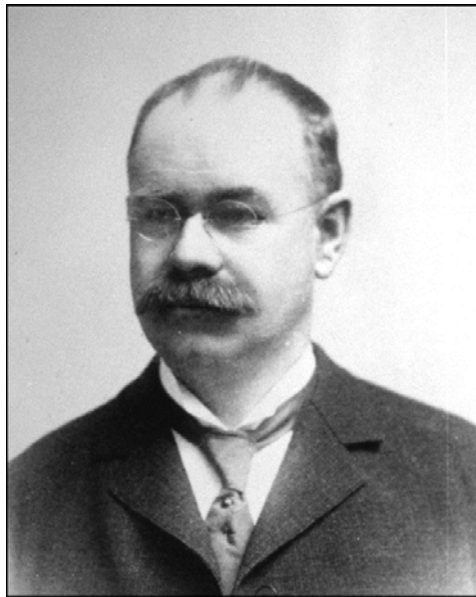
Herman Hollerith (1860-1929)

Herman Hollerith studied engineering at City College of New York and the Columbia School of Mines. On completing his degree, Hollerith spent a couple of years working as a statistician for the U.S. Census Bureau before accepting a teaching position at MIT. After two years there, Hollerith left academia for a position with the U.S. Patent Office that helped pave the way for his own later career as an inventor. While at the Census Bureau, Hollerith had become convinced that the data produced by the census could be counted more quickly and accurately by machine. In the late 1880s, he designed and built a tabulating machine that was used to conduct the 1890 census in record time. The company he founded to commercialize his invention was originally called the Tabulating Machine Company but changed its name to International Business Machines in 1924. Hollerith's card-based tabulating system also pioneered the technique of textual encoding described in this chapter—a contribution that was reflected in the fact that early versions of the FORTRAN language used the letter ʜ (for Hollerith) to indicate text data.

Although you have had a chance to see a number of programs that display graphical objects on a canvas, the only type of data with which you have actually worked—in the traditional sense of using as the basis of calculation or similar forms of data manipulation—has been numeric data, as represented by the types **int** and **double** and numbers as their basic data type. As Juster's Mathemagician would insist, numbers are certainly important, but there are many other kinds of data in the world. Particularly since the development of the personal computer in the early 1980s, computers have worked less with numeric data than they have with **text data**, which represents information composed of individual characters that appear on the keyboard and the screen. The ability of computers to process text data has led to the development of word processing systems, electronic mail, the World Wide Web, search engines, and a wide variety of other useful applications.

The concept of text data was introduced informally in Chapter 2, beginning with the first line of code in this book, which includes the string

```
"hello, world"
```

Since then, you have used string values from time to time but, in each case, have looked at those strings only as an integrated whole. This perspective is an essential one, and indeed will be true most of the time that you use string data. To unlock the full power of strings, you need to know how to manipulate strings in more sophisticated ways.

Because a string is composed of individual characters, it is important for you to understand how characters work and how they are represented inside the computer. Thus, this chapter focuses on the data type **char** as well as the data type **String**. Before examining the details of either type, however, it is helpful to begin by taking a more general look at how data can be represented inside the computer.

9.1 The principle of enumeration

As the use of computing technology grows, more and more information is stored electronically. To store information within a computer, it is necessary to represent the data in a form the machine can use. The representation of a particular item depends on its data type. Integers have one representation inside the computer; floating-point numbers have a different one. Even though you do not know exactly what those representations look like, you have relied on the fact that the computer is able to store numbers in its internal memory. There are, however, many types of useful data other than numbers, so computers must be able to represent nonnumeric data as well.

To gain insight into the nature of nonnumeric data, think for a moment about the information that you yourself provide to institutions and agencies over the course of a year. For example, if you live in the United States, you supply data to the Internal Revenue Service with your annual tax return. Much of that information is numeric—your salary, deductions, taxes, withholdings, and the like. Some consists of text data, such as your name, address, and occupation. But other items on your tax return cannot easily be classified into either of these forms. For example, one of the questions is

Filing status (check one):
☐ single
☐ married filing joint return
☐ married filing separate return
☐ head of household
☐ qualifying widow(er)

As with every other entry on the form, your answer represents data. Your response, however, is neither numeric data nor text data. The best way to describe the data type would be simply to call it *filing status data*—an entirely new data type whose domain consists of five values: *single, married filing joint return, married filing separate return, head of household,* and *qualifying widow(er)*.

You can easily imagine many other data types that have a similar structure. For example, other forms might ask you for your sex, ethnicity, or status as a student. In each case, you would choose a response from a list of possibilities that constitutes the domain of a distinct conceptual type. The process of listing all the elements in the domain of a type is called **enumeration**. A type defined by listing all of its elements is called an **enumerated type**.

Because the title of this chapter is "Strings and Characters," discussing enumerated types might seem like a digression. As it happens, though, characters are similar in structure to enumerated types. Understanding how enumerated types work will help you appreciate how characters work.

At this point, however, enumerated types are an abstract concept. To understand how they apply to programming, you need to learn how the computer represents such values internally. You must also learn how to use enumerated types in the context of a Java program. The next two sections address these issues.

Representing enumerated types inside the machine

If the Internal Revenue Service decides to review your tax return, the first step in the process is to enter the data from your return into a computer system. To store that data, the computer must have a way of representing each of the different data items, including your filing status. If you were developing a strategy for recording a taxpayer's filing status, what would you do?

The insight you need to solve this problem comes from building on the capabilities you know computers have. Computers are good at working with numbers. That's how they're built. As part of their basic hardware operation, they can store, add, subtract, compare, and do all sorts of other things with numbers. The fact that computers are good at manipulating numbers suggests a solution to the problem of representing an enumerated type. To represent a finite set of values of any type, all you have to do is give each value a number. For example, given the list of allowable filing status values, you could simply count them off, letting *single* be 1, *married filing joint return* be 2, *married filing separate return* be 3, and so on. (In fact, these numeric codes are listed directly on the tax form.) Assigning an integer to each of the different possibilities means that you can use that integer to represent the corresponding filing status.

Thus all you have to do to define a representation for any enumerated type is to number its elements. The process of assigning an integer to each element of an enumerated type is called **integer encoding**—the integer acts as a coded representation of the original value.

Representing enumerated types as integers

For most of its history, Java programmers have adopted the strategy of integer encoding in the most explicit way possible. Because the language lacked any higher-level support for enumerated types, the only mechanism that was available was to define named constants for each value you wanted to represent. Thus, in a program that needed to

record the various filing categories offered by the Internal Revenue Service, you might expect to see the following definitions in a Java program:

```
public static final int SINGLE = 1;
public static final int MARRIED_FILING_JOINT_RETURN = 2;
public static final int MARRIED_FILING_SEPARATELY = 3;
public static final int HEAD_OF_HOUSEHOLD = 4;
public static final int QUALIFYING_WIDOW_OR_WIDOWER = 5;
```

These definitions make it possible to use these names in your program. For example, if you wanted to perform some code only if the filing status were *single,* you would use the following **if** statement:

```
if (filingStatus == SINGLE)
```

In using integer constants to represent enumerated types, there are a few cautions that you should keep in mind:

- The variables that hold the enumerated values must be declared as being of type **int**. In many ways, this fact is unfortunate because the conceptual domain of the value is by no means everything encompassed by the type **int**. It is not at all clear what a filing status of –1 or 999 might be, although those are legal integers.
- The definition of constants in the program does not permit you to enter these values externally. In particular, if your program were to request the filing status by issuing the command

```
int filingStatus = readInt("Enter filing status: ");
```

 you could not enter **MARRIED_FILING_SEPARATELY** in response.
- There is no automatic mechanism for displaying the value of an enumerated type so that its name appears in the output. If the value of **filingStatus** were **SINGLE**, calling

```
println("filing status = " + filingStatus);
```

 would simply display a line indicating that the filing status was 1.

Despite these shortcomings, enumerated types are still valuable because they increase the readability of the code.

Fortunately, the problems associated with using named constants to achieve the effect of enumerated types are going away with the introduction of a new facility in Java Standard Edition 5.0 that supports the creation of programmer-defined enumerated types. Unfortunately, that facility is too new to use at this point in its development. It is, for example, not yet supported on the Macintosh platform or in anything other than the most recent version of the Java runtime. Until it matures, we won't be able to rely on this facility and continue to support a wide range of platforms.

9.2 Characters

Characters form the basis for all text data processing. Although strings certainly occur more often in programs than single characters, characters are the fundamental type—the "atoms" used to construct all other forms of text data. Understanding how characters work is therefore critical to understanding all other aspects of text processing. In a sense, characters constitute a built-in enumerated type, although the complete list of possible

characters is too large to list in its entirety. It is more appropriate to describe the domain of all characters as a **scalar type,** which is any type that can be interpreted as an integer. Scalar types are extremely useful in Java because you can use them in any context in which an integer might appear. For example, a variable of a scalar type can be used as the control expression in a `switch` statement.

The data type `char`

In Java, single characters are represented using the data type `char`, which is one of the predefined data types. Like all the basic types introduced in Chapter 3, the type `char` consists of a domain of legal values and a set of operations for manipulating those values. Conceptually, the domain of the data type `char` is the set of symbols that can be displayed on a screen or typed on a keyboard. These symbols—the letters, digits, punctuation marks, spacebar, Return key, and so forth—are the building blocks for all text data.

Because `char` is a scalar type, the set of operations available for characters is the same as that for integers. Understanding what those operations mean in the character domain, however, requires looking more closely at how characters are represented inside the machine.

The ASCII and Unicode coding systems

Single characters are represented inside the machine just like any other scalar type. Conceptually, the central idea is that you can assign every character a number by writing them all down in a list and then counting them off one at a time. The code used to represent a particular character is called its **character code**. For example, you could let the integer 1 represent the letter *A*, the integer 2 represent the letter *B*, and so on. After you got to the point of letting 26 represent the letter *Z*, you could then keep going and number each of the lowercase letters, digits, punctuation marks, and other characters with the integers 27, 28, 29, and so on.

Even though it is technically possible to design a computer in which the number 1 represents the letter *A*, it would certainly be a mistake to do so. In today's world, information is often shared between different computers: you might copy a program from one machine to another on a floppy disk or arrange to have your computer communicate directly with others over a national or international network. To make that kind of communication possible, computers must be able to "talk to each other" in a common language. An essential feature of that common language is that the computers use the same codes to represent characters, so that the letter *A* on one machine does not come out as a *Z* on another.

In the early days of computing, different computers actually used different character codes. The letter *A* might have a particular integer representation on one machine but an entirely different representation on a computer made by some other manufacturer. Even the set of available characters was subject to change. One computer, for example, might have the character ¢ on its keyboard, while another computer would not be able to represent that character at all. Computer communication was plagued by all the difficulties that people speaking different languages encounter.

Over time, however, the enormous advantage that comes from enabling computers to communicate effectively led to the adoption of a coding system for characters called **ASCII,** which stands for the *American Standard Code for Information Interchange*. The ASCII coding system became quite widespread in the 1980s and represented something of a standard. It was, however, closely tied to the characters one uses to represent

English and not the many other languages in the world. Particularly with the rise of the World Wide Web in the 1990s, it became necessary to expand the domain of the character type to encompass a much broader collection of languages. The result of that expansion was a new coding system called **Unicode**, which is intended to be more universal in its application. Java was one of the first languages to adopt Unicode as its representation for characters, making it much more appropriate for the international framework in which computers operate today.

The Unicode system does make it a little harder to illustrate the principle of enumeration, because there is no way to write down a list of all the characters. The ASCII code supported only 256 possible characters, of which only the first 128 were fully standardized. A character set of that size was by no means sufficient to represent all available languages, but it was nonetheless easy to make a table of ASCII values. Unicode has space for 65,536 characters, although not all of those characters have yet been defined. This limit may still prove too small, but it is certainly an improvement in flexibility over the far more limited range of ASCII.

Fortunately, the designers of Unicode decided to incorporate the standard ASCII characters as the first 128 elements in the Unicode set. These characters are shown with their codes in Figure 9-1, which shows the ASCII portion of the Unicode table. Most of the entries in the table are familiar characters that appear on the keyboard, but there are several less familiar entries represented by a backward slash (\), usually called a **backslash,** followed by a single letter or a sequence of digits. These entries are called **special characters** and are discussed in a separate section later in this chapter.

FIGURE 9-1	The ASCII portion of the Unicode table						
	0	1	2	3	4	5	6	7
00x	\000	\001	\002	\003	\004	\005	\006	\007
010	\b	\t	\n	\013	\f	\r	\016	\017
020	\020	\021	\022	\023	\024	\025	\026	\027
030	\030	\031	\032	\033	\034	\035	\036	\037
040	*space*	!	"	#	$	%	&	'
050	(	)	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	^	_
140	`	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{			}	~	\177

There is, however, another aspect of the table that is likely to be even more confusing than the special characters. The column headings run from 0 to 7 instead of the 0 to 9 one might expect in a table whose entries specify numeric codes. Similarly, the row headings at first appear to be increasing at a predictable rate but then jump from 070 to 100. The reason for this behavior is that ASCII and Unicode values are not typically presented in decimal (base 10), but rather in a base that is easier to convert to the internal representation of numbers as a sequence of bits. The bases that permit this kind of easy translation are those that derive from a power of two, of which the most common—and the only ones supported directly by Java—are **octal** (base 8) and **hexadecimal** (base 16). Figure 9-1 uses octal notation, in which the only digits fall in the range from 0 to 7. Numeric bases were discussed briefly in Chapter 6; for now, the only thing you need to remember about base-8 notation is that every digit position is worth 8 times as much as the digit to its right. For example, the octal number 177 corresponds to the integer

$$1 \times 64 \ + \ 7 \times 8 \ + \ 7$$

which works out to be 127 in traditional decimal notation. In this text, numbers that are written in a base other than decimal will be marked with a subscript indicating the base, so that 177_8 specifies an octal value.

The internal code for each character in Figure 9-1 is the sum of the octal row and column number associated with that entry. For example, the letter *A* halfway down the chart is in the row labeled 100 and the column labeled 1. The Unicode value for the letter *A* is therefore $100_8 + 1_8$, which works out to be 101_8 or 65_{10}. You can use the table to find the code for any character in this same way. In most cases, however, you will not need to do so. Although it is important to know that characters are represented internally using a numeric code, it is not generally useful to know what numeric value corresponds to a particular character. When you type the letter *A*, the hardware logic built into the keyboard automatically translates that character into the Unicode value 65 which is then sent to the computer. Similarly, when the computer sends the Unicode value 65 to the screen, the letter *A* appears.

Character constants

When you want to refer to a specific character in a Java program, the standard approach is to specify a **character constant**, which is written by enclosing the desired character in single quotation marks. For example, to indicate the Unicode representation for the letter *A,* all you have to write is `'A'`. The Java compiler knows that this notation means to use the Unicode value for the letter *A,* which happens to be 65. Similarly, you can indicate the space character by writing `' '` or the digit 9 by writing `'9'`. Note that the constant `'9'` refers to a *character* and should not be confused with the *integer* value 9. As an integer, the value `'9'` is the value for that character given in the Unicode table, which is 71_8 or 57.

Since Java represents characters internally as their integer codes in the Unicode table, you could in most cases replace the character constant `'A'` with the integer 65. The program would work in exactly the same way but would be much harder to read. You need to keep in mind that some other programmer will eventually come along and have to make sense out of what you've written. Unless that programmer has memorized

> Avoid using integer constants to refer to ASCII characters within a program. All character constants should be indicated by enclosing the character in single quotation marks, as in `'A'` or `'*'`.

the Unicode table, seeing the integer 65 written as part of the program won't immediately

conjure up an image of the letter *A*. On the other hand, the character constant `'A'` conveys that meaning directly.

This text includes Figure 9-1 to give you a more concrete understanding of how characters are represented inside the machine. As soon as you have that idea in mind, you should forget about the specific character codes and concentrate instead only on the character itself.

Important properties of the Unicode representation

Even though it is important not to think about specific character codes, the following two structural properties of the Unicode table are worth remembering:

1. The codes for the characters representing the digits 0 through 9 are consecutive. Even though you do not need to know exactly what code corresponds to the digit character `'0'`, you know that the code for the digit `'1'` is the next larger integer. Similarly, if you add 9 to the code for `'0'`, you get the code for the character `'9'`.

2. The letters in the alphabet are divided into two separate ranges: one for the uppercase letters *(A–Z)*, and one for the lowercase letters *(a–z)*. Within each range, however, the Unicode values are consecutive, so that you can count through the letters one at a time in order.

Each of these properties will be useful in programs at various points later in this text.

Special characters

Most of the characters in Figure 9-1 are the familiar ones that can be displayed on the screen. These characters are called **printing characters**. The Unicode table, however, also includes various **special characters**, which are used to perform a particular action, which are represented by a backslash followed by a character or a sequence of digits. The combination of the backslash and the characters that follow it is called an **escape sequence**. Figure 9-2 lists the predefined escape sequences.

FIGURE 9-2	Escape sequences used in character and string constants
Sequence	**Interpretation**
`\b`	Backspace
`\f`	Formfeed (starts a new page)
`\n`	Newline (moves to the beginning of the next line)
`\r`	Return (returns to the beginning of the line without advancing)
`\t`	Tab (moves horizontally to the next tab stop)
`\0`	Null character (the character whose Unicode value is 0)
`\\`	The character `\` itself
`\'`	The character `'` (requires the backslash only in character constants)
`\"`	The character `"` (requires the backslash only in string constants)
`\`*ddd*	The character whose Unicode value is the octal number *ddd*
`\u`*xxxx*	The character whose Unicode value is the hexadecimal number *xxxx*

You can include special characters in character constants by writing the escape sequence as part of the constant. Although each escape sequence consists of several characters, each sequence is translated into a single Unicode value inside the machine. The codes for the special characters are included in Figure 9-1.

When the compiler sees the backslash character, it expects it to be the first character in an escape sequence. If you want to represent the backslash character itself, you therefore have to use two consecutive backslashes inside single quotation marks like this: `'\\'`. Similarly, the single quotation mark, when used as a character constant, must also be preceded by a backslash: `'\''`.

Special characters can also be used in string constants. The fact that a double quotation mark is used to indicate the end of a string means that the double quotation mark must be marked as a special character if it is part of a string. For example, if you write a program containing the line

```
println("\"Bother,\" said Pooh.");
```

the output is
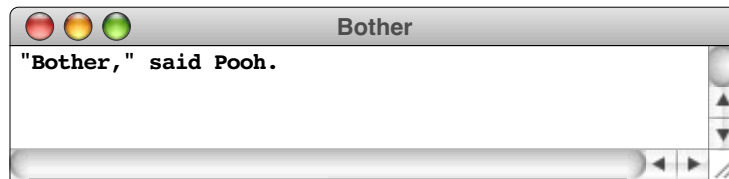
Many of the special characters in Unicode do not have explicit names and are instead represented in programs by specifying their internal numeric codes. Those characters can be specified in octal using the format *ddd* or in hexadecimal as \\u*xxxx*.

Character arithmetic

In Java, character values can be manipulated as if they were integers. You don't need a type cast to convert a character to its integer equivalent, although specifying such casts can sometimes make your programs easier to read. When character values are used in expressions, the result is defined according to the internal Unicode values. For example, the character `'A'`, which is represented internally using the Unicode value 65, is treated as the integer 65 whenever it is used in an arithmetic context. Conversions from integers to characters use the same strategy of relying on the internal Unicode values. In Java, however, you do need a type cast to convert an integer into a character because the range of type `char` is smaller than the range of type `int`. Java's designers were careful to require explicit type casts in cases where information might be lost in the conversion.

You can use the convertibility of integers and characters to write a `randomLetter` method that returns a randomly chosen uppercase letter. The simplest implementation is

```
private char randomLetter() {
    return (char) rgen.nextInt('A', 'Z');
}
```

where `rgen` is an instance variable containing a `RandomGenerator`, as described in Chapter 6. This implementation, however, may end up confusing its readers who expect the call to `rgen.nextInt` to take integers as parameters rather than characters. For the

benefit of those readers, it is probably better to make the conversions explicit by
introducing type casts, as follows:

```
private char randomLetter() {
    return (char) rgen.nextInt((int) 'A', (int) 'Z');
}
```

Even though it is legal to apply any arithmetic operation to values of type `char`, not all
operations are meaningful in that domain. For example, it is legal to multiply `'A'` by `'B'`
as part of a program. To determine the result, the computer takes the internal codes, 65
and 66, and multiplies them to get 4290. The problem is that this integer means nothing
in the character world. Only a few of the arithmetic operations are likely to be useful
when applied to characters. The operations that generally make sense are:

- *Adding an integer to a character*. If `c` is a character and `n` is an integer, the expression
  `c + n` represents the character code that comes `n` characters after `c` in the coding
  sequence. For example, the expression `'0' + n` computes the character code of the nth
  digit, if `n` is between 0 and 9. Thus `'0' + 5` computes the character code for `'5'`.
  Similarly, the expression `'A' + n – 1` computes the character code of the nth letter in the
  alphabet, assuming that `n` is between 1 and 26. The result of this operation is logically
  a `char`, and I would not use a type cast if that were the intended result.
- *Subtracting an integer from a character*. The expression `c – n` represents the code of
  the character that comes `n` characters before `c` in the coding sequence. For example,
  the expression `'Z' – 2` computes the character code for `'X'`. The result of this
  operation is logically a `char`.
- *Subtracting one character from another*. If `c1` and `c2` are both characters, the
  expression `c1 – c2` represents the distance between those characters in coding
  sequence. For example, if you look back to Figure 9-1 and compute the Unicode
  values of each character, you can determine that `'a' – 'A'` is 32. More importantly,
  the distance between a lowercase character and its uppercase counterpart is constant,
  so that `'z' – 'Z'` is also 32. The result of this operation is logically an `int`.
- *Comparing two characters against each other*. Comparing two character values using
  any of the relational operators is a common operation, often used to determine
  alphabetical ordering. For example, the expression `c1 < c2` is `true` if `c1` comes before
  `c2` in the Unicode table.

To see how these operations apply to practical problems, consider how the computer
executes a method like `readInt`. When a user types a number, such as 102, the computer
receives the individual keystrokes as characters and must therefore work with the input
values `'1'`, `'0'`, and `'2'`. Because the `readInt` method must return an integer, it needs
to translate the character into the integers they represent. To do so, `readInt` takes
advantage of the fact that the digits are consecutive in the Unicode sequence.

As an illustration, suppose that `readInt` has just read a character from the keyboard
and stored it in the variable `ch`. It can convert the character to its numeric form by
evaluating the expression

```
ch – '0'
```

Assuming that `ch` contains a digit character, the difference between its Unicode value and
the Unicode value for the digit `'0'` must correspond to the decimal value of that digit.
Suppose, for example, that the variable `ch` contains the character `'9'`. If you consult the
Unicode table, you can determine that the character `'9'` has the internal code 57. The

digit `'0'` has the Unicode value 48, and 57 – 48 is 9. The key point is that the method makes no assumption that `'0'` has the Unicode value 48, which means that the same method would work even if Java someday decided to use a different character set. The only assumption is that the codes for the digits form a consecutive sequence.

But how can **readInt** determine whether the character **ch** is in fact a digit? Once again, it can take advantage of the fact that the digits are consecutive in the Unicode table. The statement

```
if (ch >= '0' && ch <= '9') . . .
```

distinguishes the digit characters from the rest of the Unicode set. Similarly, the statement

```
if (ch >= 'A' && ch <= 'Z') . . .
```

identifies the uppercase letters, and

```
if (ch >= 'a' && ch <= 'z') . . .
```

identifies the lowercase letters.

Useful methods in the **Character** class

As it happens, however, you won't usually encounter the **if** statements used at the end of the preceding section in a typical Java program. The operations for checking whether a character is a digit or a letter are so common that the designers of Java made them methods in the **Character** class, which is defined in the package **java.lang** and is therefore available in any Java program without an **import** statement.

The **Character** class declares several useful methods for manipulating character values, of which the most important are shown in Figure 9-3. Like the methods in the

---

FIGURE 9-3   **Useful static methods in the Character class**

`static boolean isDigit(char ch)`
Determines if the specified character is a digit.
`static boolean isJavaIdentifierPart(char ch)`
Determines if the specified character may be part of a Java identifier.
`static boolean isJavaIdentifierStart(char ch)`
Determines if the specified character is permissible as the first character in a Java identifier.
`static boolean isLetter(char ch)`
Determines if the specified character is a letter.
`static boolean isLetterOrDigit(char ch)`
Determines if the specified character is a letter or digit.
`static boolean isLowerCase(char ch)`
Determines if the specified character is a lowercase character.
`static boolean isUpperCase(char ch)`
Determines if the specified character is an uppercase character.
`static boolean isWhitespace(char ch)`
Determines if the specified character is white space according to Java.
`static char toLowerCase(char ch)`
Converts **ch** to its lowercase equivalent, if one exists. If not, **ch** is returned unchanged.
`static char toUpperCase(char ch)`
Converts **ch** to its uppercase equivalent, if one exists. If not, **ch** is returned unchanged.

Math class, these methods are declared to be static, which means that they don't operate on an object but instead act more like traditional functions in mathematics. For example, you can convert a lowercase character **ch** into its uppercase equivalent by writing

```
ch = Character.toUpperCase(ch);
```

Although **toLowerCase** and **toUpperCase** are already available in the **Character** class, you will be able to appreciate their operation more if you try to implement them from scratch. Once again, you can ignore the actual Unicode values involved and rely only on the fact that letters are contiguous. If **ch** contains a character code for an uppercase letter, you can convert it to its lowercase form by adding the constant difference in value that separates the uppercase and lowercase characters. Rather than write that difference as an explicit constant, however, the program is easier to read if you express it using character arithmetic as **'a' - 'A'**. Thus, you could implement the **toLowerCase** method as follows:

```
public static char toLowerCase(char ch) {
   if (ch >= 'A' && ch <= 'Z') {
      return ch + 'a' - 'A';
   } else {
      return ch;
   }
}
```

The method **toUpperCase** has a similar implementation. Note that this method is declared here to be **public** rather than the traditional **private**. The corresponding method in the **Character** class must certainly be public, or you would not be able to call it. Similarly, the method is declared to be **static** to match the implementation in the **Character** class.

Even though the methods defined in the **Character** class are easy to implement, it is good programming practice to use the library methods instead of writing your own. There are four principal reasons for doing so.

1. Because the methods from the **Character** class are standard, programs you write will be easier for other programmers to read. Assuming those programmers are at all experienced in Java, they will recognize the methods in that class and know exactly what they mean.
2. It is easier to rely on library methods for correctness than on your own. Because the Java libraries are used by millions of client programmers, there is considerable pressure on the implementors to get the methods right. If you rewrite library methods yourself, the chance of introducing a bug is much larger.
3. The code you write yourself will not take advantage of the international applicability of Unicode. The do-it-yourself implementation of **toLowerCase** shown earlier assumes that the characters are in the Roman alphabet, which is no longer a reasonable assumption. Other alphabets also have uppercase and lowercase characters, and the implementation in the **Character** class knows how to convert those as well.
4. The implementations of methods in the library packages are typically more efficient than those you would write yourself. How these more efficient implementations work is beyond the scope of this chapter, but the important point is that you can take advantage of that added efficiency by using the library forms.

Control statements involving characters

Because **char** is a scalar type, you can use it in all the statement forms in which integers appear. For example, if **ch** is declared to be of type **char**, you can use the following **for** header line to execute a loop 26 times, once for each uppercase letter in the alphabet:

```
for (char ch = 'A'; ch <= 'Z'; ch++)
```

Similarly, you can use a character as the control expression in a **switch** statement. For example, the following predicate method returns **true** if its argument is a vowel in the English language:

```
private boolean isEnglishVowel(char ch) {
   switch (Character.toLowerCase(ch)) {
      case 'a': case 'e': case 'i': case 'o': case 'u':
         return true;
      default:
         return false;
   }
}
```

Note that the implementation uses the **toLowerCase** method to recognize vowels in both their uppercase and lowercase forms.

9.3 Strings as an abstract idea

The real power of using characters comes from the fact that you can string them together, one after another, to form a sequence of characters called a **string,** which is represented in Java by a class in the always-imported **java.lang** package called **String**. Because English explanations of strings get much harder to read if every occurrence of the noun that means a sequence of characters is written in its Java form, I will usually talk about strings in an abstract sense and just call them strings with no uppercase **S** and no special code font. I leave it to you to remember that strings—as an abstract idea—are represented in Java by a class named **String**. As noted in the introduction to this chapter, you have been using strings since the very first program in the book to display messages. As you will discover, however, there is a lot more you need to learn about strings in order to unlock the enormous power they bring to programming. To understand strings in their entirety, you must consider them from several different perspectives at differing levels of detail.

As you have found many times in this text, you can approach programming from both a reductionistic and a holistic perspective. When you concern yourself with the internal details of data representation, you are taking the reductionistic view. From this perspective, your job is to understand how characters are stored in the computer's memory, how a sequence of those characters can be stored to form a string, and how, for example, a 200-character string can fit inside the same variable that holds a 2-character string. These are all interesting questions, and you will discover the answers in due course. When you consider strings from the holistic perspective, however, your job is to understand how to manipulate a string as a single logical unit. By focusing on the abstract behavior of strings, you can learn how to use them effectively without getting bogged down in details.

To a certain extent, Java forces a holistic view of strings because the **String** class provides you with very little access to the underlying representation. Even so, it is possible to contrast the reductionistic view in which you think about the individual characters with the more holistic view in which you don't. So far in this text, you have

been working entirely at the holistic level. The remainder of this chapter describes methods in the **String** class that operate at both the character and whole-string level.

The notion of an abstract type

The principal advantage of the **String** class—and of object-oriented class definitions in general—is that the definition makes it possible to work with strings as an **abstract type** in which the fundamental operations are defined only by their *behavior* and not in terms of the underlying *representation*. In Java, abstract types correspond to class definitions that specify the operations that can be performed on objects of that type. The legal operations for a particular abstract type are called its **primitive operations** and are defined as public methods. Details of how those operations work and how the information is represented internally are hidden away in the implementation of the class. Whenever a client wants to manipulate values of an abstract type, the client must use the methods provided by the class.

In the context of strings, what are the primitive operations that you might want to perform? To begin with, you already know how to specify a string constant in a program, to combine strings using concatenation, and to perform string input and output using the **println** and **readLine** methods.

What else might you want to do? When working with strings, you might, for example, want to perform any of the following operations:

- Find out how long a string is
- Select the first character—or, more generally, the ith character—within a string
- Extract a piece of a string to form a shorter one
- Determine whether two strings are equal
- Compare two strings to see which comes first in alphabetical order
- Determine whether a string contains a particular character or set of characters

There are other operations you might consider, but this list offers an interesting and useful start. Each of these operations is provided by a method in the **String** class, which gives you the tools you need to use strings without requiring you to comprehend the details of the underlying representation. The fact that you do not need to understand those details is the essence of data abstraction.

9.4 Using the methods in the **String** class

As noted in Chapter 6, the best way to learn about a class and its methods is to consult the javadoc documentation. If you do so, you will find that the **String** class implements lots of methods, most of which you will never use. The ones that you are most likely to encounter are shown in Figure 9-4 and are described in the sections that follow.

As you read through the descriptions of these methods, it is important to keep in mind that the **String** class is **immutable,** which simply means that none of its methods ever change its internal state. For many students, this behavior—which is actually quite useful—seems counterintuitive. After discovering that the list in Figure 9-4 includes a **toLowerCase** method, most students expect that the line

        str.toLowerCase()

FIGURE 9-4	Important methods in the String class

`int length()`
Returns the length of the string.
`char charAt(int index)`
Returns the character at the specified index.
`String concat(String s2)`
Concatenates **s2** to the end of the receiver, returning a new string with the receiver unchanged.
`String substring(int p1, int p2)`
Returns the substring beginning at **p1** and extending up to but not including **p2**.
`String substring(int p1)`
Returns the substring beginning at **p1** and extending up to the end of the string.
`String trim()`
Returns the substring formed by deleting any white space at the beginning or end of the string.
`boolean equals(String s2)`
Returns true if the string **s2** is equal to the receiver.
`boolean equalsIgnoreCase(String s2)`
Returns true if the string **s2** is equal to the receiver ignoring distinctions in case.
`int compareTo(String s2)`
Returns a number whose sign indicates how the strings compare in lexicographic order.
`int indexOf(char c)` *or* `indexOf(String s)`
Returns the index of the first occurrence of the character or string, or -1 if it does not appear.
`int indexOf(char c, int start)` *or* `indexOf(String s, int start)`
Like **indexOf** with one argument, but starts at the specified index position.
`boolean startsWith(String prefix)`
Returns true if the string starts with the specified prefix.
`boolean endsWith(String suffix)`
Returns true if the string ends with the specified suffix.
`string toUpperCase()`
Converts the string to uppercase.
`string toLowerCase()`
Converts the string to lowercase.

will convert the characters in the string **str** to lower case. As the bug symbol suggests, however, this expectation is incorrect. What the **toLowerCase** method in fact does is to *return* an entirely new string in which those conversions have been performed. Thus, to change the value stored in the string variable **str** so that all letters within it appear in lowercase, you need to use an assignment statement, such as

```
str = str.toLowerCase();
```

Determining the length of a string

When writing programs to manipulate strings, you often need to know how many characters a particular string contains. The total number of characters a string contains—counting all letters, digits, spaces, punctuation marks, and special characters—is called the **length** of the string.

Using the **String** class, you can obtain the length of a string **s** by calling the method **length**. For example, the first string you encountered in this book was

```
"hello, world"
```

This string has length 12—five characters in the word **hello**, five more in **world**, one comma, and one space. Thus, if you assigned that string to the variable message by writing

```
String message = "hello, world";
```

you could then compute its length by writing

```
message.length()
```

which would return the value 12. Note that the **String** class uses the object-oriented receiver notation. You have a **String** variable called **message**. To determine its length, the object-oriented paradigm asks you to send that string a message asking it to give you back its length.

The following **run** method for a **ConsoleProgram** reads in a single line of text from the user and reports its length:

```
public void run() {
    String line = readLine("Enter a string: ");
    println("That string has " + line.length() + " characters.");
}
```

Selecting characters from a string

In Java, positions within a string are numbered starting from 0. For example, the individual characters in the string **"hello, world"** are numbered as in the following diagram:

h	e	l	l	o	,		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10	11

The position number written underneath each character in the string is called its **index.**

To enable you to select a particular character in a string given its index, the **String** class provides a method called **charAt** that takes an integer representing the index and returns a character. For example, if the variable **str** contains the string **"hello, world"**, calling

```
str.charAt(0)
```

returns the character **'h'**. Similarly, calling **str.charAt(5)** returns **','**. Be sure to remember that Java numbers characters starting with 0, not 1. It is easy to forget this rule and assume that **str.charAt(5)** will return the fifth character in the string; **str.charAt(5)** returns the character at index position 5, which is the *sixth* character as you would number character positions in English.

Concatenation

Figure 9-4 shows that the String class implements a **concat** method, which takes an existing string as a receiver and a new string as an argument and connects them, end to end, with no intervening characters. Thus, if the variable **str** contains **"hello, world"**, you could generate a new string that included an exclamation point on the end by writing

```
str.concat("!")
```

It is critical to note, however, that this method call simply returns a new string value and does not change the value of the variable **str**. To do so using the **concat** method, you would have to write

```
str = str.concat("!");
```

In Java, no one actually uses the **concat** method because its function is built into Java in the form of the **+** operator. Thus, what you would be more likely to see is

```
str = str + "!";
```

or, even more compactly,

```
str += "!";
```

In either form, concatenation always combines the two strings with no intervening space. If you want to put a space between two words represented as string values, you have to perform an additional concatenation step. For example, if the variable **word1** contains **"hello"** and the variable **word2** contains **"world"**, you need to write

```
word1 + " " + word2
```

to get the string **"hello world"**. You could also write this expression using the **concat** method as

```
word1.concat(" ").concat(word2)
```

which begins to supply powerful evidence of the value of the **+** operator.

The **+** operator also has the wonderful property that it converts any operands that are not strings to their string representation, which is why it works to write statements like

```
println("The answer is " + answer);
```

This statement works no matter what type the variable **answer** is. Before Java concatenates the two values, it will convert **answer** to its string representation.

As a simple example of the use of concatenation, the following method returns a string consisting of a specified number of copies of the string passed as the second argument:

```
private String concatNCopies(int n, String str) {
   String result = "";
   for (int i = 0; i < n; i++) {
      result += str;
   }
   return result;
}
```

This method is particularly handy if you want to generate separators in console output. For example, the statement

```
println(concatNCopies(72, "-"));
```

prints out a line of 72 hyphens.

In a way, the implementation strategy used in **concatNCopies** is similar to that used in the **factorial** method presented in Chapter 5. In both cases, the method uses a local variable to keep track of the partially computed result during each cycle of a **for** loop. In the **concatNCopies** method, each cycle in the **for** loop concatenates the value of **str**

onto the end of the previous value of **result**. Because each cycle adds one copy of **str** to the end of **result**, the final value of **result** after **n** cycles must consist of **n** copies of that string.

In each of the two methods—**factorial** and **concatNCopies**—the initialization of the variable used to hold the result is worthy of some note. In the **factorial** method, the variable **product** is initialized to 1, so that multiplying it by each successive value of **i** properly keeps track of the result as the computation proceeds. In the case of **concatNCopies**, the corresponding statement initializes the string variable **result** so that it grows through concatenation. After the first cycle of the loop, the variable **result** must consist of one copy of the string **str**. Prior to the first cycle, therefore, **result** must contain zero copies of the string, which means it has no characters at all. The string with no characters at all is called the **empty string** and is written in Java using adjacent double quotes: **""**. Whenever you need to construct a new string by concatenating successive parts onto an existing string variable, you should initialize that variable to the empty string.

Extracting parts of a string

Concatenation makes longer strings from shorter pieces. You often need to do the reverse: separate a string into the shorter pieces it contains. A string that is part of a longer string is called a **substring**. The **String** class provides a method **substring(p1, p2)**, the effect of which is to extract the characters in **s** lying between positions **p1** up to but not including **p2**. Thus if **str** contains the string **"hello world"**, the method call

```
str.substring(1, 4)
```

returns the string **"ell"**. As you know, numbering in Java begins at 0, so the character at index position 1 is the character **'e'**.

The second argument in the **substring** method is optional. If it is missing, substring returns the characters starting at the index position specified by the first argument and continuing through the end of the string.

As an example of the use of **substring**, the method **secondHalf(s)** returns the substring consisting of the last half of the characters in **s**, including the middle character if the length of the string is odd:

```
private String secondHalf(String str) {
    return str.substring(str.length() / 2);
}
```

Comparing one string with another

At many times in your programming, you will need to check to see whether two strings have the same value. When you do, you are almost certain to code this test at least once in the following incorrect form:

```
if (s1 == s2) . . .
```

The problem here is that strings are objects in Java and the relational operators like **==** are defined in the conventional mathematical way only for the primitive Java types like **int**

and **char**. What makes this problem all the more insidious is that the **==** operator does something when applied to strings, but not what you'd initially expect. For objects, the **==** operator tests whether the two sides are the same object. What you need here is to test whether two different **String** objects have the same value.

To accomplish what you need, the **String** class defines a method called **equals**, which you can use to test two strings for equality. Like the other methods in the **String** class, **equals** is applied to a receiver object, which means that the syntax for an equality test looks like this:

> When comparing string values in Java, remember to use the **equals** and **compareTo** methods, and *not the relational operators*. The compiler will not detect this error, but the test will have unpredictable results.

```
if (s1.equals(s2)) . . .
```

There is also a method **equalsIgnoreCase** that checks whether two strings are equal independent of uppercase/lowercase distinctions.

You will sometimes also find it useful to determine how two strings relate to each other in alphabetical order. The **String** class provides the method **compareTo** for this purpose. The **compareTo** method is called in typical receiver-based fashion

```
s1.compareTo(s2)
```

and returns an integer whose sign indicates the relationship between the two strings, as follows:

- If **s1** precedes **s2** in alphabetical order, **compareTo** returns a negative integer.
- If **s1** follows **s2** in alphabetical order, **compareTo** returns a positive integer.
- If the two strings are exactly the same, **compareTo** returns 0.

Thus, if you want to determine whether **s1** comes before **s2** in alphabetical order, you need to write

```
if (s1.compareTo(s2) < 0) . . .
```

The "alphabetical order" computers use is different from the order that dictionaries use in certain respects. When **compareTo** compares two strings, it compares them using the numeric ordering imposed by the underlying character codes. This order is called **lexicographic order** and differs from traditional alphabetical order in several respects. For example, in an alphabetical index, you will find the entry for *aardvark* before the entry for *Achilles,* because traditional alphabetical ordering does not consider uppercase and lowercase letters separately. If the **compareTo** method is called with the arguments **"aardvark"** and **"Achilles"**, the method simply compares the Unicode values. In Unicode, the lowercase character **'a'** comes after an uppercase **'A'**. In lexicographic order, the string **"Achilles"** comes first. Thus the method call

```
"aardvark".compareTo("Achilles")
```

returns a positive integer.

When you call **compareTo**, it compares the strings starting with the first character in each. If those characters are different, **compareTo** considers how the two character values relate to each other in the Unicode sequence and returns an integer that indicates that

result. If the first characters match, **compareTo** goes on to look at the second characters, continuing this process until a difference is detected. If **compareTo** runs out of characters in one of the two strings, that string is automatically considered to precede the longer one, just as in traditional alphabetical ordering. For example,

```
"abc".compareTo("abcdefg")
```

returns a negative integer. Only if the two strings match all the way down the line and end at the same place does **compareTo** return the value 0.

Searching within a string

From time to time, you will find it useful to search a string to see whether it contains a particular character or substring. To do so, the **String** class provides a method called **indexOf**, which comes in several forms. The simplest form of the call is illustrated by the statement

```
int pos = str.indexOf(search);
```

where *search* is what you are looking for, which can be either a string or a character. When **indexOf** is called, the method searches through the string **str** looking for the first occurrence of the search value. If the search value is found, **indexOf** returns the index position at which the match begins. If the character does not appear before the end of the string, **indexOf** returns the value –1.

The **indexOf** method also takes an optional second argument that indicates the index position at which to start the search. The effect of both styles of the **indexOf** method is illustrated by the following examples, which assume that the variable **str** contains the string **"hello, world"**:

```
str.indexOf('o')      returns   4
str.indexOf('o', 5)   returns   8
str.indexOf('x')      returns  −1
```

As with string comparison, the methods for searching a string consider uppercase and lowercase characters to be different.

You can use **indexOf** to implement a method that generates an **acronym**, which is a new word formed by combining, in order, the initial letters of a series of words. For example, the word *scuba* is an acronym formed from the first letters in *self contained underwater breathing apparatus*. The method **acronym** takes a string composed of separate words and return its acronym. Thus, calling the method

```
acronym("self contained underwater breathing apparatus")
```

returns **"scuba"**.

Provided that the words are separated by a single space and that no extraneous characters appear, the implementation of **acronym** can simply take the very first letter and then go into a loop searching for each space. Whenever it finds one, it can concatenate the next character onto the end of the string variable used to hold the result. When no more spaces appear in the string, the acronym is complete. This strategy can be translated into a Java implementation as follows:

```
private String  acronym(String str) {
   String result = str.substring(0, 1);
   int pos = str.indexOf(' ');
   while (pos != -1) {
      result += str.substring(pos + 1, pos + 2);
      pos = str.indexOf(' ', pos + 1);
   }
   return result;
}
```

The Java **String** class also contains two useful methods for checking to see whether a string begins or ends with a particular substring. The **startsWith** method returns **true** if the string on which it is invoked begins with the string passed as an argument. Thus, the statements

```
String answer = readLine("Would you like to play a game? ");
if (answer.startsWith("y") || answer.startsWith("Y")) {
   . . . code to play the game  . . .
}
```

executes the body of the **if** statement only if the user enters an answer to the question that begins with an upper- or lowercase **Y**. The **String** class also contains a symmetric **endsWith** method that returns **true** if the string ends with the specified substring.

Case conversion

The **String** class includes two methods, **toUpperCase** and **toLowerCase**, that convert the case of any alphabetic characters to the indicated case. For example, if **str** contains the string **"hello, world"**, calling the method

```
str.toUpperCase()
```

returns the string **"HELLO, WORLD"** Note that any nonalphabetic characters appearing in the string—such as the comma, space, and period in this example—are unaffected.

Summary

With this chapter, you have begun the process of understanding how to work with *text data*. In Java, the most common form of text data is a *string,* which is an ordered collection of individual characters. Individual characters are represented using the data type **char**, which is one of the primitive types defined in Java. Characters are represented inside the hardware as numeric values using an encoding system called Unicode. Strings themselves are represented using the class **String**, which is defined formally as part of the **java.lang** package, but is in fact integrated directly into the language in many ways.

In this chapter, you learned how to manipulate strings through the methods provided by the **String** class, which allows you to focus on the abstract behavior of a string without having to be concerned about its concrete representation.

Important points raised in this chapter include:

- Types whose conceptual values are not numbers can usually be represented inside the computer by numbering the elements in the domain of the type and then using those numbers as codes for the original values. Types defined by counting off their elements are called *enumerated types*.

- Java makes it possible to define new enumerated types in two ways. The older strategy is simply to use **int** for such types and to define named constants for the specific values. The more modern strategy is to use the **enum** keyword introduced in Java 5.0; that strategy, however, is beyond the current scope of this text.

- Characters are represented internally as integers according to a predefined coding scheme called *Unicode,* which makes it possible to represent characters from a wide range of languages. Those values are represented in Java by the primitive type **char**.

- Character values can be manipulated using the standard operations of arithmetic. No type cast is required to convert a **char** to an **int**, but Java does require casts to convert in the opposite direction.

- The **Character** class contains several methods for classifying and changing the case of individual characters.

- The **String** class makes it possible to work with strings as an *abstract type*.

- The **String** class defines several methods for manipulating strings. These methods are summarized in Figure 9-4.

Review questions

1. In your own words, state the principle of enumeration.

2. What are the two options mentioned in this chapter for representing enumerated types in Java? Why does this text currently rely on the older strategy?

3. What is a *scalar type?*

4. How do you include a double quotation mark inside a string constant?

5. What does *ASCII* stand for?

6. What is the relationship between ASCII and Unicode?

7. By consulting Figure 9-1, determine the octal Unicode values of the characters **'$'**, **'@'**, **'\t'**, and **'x'**. Convert these values to their decimal equivalents.

8. Why is it useful to know that the digit characters are consecutive in the Unicode table?

9. What four arithmetic operations does this chapter argue make the most sense for characters?

10. What is the result of calling **Character.isDigit(5)**? What is the result of calling **Character.isDigit('5')**? Is it legal to call **Character.isDigit("5")**?

11. What is the result of calling **Character.toUpperCase('5')**?

12. What four reasons are given in the chapter for using the methods in the **Character** class in preference to writing those methods on your own?

13. True or false: It is legal to use character constants as **case** expressions within a **switch** statement.

14. What effect does the following statement have on the value of **str**?

```
str.trim()
```

15. What is the correct way to achieve the effect clearly intended by the expression in the preceding question?

16. What is meant by the term *immutable?*

17. What is the result of each of the following expressions?
    a) `"ABCDE".length()`
    b) `"".length()`
    c) `"\t".length()`
    d) `"ABC".charAt(2)`
    e) `"ABCDE".substring(0, 3)`
    f) `"ABCDE".substring(2)`
    g) `"ABCDE".indexOf("C")`
    h) `"ABCDE".indexOf('Z')`
    i) `"XYZZY".indexOf('Z', 3)`
    j) `"ABCDE".toLowerCase()`

18. What is the most important caution to keep in mind when comparing strings?

19. What is the result of each of the following expressions? (For calls to `compareTo`, simply indicate the sign of the result.)
    a) `"ABCDE".equals("abcde")`
    b) `"ABCDE".equalsIgnoreCase("abcde")`
    c) `"ABCDE".compareTo("ABCDE")`
    d) `"ABCDE".compareTo("ABC")`
    e) `"ABCDE".compareTo("abcde")`
    f) `"ABCDE".startsWith("a")`

Programming exercises

1. Implement the method `isEnglishConsonant(ch)`, which returns `true` if `ch` is a consonant in English: that is, any letter except one of the five vowels: `'a'`, `'e'`, `'i'`, `'o'`, and `'u'`. Like `isEnglishVowel`, your method should recognize consonants of both cases. Write a `ConsoleProgram` that displays all the uppercase consonants.

2. Write a method `randomWord` that returns a randomly constructed "word" consisting of randomly chosen letters. The number of letters in the word should also be chosen randomly by picking a number between the values of the named constants `MIN_LETTERS` and `MAX_LETTERS` . Write a `ConsoleProgram` that tests your method by displaying five random words.

3. Implement a method `capitalize(str)` that returns a string in which the initial character is capitalized (if it is a letter) and all other letters are converted so that they appear in lowercase form. Characters other than letters are not affected. For example, `capitalize("BOOLEAN")` and `capitalize("boolean")` should each return the string `"Boolean"`.

4. Write a method **dateString(day, month, year)** that returns a string consisting of the day of the month, a hyphen, the first three letters in the name of the month, another hyphen, and the last two digits of the year. For example, calling the method

        dateString(22, 11, 1963)

    should return the string **"22-Nov-63"**.

5. In most word games, each letter in a word is scored according to its point value, which is inversely proportional to its frequency in English words. In Scrabble™, the points are allocated as follows:

Points	Letters
1	A, E, I, L, N, O, R, S, T, U
2	D, G
3	B, C, M, P
4	F, H, V, W, Y
5	K
8	J, X
10	Q, Z

    For example, the Scrabble word **"FARM"** is worth 9 points: 4 for the *F*, 1 each for the *A* and the *R*, and 3 for the *M*. Write a **ConsoleProgram** that reads in words and prints out their score in Scrabble, not counting any of the other bonuses that occur in the game. You should ignore any characters other than uppercase letters in computing the score. In particular, lowercase letters are assumed to represent blank tiles, which can stand for any letter but which have a score of 0.

6. If the designers of the **String** class had not defined the version of **indexOf** that takes a string argument, you could implement it using the other methods available in the library. Without calling **indexOf** directly, implement a method **myIndexOf** that behaves in exactly the same way.

7. Write a method **regularPluralForm(word)** that returns the plural of **word** formed by following these standard English rules:

    a. If the word ends in *s, x, z, ch,* or *sh,* add *es* to the word.
    b. If the word ends in *y* and the *y* is preceded by a consonant, change the *y* to *ies*.
    c. In all other cases, add just an *s*.

    Write a test program and design a set of test cases to verify that your program works.

8. In English, the notion of a present action that is continuing into the future is expressed using the present progressive tense, which involves the addition of an *ing* suffix to the verb. For example, the sentence *I think* conveys a sense that one is capable of thinking; by contrast, the sentence *I am thinking* conveys the impression that one is actually in the process of thinking. The *ing* form of the verb is called the **present participle.**

    Unfortunately, creating the present participle is not always as simple as adding the *ing* ending. One common exception is words that end in a silent *e*, such as *cogitate*. In such cases, the *e* is usually dropped, so that the participle form becomes

cogitating. Another common exception involves words that end with a single consonant, which typically gets doubled in the participle form. For example, the verb *program* becomes *programming*.

Although there are many exceptions, you can construct a large fraction of the legal participle forms in English by applying the following rules:

a. If the word ends in an *e* preceded by a consonant, take the *e* away before adding the *ing* suffix. Thus, *move* should become *moving*. If the *e* is not preceded by a consonant, it should remain in place, so that *see* becomes *seeing*.

b. If the word ends in a consonant preceded by a vowel, insert an extra copy of that consonant before adding the *ing* suffix. Thus, *jam* should become *jamming*. If, however, there is more than one consonant at the end of the word, no such doubling takes place, so that *walk* becomes *walking*.

c. In all other circumstances, simply add the *ing* suffix.

Write a method **presentParticiple** that takes an English verb, which you may assume is entirely lowercase and at least two characters long, and forms the participle using these rules. Write a **ConsoleProgram** to test your method.

9. Like most other languages, English include two types of numbers: **cardinal numbers** (such as *one, two, three,* and *four*) that are used in counting, and **ordinal numbers** (such as *first, second, third,* and *fourth*) that are used to indicate a position in a sequence. In numeric form, ordinals are usually indicated by writing the digits in the number, followed by the last two letters of the English word that names the corresponding ordinal. Thus, the ordinal numbers *first, second, third,* and *fourth* often appear in print as *1st, 2nd, 3rd,* and *4th*.

The general rule for determining the suffix of an ordinal can be defined as follows:

> Numbers ending in the digit 1, 2, and 3, take the suffixes **"st"**, **"nd"**, and **"rd"**, respectively, <u>unless</u> the number ends with the two-digit combination 11, 12, or 13. Those numbers, and any numbers not ending with a 1, 2, or 3, take the suffix **"th"**.

Your task in this problem is to write a function **ordinalForm(n)** that takes an integer **n** and returns a string indicating the corresponding ordinal number. For example, your function should return the following values:

ordinalForm(1)	*returns the string*	**"1st"**
ordinalForm(2)	*returns the string*	**"2nd"**
ordinalForm(3)	*returns the string*	**"3rd"**
ordinalForm(10)	*returns the string*	**"10th"**
ordinalForm(11)	*returns the string*	**"11th"**
ordinalForm(12)	*returns the string*	**"12th"**
ordinalForm(21)	*returns the string*	**"21st"**
ordinalForm(42)	*returns the string*	**"42nd"**
ordinalForm(101)	*returns the string*	**"101st"**
ordinalForm(111)	*returns the string*	**"111th"**

10. One of the simplest types of codes used to make it harder for someone to read a message is a **letter-substitution cipher,** in which each letter in the original message is replaced by some different letter in the coded version of that message. A

particularly simple type of letter-substitution cipher is a **Caesar cipher**—so named because the Roman historian Suetonius records that Julius Caesar used such a cipher—in which each letter is replaced by its counterpart a fixed distance ahead in the alphabet. A Caesar cipher is *cyclic* in the sense that any operations take shift a letter beyond *Z* simply circle back to the beginning and start over again with *A*.

As an example, suppose that you wanted to encode a message by shifting every letter ahead four places. In that Caesar cipher, each *A* becomes an *E*, *B* becomes *F*, *Z* becomes *D* (because it cycles back to the beginning), and so on.

To solve this problem, you should first define a method

```
private String encodeString(String str, int shift)
```

that returns a new string formed by shifting every letter in **str** forward the number of letters indicated by **shift**, cycling back to the beginning of the alphabet if necessary. After you have implemented **encodeString**, write a **ConsoleProgram** that duplicates the examples shown in the following sample run:

```
CaeserCipher
This program encodes a message using a Caesar cipher.
Enter the number of character positions to shift: 13
Enter a message: This is a secret message.
Encoded message: Guvf vf n frperg zrffntr.
```
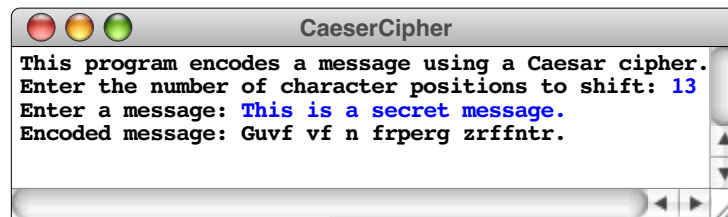
Note that the coding operation applies only to letters; any other character is included unchanged in the output. Moreover, the case of letters is unaffected: lowercase letters come out as lowercase, and uppercase letters come out as uppercase.

Write your program so that a negative value of **shift** means that letters are shifted toward the beginning of the alphabet instead of toward the end, as illustrated by the following sample run:

```
CaeserCipher
This program encodes a message using a Caesar cipher.
Enter the number of character positions to shift: –1
Enter a message: IBM 9000
Encoded message: HAL 9000
```
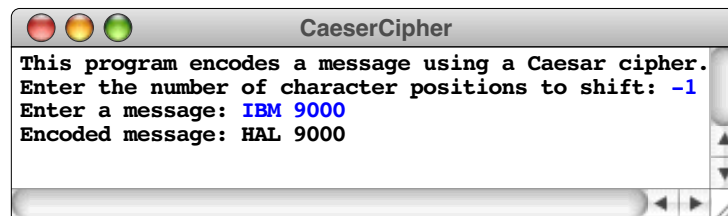
11. When large numbers are written out on paper, it is traditional—at least in the United States—to use commas to separate the digits into groups of three. For example, the number one million is usually written in the following form:

```
1,000,000
```

To make it easier for programmers to display numbers in this fashion, implement a method

```
private String addCommasToNumericString(String digits)
```

that takes a string of decimal digits representing a number and returns the string formed by inserting commas at every third position, starting on the right. For

example, if you were to execute the main program

```
public void run() {
   while (true) {
      String digits = readLine("Enter a numeric string: ");
      if (digits.length() == 0) break;
      println(addCommasToNumericString(digits));
   }
}
```

your implementation of the **addCommasToNumericString** method should be able to produce the following sample run:

```
●●●                          AddCommas
Enter a string of digits: 17
17
Enter a string of digits: 1001
1,001
Enter a string of digits: 12345678
12,345,678
Enter a string of digits: 999999999
999,999,999
Enter a string of digits:
```
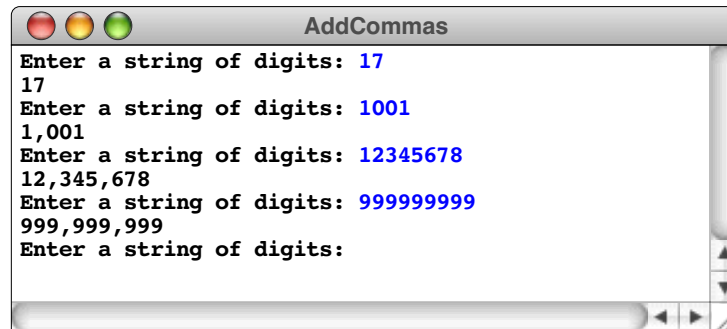
12. A **palindrome** is a word that reads identically backward and forward, such as *level* or *noon*. Write a predicate method **isPalindrome(str)** that returns **true** if the string **str** is a palindrome. In addition, design and write a test program that calls **isPalindrome** to demonstrate that it works. In writing the program, concentrate on how to solve the problem simply rather than how to you make your solution more efficient.

13. The concept of a palindrome introduced in exercise xxx is often extended to full sentences by ignoring punctuation and differences in the case of letters. For example, the sentence

   Madam, I'm Adam.

is a sentence palindrome, because if you only look at the letters and ignore any distinction between uppercase and lowercase letters, it reads identically backward and forward. Write a predicate method **isSentencePalindrome(str)** that returns **true** if the string **str** fits this definition of a sentence palindrome. For example, you should be able to use your method to write a main program capable of producing the following sample run:

```
●●●                    SentencePalindrome
This program checks for sentence palindromes.
Indicate the end of the input with a blank line.
Enter a string: Madam, I'm Adam.
That string is a palindrome.
Enter a string: A man, a plan, a canal: Panama!
That string is a palindrome.
Enter a string: Not a palindrome.
That string is not a palindrome.
Enter a string:
```
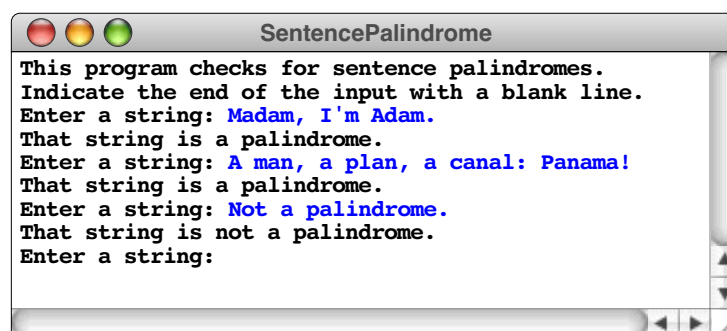
Chapter 10
Arrays and ArrayLists

Little boxes on a hillside, little boxes made of ticky-tacky
Little boxes, little boxes, little boxes all the same
There's a green one and a pink one and a blue one and a yellow one
And they're all made out of ticky-tacky and they all look just the same

— Malvina Reynolds, "Little Boxes," 1962

Dan Bricklin and Bob Frankston

In modern computing, one of the most visible applications of the array structure described in this chapter is the electronic spreadsheet, which uses a two-dimensional array to store tabular data. The first electronic spreadsheet was VisiCalc, which was released in 1979 by Software Arts Incorporated, a small startup company founded by MIT graduates Dan Bricklin and Bob Frankston. VisiCalc proved to be a popular application, leading many larger firms to develop competing products, including Lotus 1 2 3 and, more recently, Microsoft Excel.

Up to now, most of the programs in this book have worked with individual data items. Much of the power of computing, however, comes from the ability to work with collections of data. This chapter introduces the idea of an **array,** which is an ordered collection of values of the same type. Arrays are important in programming largely because such collections occur quite often in the real world. Whenever you want to represent a set of independent values in which it makes sense to think about those values as forming a sequence, arrays—or one of the Java library classes that builds on top of the standard array mechanism—are likely to be the appropriate strategy.

10.1 Introduction to arrays

An **array** is a collection of individual data values with two distinguishing characteristics:

1. *An array is ordered.* You must be able to count off the individual components of an array in order: here is the first, here is the second, and so on.
2. *An array is homogeneous.* Every value stored in an array must be of the same type. Thus, you can define an array of integers or an array of floating-point numbers but not an array in which the two types are mixed.

From an intuitive point of view, it is best to think of an array as a sequence of boxes, one box for each data value in the array. Each of the values in an array is called an **element.** For example, the following diagram represents an array with five elements:
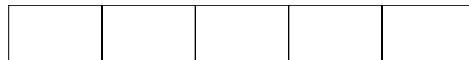
Even before you get around to specifying the values of the individual elements, any array has two fundamental properties that apply to the array as a whole:

• The **element type,** which is the type of value that may be stored in the elements of the array
• The **length,** which is the number of elements the array contains

In Java, these properties are specified at different times. You define the element type when you declare the array variable; you indicate the length when you create its initial value. In most cases, however, you can specify both properties in the same declaration, as described in the following section.

Array declaration

Like any other variable in Java, an array must be declared before it is used. The most common form for an array declaration is shown in the syntax box to the right. For example, the declaration

> `int[] intArray = new int[10];`

declares an array named **intArray** with 10 elements, each of which is of type **int**. You can represent this declaration graphically by drawing a row of 10 boxes and giving the entire collection the name **intArray**:
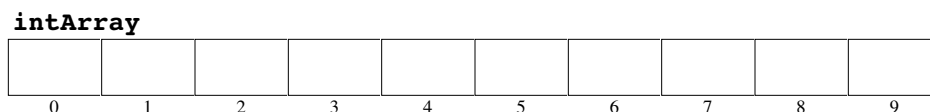
Typical syntax for array declarations:

> *type***[]** *name* **= new** *type***[***length***];**

where:
    type is the type of each element in the array
    name is the name of the variable being declared as an array
    length is the number of elements that are allocated as part of the array

As you will discover in the section entitled "Internal representation of arrays" later in this chapter, this picture has been simplified considerably from what is actually stored inside the machine. It is nonetheless perfectly useful as a model for arrays at a holistic level.

Each element in the array is identified by a numeric value called its **index**. In Java, the index numbers for an array always begin with 0 and run up to the length of the array minus one. Thus, in an array with 10 elements, the index numbers are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, as the preceding diagram shows.

Although it is possible to use integer values—such as the 10 in the preceding example—to specify the length of an array, it is much more common to use named constants for this purpose. Suppose, for example, that you have been asked to define an array capable of holding the scores for a sporting event, such as gymnastics or figure skating, in which scores are assigned by a panel of judges. Each judge rates the performance on a scale from 0 to 10, with 10 being the highest. Because a score may include a decimal fraction, as in 9.9, each element of the array must be of type **double**, which is the standard floating-point type. Because the number of judges might vary from application to application, you might want to declare it as a constant to make it easy to change. In this case, the declaration of an array called scores might look like this:

```
private static final int N_JUDGES = 5;

double[] scores = new double[N_JUDGES];
```

This declaration introduces a new array called **scores** with five elements, as shown in the following diagram:

scores

0	1	2	3	4
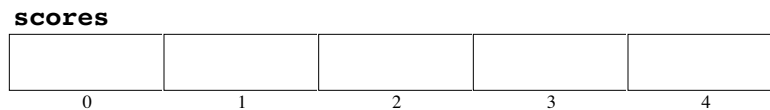
In Java, the value used to specify the length of an array need not be constant. If you wanted to make your sports-scoring program more general, you could read the number of judges from the user, as shown in the following code:

```
int nJudges = readInt("Enter number of judges: ");
double[] scores = new double[nJudges];
```

Array selection

To refer to a specific element within an array, you need to specify both the array name and the index corresponding to the position of that element within the array. The process of identifying a particular element within an array is called **selection** and is indicated in Java by writing the name of the array and following it with the index written in square brackets. The result is a **selection expression,** which has the following form:

*array***[***index***]**

Within a program, a selection expression acts just like a simple variable. You can use it in an expression, and, in particular, you can assign a value to it. Thus, if the first judge (judge #0, since Java counts array elements beginning at zero) awarded the contestant a score of 9.2, you could store that score in the array by writing the assignment statement

```
scores[0] = 9.2;
```

The effect of this assignment can be diagrammed as follows:

scores

9.2				
0	1	2	3	4

You could then go ahead and assign scores for each of the other four judges using, for example, the statements

```
scores[1] = 9.9;
scores[2] = 9.7;
scores[3] = 9.0;
scores[4] = 9.6;
```

Executing these statements results in the following picture:

scores

9.2	9.9	9.7	9.0	9.6
0	1	2	3	4

In working with arrays, it is essential to understand the distinction between the *index* of an array element and the *value* of that element. For instance, the first box in the array has index 0, and its value is 9.2. It is also important to remember that you can change the values in an array but never the index numbers.

The real power of array selection comes from the fact that the index value need not be constant, but can be any expression that evaluates to an integer or any other scalar type. In many cases, the selection expression is the index variable of a **for** loop, which makes it easy to perform an operation on each element of the array in turn. For example, you can set each element in the **scores** array to 0.0 with the following statement:

```
for (int i = 0; i < nJudges; i++) {
    scores[i] = 0.0;
}
```

Example of a simple array

The program **GymnasticsJudge** given in Figure 10-1 provides a simple example of array manipulation. This program asks the user to enter the score for each judge and then displays the average score.

Running the **GymnasticsJudge** program with the data used in the examples in the preceding section produces the following sample run:

```
         GymnasticsJudge
Enter number of judges: 5
Enter score for judge 0: 9.2
Enter score for judge 1: 9.9
Enter score for judge 2: 9.7
Enter score for judge 3: 9.0
Enter score for judge 4: 9.6
The average score is 9.48
```
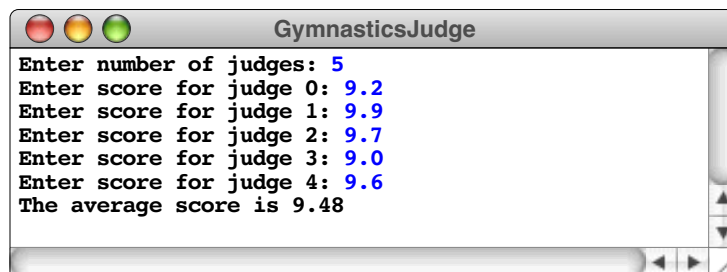
FIGURE 10-1   **Program to average an array of gymnastics scores**

```
/*
 * File: GymnasticsJudge.java
 * --------------------------
 * This file reads in an array of scores and computes the
 * average.
 */

import acm.program.*;

public class GymnasticsJudge extends ConsoleProgram {

/** Runs the program */
   public void run() {
      int nJudges = readInt("Enter number of judges: ");
      double[] scores = new double[nJudges];
      for (int i = 0; i < nJudges; i++) {
         scores[i] = readDouble("Enter score for judge " + i + ": ");
      }
      double total = 0;
      for (int i = 0; i < nJudges; i++) {
         total += scores[i];
      }
      double averageScore = total / nJudges;
      println("The average score is " + averageScore);
   }

}
```

Changing the index range

In Java, the first element in every array is at index position 0. In many applications, however, this design may cause confusion for the user. To a nontechnical person using the **GymnasticsJudge** program, the fact that it asks about judge 0 is likely to be disconcerting. In the real world, we tend to number elements in a list beginning at 1. It would therefore be more natural for the program to ask the user to enter the scores for judges numbered from 1 to 5 instead.

There are two standard approaches for solving this problem:

1. Use the indices 0 to 4 internally, but add 1 to each index value when requesting data from the user or displaying output data. If you adopt this approach, the only part of the **GymnasticsJudge** program that needs to change is the **readDouble** statement that reads in each input value, which becomes

   ```
   scores[i] = readDouble("Enter score for judge " + (i + 1) + ": ");
   ```

2. Declare the array with an extra element so that its indices run from 0 to 5, and then ignore element 0 entirely. Using this approach, the **run** method becomes

```
public void run() {
   int nJudges = readInt("Enter number of judges: ");
   double[] scores = new double[nJudges + 1];
   for (int i = 1; i <= nJudges; i++) {
      scores[i] = readDouble("Enter score for judge " + i + ": ");
   }
   double total = 0;
   for (int i = 1; i <= nJudges; i++) {
      total += scores[i];
   }
   double averageScore = total / nJudges;
   println("The average score is " + averageScore);
}
```

Note that the declaration of the **scores** array uses the expression **nJudges + 1** to specify the array size.

The advantage of the first approach is that the internal array indices still begin with 0, which often makes it easier to use existing methods that depend on that assumption. The disadvantage is that the program requires two different sets of indices: an external set for the user and an internal set for the programmer. Even though the user sees a consistent and familiar index pattern, having to think about both sets of indices can complicate the programming process. The advantage of the second approach is that the internal indices match the user's indices.

10.2 Internal representation of arrays

Although the **GymnasticsJudge** program in Figure 10-1 is sufficient to solve a specific problem, it is too simple to illustrate several of the most important issues that arise when using arrays. The most significant weakness of the **GymnasticsJudge** example is that it is written as a single method using none of the techniques of decomposition that you have learned in the earlier chapters. In order to write more sophisticated programs that use arrays, you will need to learn how to pass an array as an argument from one method to another.

Before you can appreciate the details of how array data can be communicated between two methods, it is essential to understand how arrays are represented in memory. In Java, passing an array—or any object for that matter as you saw in Chapter 7—as an argument to a method does not initially appear to be the same as passing a simple variable. If you understand how arrays are represented inside the computer, Java's handling of arrays as arguments makes sense. If you don't understand the internal representation, Java's approach will seem completely baffling.

In Chapter 7, you had the opportunity to learn a little bit about the way memory works. In particular, you learned that objects are stored in a region of memory called the *heap* and that each object is identified by its *address* in memory. In Java, all arrays are represented internally as objects and are stored in exactly the same way. The internal representation of an array value therefore consists of some standard information common to all objects followed by the memory needed to store the elements of the array in consecutive locations within the heap.

To illustrate this process, consider what happens when you issue the local variable declaration

```
double[] scores = new double[N_JUDGES];
```

where **N_JUDGES** is defined to be the constant 5. Although you can usually think of this declaration pattern as a single operation, it actually consists of two distinct parts. The declaration to the left of the equal sign introduces a variable named **scores**, which, like all local variables, is stored on the stack. The stack, however, does not include the space for the actual elements. The array of **double**s is created by the initializer that appears to the right of the equal sign. The expression

    new double[N_JUDGES]

allocates an array object in the heap that contains information common to all objects, the length of the array (in this case, five), and enough space for five values of type **double**. The five values themselves consume 40 bytes of memory, which you can calculate as follows:

    8 bytes/element × 5 elements = 40 bytes

The **scores** variable itself is large enough to store only the address of the array object in the heap. The internal picture of memory therefore looks like the rather complicated diagram shown in Figure 10-2. This picture contains far more information than necessary if all you are concerned with is having a mental image of the collection. For that of level of abstraction, a simple box diagram is sufficient. The more accurate reductionistic model, however, is essential to understanding how array values are assigned or passed as parameters, as discussed in the section that follows.

10.3 Passing arrays as parameters

As you know from Chapter 5, the key to writing a large program is breaking it down into many methods, each of which is small enough for you to comprehend it as a unit. The individual methods communicate by passing parameters from one method to another. If a large program involves arrays, decomposing that program will often require methods to pass entire arrays as parameters. In Java, the operation of passing an array as a parameter is closely tied to the internal representation of arrays in memory and can therefore seem mysterious. Having become acquainted with that internal representation, you are now ready to learn how array parameters work and how to use them effectively.

FIGURE 10-2   **Memory layout of the scores array**

Array storage in the heap   Local variable storage on the stack
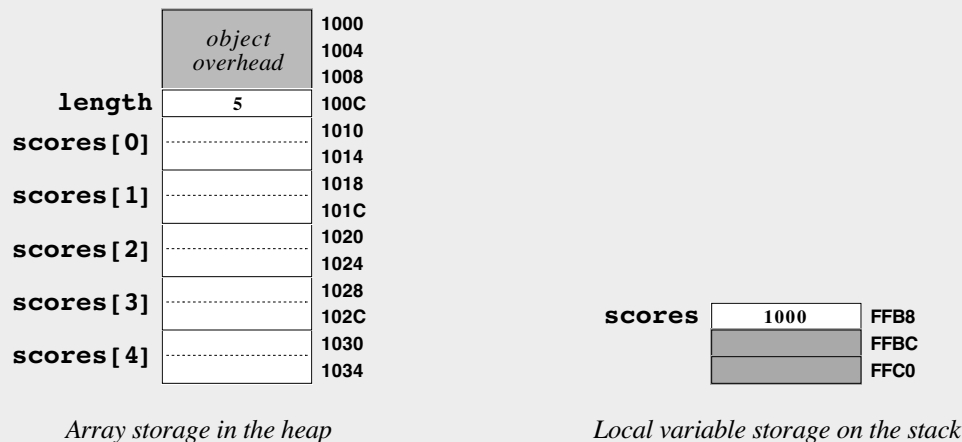
The issues involved in using arrays as parameters are best illustrated in the context of a simple example. Suppose that you have been asked to write a program that performs these steps:

1. Reads in a list of five integers
2. Reverses the elements in that list
3. Displays the list in reverse order

The following sample run illustrates the operation of the program:

```
●●●                       ReverseIntArray
This program reverses the elements in an array.
 ? 10
 ? 20
 ? 30
 ? 40
 ? 50
50
40
30
20
10
```
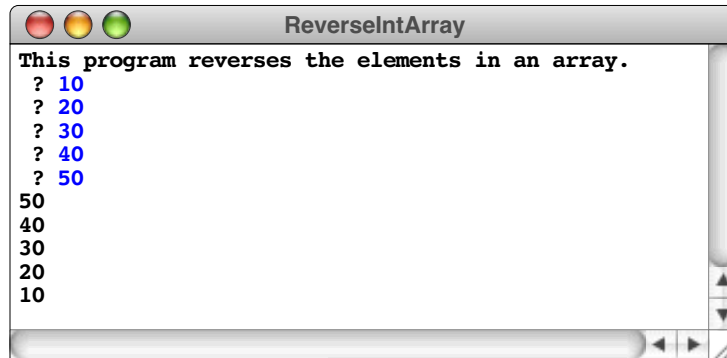
To illustrate the process of passing arrays as parameters, it is important to decompose this program into three methods that correspond to the three phases of the program operation: reading the input values and storing them in an array, reversing the elements of the array, and displaying the results. Using this decomposition, the **run** method for the program has the following structure:

```java
private static final int N_VALUES = 5;

public void run() {
    int[] array = new int[N_VALUES];
    println("This program reverses the elements in an array.");
    readArray(array);
    reverseArray(array);
    printArray(array);
}
```

This decomposition makes intuitive sense but works correctly only if Java allows methods to change the values of array elements that are passed to those methods as arguments. If you pass a primitive value like an integer, Java copies the value, which means that it is impossible for a method to change it. As you learned in Chapter 7, however, passing an object as parameter to a method means that only the reference is copied, so that the method effectively shares the object with the caller. The important question that arises in the suggested decomposition of the **ReverseArray** program is therefore whether arrays are treated as if they are primitive values or as if they are objects. The answer is that Java defines all arrays as objects, which in turn means that the elements of an array are shared between the calling method and the method being called.

Before going a bit more deeply into the details of how Java treats array parameters, it helps to expand a few of the methods in the proposed decomposition of the **ReverseArray** program. The simplest method is **printArray**, which is implemented like this:

```
    private void printArray(int[] array) {
       for (int i = 0; i < array.length; i++) {
          println(array[i]);
       }
    }
```

The declaration of the array parameter looks exactly like the definition of an array variable, but without the expression that specifies the initial value. That value is supplied by the caller, so the only thing the method needs to know is the fact that the parameter is an array of integers. The body of the loop is entirely straightforward except for the expression used for the **for** loop bound, which is written as

```
    array.length
```

Every array in Java has a **length** field, which tells you how many elements are in the array. The code for **printArray**, therefore, simply iterates over the elements in **array**, calling **println** on each one.

In the case of **printArray**, it doesn't matter how Java implements the process of passing an array to a method. If Java were to copy the entire array, the result would be just what you expect because **printArray** is merely looking up the values in the array without attempting to change them. The situation is different for **readArray**, which has to assign new values to the elements. The code, however, is very closely parallel to that of **printArray**:

```
    private void readArray(int[] array) {
       for (int i = 0; i < array.length; i++) {
          array[i] = readInt(" ? ");
       }
    }
```

Here, correct execution of the **readArray** method depends on being able to store new values in the array elements. If you think back to the diagram in Figure 10-2, you will see that the value being passed as the parameter **array** is merely the address of the array in the heap and does not contain the array elements themselves. As is true for any object passed as a parameter, this address gets copied into the corresponding parameter variable inside **readArray**, but continues to refer to the same object in the heap.

Implementing **reverseArray**

The one piece of the **ReverseArray** program that you have not yet seen is the **reverseArray** method itself. The basic algorithm is simple: to reverse an array, you need to exchange the first element with the last one, the second element with the next-to-last one, and so on until all the elements have been exchanged. Because arrays are numbered beginning at 0, the last element in an array of **n** items is at index **n – 1**, the next-to-last element is at index **n – 2**, and so forth. In fact, given any integer index **i**, the array element that occurs **i** elements from the end is always at index position

```
    n – i – 1
```

Thus, in order to reverse the elements of **array**, all you need to do is swap the values in **array[i]** and **array[n – i – 1]** for each index value **i** from the beginning of the array up to the center, which falls at index position **n / 2**. As soon as you reach the center, the elements in the second half of the array will already have their correct values because

each cycle of the **for** loop correctly repositions two array elements—one in each half. In pseudocode, the implementation of **reverseArray** is therefore

```
private void reverseArray(int[] array) {
    for (int i = 0; i < array.length / 2; i++) {
        Swap the values in array[i] and array[n – i – 1]
    }
}
```

The operation of swapping two values in an array is useful even beyond the bounds of this example. For this reason, it makes sense to define that operation as a separate method and to replace the remaining pseudocode in **reverseArray** with a single method call.

Although it is tempting to try, the method call that exchanges the two elements cannot be written as

```
swapElements(array[i], array[n – i – 1]);
```

In this call, the arguments to **swapElements** are individual array elements. Array elements act like simple variables and are therefore copied to the corresponding formal parameters. The **swapElements** method could easily interchange the local copies of these values but could not make permanent assignments to the calling arguments.

To avoid this problem, you can simply pass the entire array to the method that performs the swap operation, along with the two indices that indicate the positions that should be exchanged. For example, the call

```
swapElements(array, i, n – i – 1);
```

exchanges the elements at index positions **i** and **n – i – 1** of **array**, which is precisely what you need to replace the pseudocode in **reverseArray**.

Implementing **swapElements**

Implementing **swapElements** is a little more complicated than it at first appears. You cannot simply assign one element to another because the original value of the destination would then be lost. The easiest way to handle the problem is to use a local variable to hold one of the values temporarily. If you hold onto the value of one of the elements, you are then free to assign the other value directly, after which you can copy the first value from the temporary variable.

Suppose you want to exchange the values in **array** at positions 0 and 4. The strategy requires three separate steps.

1. Store the value in **array[0]** in the temporary variable, as illustrated in the following diagram:
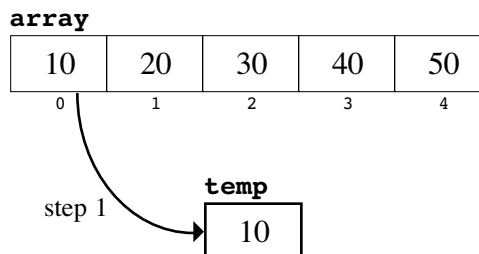
2. Copy the value from **array[4]** into **array[0]**, leaving the following configuration:
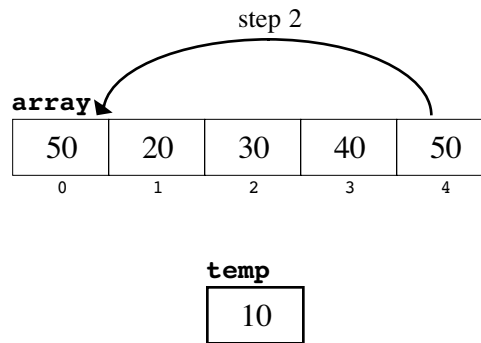
Because the old value of **array[0]** has previously been stored in **temp**, no information is lost.

3. Assign the value in **temp** to **array[4]**, as shown in this diagram:
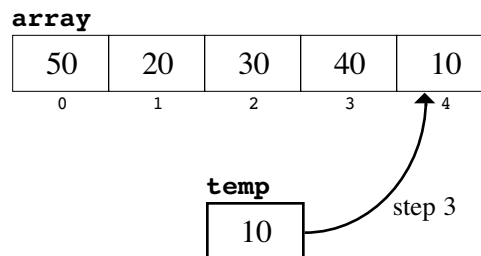
This three-step strategy is used as the basis for the following implementation of **swapElements**:

```
private void swapElements(int[] array, int p1, int p2) {
    int temp = array[p1];
    array[p1] = array[p2];
    array[p2] = temp;
}
```

This method fills in the last missing piece of the **ReverseArray** program, which appears in its complete form in Figure 10-3.

10.4 The **ArrayList** class

The **ReverseArray** program developed in the preceding section is not a very practical example, in part because the number of values is fixed by the constant **N_VALUES** in the program. When you write a program that works with arrays, you often do not know in advance how many elements there will be. The **ReverseArray** program would certainly be more versatile if the user were able to enter some kind of sentinel to indicate the end of the array. Adopting that strategy, however, requires a reformulation of the **ReverseArray** code because there is no longer any way to determine how large the array should be at the beginning.

Although it is possible to solve this problem without introducing any new classes, doing so would be much more complicated than most Java programmers would endure. In the primitive form in which they are implemented in Java, arrays lack many of the operations that you would like to apply to a list of values. In particular, you can't add

FIGURE 10-3   **Program to reverse an integer array**

```java
/*
 * File: ReverseArray.java
 * -----------------------
 * This file reads in an array of five integers and then displays
 * those elements in reverse order.
 */

import acm.program.*;

public class ReverseArray extends ConsoleProgram {

   private static final int N_VALUES = 5;

/** Runs the program */
   public void run() {
      int[] array = new int[N_VALUES];
      println("This program reverses the elements in an array.");
      readArray(array);
      reverseArray(array);
      printArray(array);
   }

/** Reads the data into the array */
   private void readArray(int[] array) {
      for (int i = 0; i < array.length; i++) {
         array[i] = readInt(" ? ");
      }
   }

/** Prints out the data from the array, one per line */
   private void printArray(int[] array) {
      for (int i = 0; i < array.length; i++) {
         println(array[i]);
      }
   }

/** Reverses the data in the array */
   private void reverseArray(int[] array) {
      for (int i = 0; i < array.length / 2; i++) {
         swapElements(array, i, array.length - i - 1);
      }
   }

/** Exchanges two elements in an array */
   private void swapElements(int[] array, int p1, int p2) {
      int temp = array[p1];
      array[p1] = array[p2];
      array[p2] = temp;
   }

}
```

elements to an array without allocating a new array at a larger size and then copying all the elements from the old array into the new one.

To streamline the process of working with dynamic arrays, Java programmers often avoid using arrays in favor of the **ArrayList** class defined in the package **java.util**. The **ArrayList** class provides the traditional operations associated with arrays along with a set of additional operations that turn out to be quite useful. The most important operations are summarized in Figure 10-4.

If you use the **ArrayList** class, your code will change markedly in its syntax from the array-based model. The **ArrayList** is a standard object class, which means that each of its operations is invoked as a method call. The brackets used for array notation go away. The resulting code, however, does introduce a number of complexities, of which the following are the most important:

- The elements of an **ArrayList** are declared to be of the class **Object**, which is the ultimate superclass of every Java class. That means that any Java object can be stored in an **ArrayList**, but that you must usually use a type cast to convert that object to a more specific type.
- Values that are not objects—specifically the primitive Java types—cannot be directly stored in an **ArrayList**. This restriction makes it more difficult to use the **ArrayList** class when the elements are, for example, integers or characters than it is if those values are actual objects like strings. You can, however, get around this problem by using the wrapper classes introduced in Chapter 7 to encapsulate a primitive value inside an object.

Figure 10-5 shows the implementation of a **ReverseArrayList** program that uses an

---

FIGURE 10-4   **Important methods in the ArrayList class**

`int size()`
Returns the number of elements in the **ArrayList**.
`void add(Object element)`
Adds a new element to the end of the **ArrayList**.
`void add(int index, Object element)`
Inserts a new element into the **ArrayList** before the position specified by **index**.
`void remove(int index)`
Removes the element at the specified position.
`void remove(Object element)`
Removes the first instance of the specified element, if any.
`void clear()`
Removes all elements from the **ArrayList**.
`Object get(int index)`
Returns the object at the specified index.
`void set(int index, Object value)`
Sets the element at the specified index to the new value.
`int indexOf(Object value)`
Returns the index of the first occurrence of the specified value, or -1 if it does not appear.
`boolean contains(Object value)`
Returns **true** if the **ArrayList** contains the specified value.
`boolean isEmpty()`
Returns **true** if the **ArrayList** contains no elements.

Note: Several of the methods shown here as having no return value in fact return something. In most cases, it is better not *to know what these methods return and to use them strictly as procedures.*

FIGURE 10-5   Program to reverse an ArrayList of integers

```java
/*
 * File: ReverseArrayList.java
 * ---------------------------
 * This file reads in an array of integers and then displays
 * those elements in reverse order. This version uses an ArrayList
 * to allow the size of the array to vary dynamically.
 */

import acm.program.*;
import java.util.*;

public class ReverseArrayList extends ConsoleProgram {

/** Define the sentinel value */
   private static final int SENTINEL = 0;

/** Runs the program */
   public void run() {
      println("This program reverses the elements in an ArrayList.");
      println("Use " + SENTINEL + " to signal the end of the list.");
      ArrayList list = readArrayList();
      reverseArrayList(list);
      printArrayList(list);
   }

/** Reads the data into the list */
   private ArrayList readArrayList() {
      ArrayList list = new ArrayList();
      while (true) {
         int value = readInt(" ? ");
         if (value == SENTINEL) break;
         list.add(new Integer(value));
      }
      return list;
   }

/** Prints out the data from the list, one per line */
   private void printArrayList(ArrayList list) {
      for (int i = 0; i < list.size(); i++) {
         Integer valueAsInteger = (Integer) list.get(i);
         println(valueAsInteger.intValue());
      }
   }

/** Reverses the data in an ArrayList */
   private void reverseArrayList(ArrayList list) {
      for (int i = 0; i < list.size() / 2; i++) {
         swapElements(list, i, list.size() - i - 1);
      }
   }

/** Exchanges two elements in an ArrayList */
   private void swapElements(ArrayList list, int p1, int p2) {
      Object temp = list.get(p1);
      list.set(p1, list.get(p2));
      list.set(p2, temp);
   }
}
```

ArrayList rather than an array to hold the elements. The conceptual effect of this program is the same as that of the **ReverseArray** program from Figure 10-3, but the implementation differs in the following ways:

- The end of the input is indicated by a sentinel and not by counting the number of items.
- The values read in from the user are not stored at a specific index position but are instead added to the end of the **ArrayList** using the **add** method.
- The values stored in the array must be objects of class **Integer** rather than primitive values of type **int**. Using wrapper classes means that the program must explicitly create the wrapper value using a constructor before storing it into the **ArrayList** and call **intValue** to retrieve the primitive value at the end.
- The **ReverseArrayList** program uses the **get** and **set** methods to read and write values in the array. The square brackets used to denote selection in an array are not used for an **ArrayList**.
- The **size** method returns the number of elements because the **ArrayList** class does not support a **length** field in the way that arrays do.

10.5 Using arrays for tabulation

The data structure of a program is typically designed to reflect the organization of data in the real-world domain of the application. If you are writing a program to solve a problem that involves a list of values, you should probably use an array to represent that list in the program. For example, in the **GymnasticsJudge** program shown in Figure 10-1, the problem involves a list of scores—one for each of five judges. Because the individual scores form a list in the conceptual domain of the application, it is not surprising that an array is used to represent the data in the program. The array elements have a direct correspondence to the individual data items in the list. Thus, **scores[0]** corresponds to the score for judge #0, **scores[1]** to the score for judge #1, and so on.

In general, whenever an application involves data that can be represented in the form of a list like

$$a_0, a_1, a_2, a_3, a_4, \ldots, a_{N-1}$$

an array is the natural choice for the underlying representation. It is also quite common for programmers to refer to the index of an array element as a **subscript,** reflecting the fact that arrays are used to hold data that would typically be written with subscripts in mathematics.

There are, however, important uses of arrays in which the relationship between the data in the application domain and the data in the program takes a different form. Instead of storing the data values in successive elements of an array, for some applications it makes more sense to use the data to generate array indices. Those indices are then used to select elements in an array that records some statistical property of the data as a whole.

Understanding how this approach works and appreciating how it differs from more traditional uses of arrays requires looking at a concrete example. Suppose you want to write a program that reads lines of text from the user and keeps track of how often each of the 26 letters appears. When the user types a blank line to signal the end of the input, the program should display a table indicating how many times each letter appears in the input data.

In order to generate this kind of letter-frequency table, the program has to search each line of text character by character. Every time a letter appears, the program must update a

running count that keeps track of how often that letter has appeared so far in the input. The interesting part of the problem lies in designing the data structure necessary to maintain a count for each of the 26 letters.

It is possible to solve this problem without arrays by defining 26 separate variables— **nA**, **nB**, **nC**, and so forth up to **nZ**—and then using a **switch** statement to check all 26 cases:

```
switch (Character.toUpperCase(ch)) {
    case 'A': nA++; break;
    case 'B': nB++; break;
    case 'C': nC++; break;
    . . .
    case 'Z': nZ++; break;
}
```

This process results in a long, repetitive program. A better approach is to combine the 26 individual variables into an array and then use the character code to select the appropriate element within the array. Each element contains an integer representing the current count of the letter that corresponds to that index in the array. If you call the array **letterCounts**, you can declare it by writing

```
int[] letterCounts = new int[26];
```

This declaration allocates space for an integer array with 26 elements, as shown in this diagram:

letterCounts

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
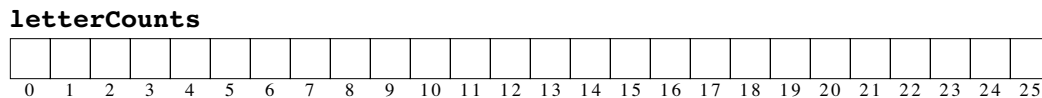
Each time a letter character appears in the input, you need to increment the corresponding element in **letterCounts**. Finding the element to increment is simply a matter of converting the character into an integer in the range 0 to 25 by using character arithmetic as discussed in Chapter 9. The code for the **CountLetterFrequencies** program appears in Figure 10-6.

10.6 Initialization of arrays

Like any other variable, array variables can be declared to be either local variables or instance variables. In either case, an array may also be given an initial set of values using a very convenient syntax, in which the equal sign specifying the initial value is followed by a list of the initial values for each element, enclosed in curly braces. For example, the declaration

```
int[] digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

introduces a local variable called **digits** in which each of the 10 elements is initialized to its own index number.

As a second example, imagine you're writing a program that requires an array containing the names of all U.S. cities with populations of over 1,000,000. Taking data from the 1990 census, you could declare that array as a named constant using the following code:

FIGURE   10-6   Program to produce a letter-frequency table

```java
/*
 * File: CountLetterFrequencies.java
 * --------------------------------
 * This program creates a table of the letter frequencies in a
 * paragraph of input text terminated by a blank line.
 */

import acm.program.*;

public class CountLetterFrequencies extends ConsoleProgram {

/** Runs the program */
   public void run() {
      println("This program counts letter frequencies.");
      println("Enter a blank line to indicate the end of the text.");
      initFrequencyTable();
      while (true) {
         String line = readLine();
         if (line.length() == 0) break;
         countLetterFrequencies(line);
      }
      printFrequencyTable();
   }

/** Initializes the frequency table to contain zeros */
   private void initFrequencyTable() {
      frequencyTable = new int[26];
      for (int i = 0; i < 26; i++) {
         frequencyTable[i] = 0;
      }
   }

/** Counts the letter frequencies in a line of text */
   private void countLetterFrequencies(String line) {
      for (int i = 0; i < line.length(); i++) {
         char ch = line.charAt(i);
         if (Character.isLetter(ch)) {
            int index = Character.toUpperCase(ch) - 'A';
            frequencyTable[index]++;
         }
      }
   }

/** Displays the frequency table */
   private void printFrequencyTable() {
      for (char ch = 'A'; ch <= 'Z'; ch++) {
         int index = ch - 'A';
         println(ch + ": " + frequencyTable[index]);
      }
   }

/** Private instance variables */
   private int[] frequencyTable;

}
```

```
private static final String[] CITIES_OVER_ONE_MILLION = {
    "New York",
    "Los Angeles",
    "Chicago",
    "Houston",
    "Philadelphia",
    "San Diego",
    "Detroit",
    "Dallas",
};
```

If you then want to add the figures from the 2000 census, Phoenix, Las Vegas, and San Antonio join this list. To do so, you correct the array simply by adding the new names to the initializer list. The compiler then expands the array size to accommodate the new values.

Note that the last initializer for the **CITIES_OVER_ONE_MILLION** array is followed by a comma. This comma is optional, but it is often makes sense to include it. Doing so allows you to add new cities at the end without having to change the existing entries.

10.7 Multidimensional arrays

In Java, the elements of an array can be of any type. In particular, the elements of an array can themselves be arrays. Arrays of arrays are called **multidimensional arrays.** The most common form of multidimensional array is the two-dimensional array, which is most often used to represent data in which the individual entries form a rectangular structure marked off into rows and columns. This type of two-dimensional structure is called a **matrix.** Arrays of three or more dimensions are also legal in Java but occur much less frequently.

As an example of a two-dimensional array, suppose you wanted to represent a game of tic-tac-toe as part of a program. As you probably know, tic-tac-toe is played on a board consisting of three rows and three columns, as follows:
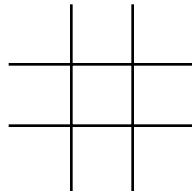
Players take turns placing the letters *X* and *O* in the empty squares, trying to line up three identical symbols horizontally, vertically, or diagonally.

To represent the tic-tac-toe board, the most sensible strategy is to use a two-dimensional array with three rows and three columns. Although you could also define an enumeration type to represent the three possible contents of each square—empty, *X,* and *O*—it is probably simpler in this case to use **char** as the element type and to represent the three legal states for each square using the characters **' '**, **'X'**, and **'O'**. The declaration for the tic-tac-toe board would then be written as

```
char[][] board = new char[3][3];
```

Given this declaration, you could then refer to the characters representing the individual squares on the board by supplying two separate indices, one specifying the row number and another specifying the column number. In this representation, each number varies over the range 0 to 2, and the individual positions in the board have the following names:

board[0][0]	**board[0][1]**	**board[0][2]**
board[1][0]	**board[1][1]**	**board[1][2]**
board[2][0]	**board[2][1]**	**board[2][2]**

Internally, Java represents the variable **board** as an array of three elements, each of which is an array of three characters. The memory allocated to **board** consists of nine characters arranged in the following form:

		board[0][0]
board[0]		**board[0][1]**
		board[0][2]
		board[1][0]
board[1]		**board[1][1]**
		board[1][2]
		board[2][0]
board[2]		**board[2][1]**
		board[2][2]
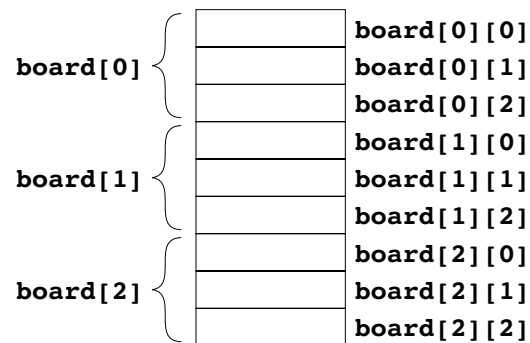
In the two-dimensional diagram of the **board** array, the first index is assumed to indicate the row number. This choice, however, is arbitrary because the two-dimensional geometry of the matrix is entirely conceptual; in memory, these values form a one-dimensional list. If you wanted the first index to indicate the column and the second to indicate the row, the only methods you would need to change would be those that depend on the conceptual geometry, such as a method that displays the current state of the board. In terms of the internal arrangement, however, it is always true that the first index value varies least rapidly as the array elements are positioned in memory. Thus all the elements of **board[0]** appear in memory before any elements of **board[1]**.

Passing multidimensional arrays to methods

Multidimensional arrays are passed between methods just as single dimensional arrays are. The parameter declaration in the method header looks like the original declaration of the variable and includes the index information. For example, the following method displays the current state of the **board** array:

```
private void displayBoard(char[][] board) {
   for (int row = 0; row < 2; row++) {
      if (row > 0) println("-----+-----+-----");
      println("     |     |");
      print("  ");
      for (int col = 0; col < 2; col++) {
         if (col > 0) print("  |  ");
         print(board[row][col]);
      }
      println();
      println("     |     |");
   }
}
```

Much of the code in **DisplayBoard** is used to format the output so that the board appears in an easy-to-read form.

Initializing multidimensional arrays

You can use static initialization with multidimensional arrays just like single dimensional arrays. To emphasize the overall structure, the values used to initialize each internal array are usually enclosed in additional set of curly braces. For example, the declaration

```
static double identityMatrix[3][3] = {
    { 1.0, 0.0, 0.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 0.0, 1.0 }
};
```

declares a 3x3 matrix of floating-point numbers and initializes it to contain the following values:

1.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	1.0

This particular matrix comes up frequently in mathematical applications and is called the **identity matrix.**

Summary

In this chapter, you have learned about two strategies that Java provides for representing lists of data: a language-level facility called the *array* and a class in the **java.util** package called **ArrayList**. These two strategies are similar in many respects, but have different properties that make each strategy appropriate for particular applications.

Arrays are built into Java and are supported by the syntax of the language. As in most programming languages, Java uses arrays to represent collections of data that are both *ordered* and *homogeneous*. Each component of an array is called an *element* and is identified by a numeric *index*. In Java, all arrays begin with index number 0. The number of elements in an array is called its *length*.

The process of creating an array variable consists of two steps: declaring the array variable and allocating space for the elements. In most applications, it is easiest to combine these operations into a single declaration line with the following paradigmatic form:

> *type***[]** *name* **= new** *type***[***length***];**

This declaration creates an array variable called *name* with *length* elements, each of which is an instance of *type*. You can also allocate an array by listing its elements in curly braces, as in the following example, which creates a ten-element string array in which each element is initialized to the English name of that number:

```
String[] digitNames = {
    "zero", "one", "two", "three", "four",
    "five", "six", "seven", "eight", "nine"
};
```

The process of referring to an individual element in an array is called *selection* and is indicated by writing an expression in square brackets after the name of the array. The index of an array is typically an `int`, but can also be any scalar type, such as a `char`. Selecting an element outside the index bounds of an array generates an error. You can determine the length of a Java array by selecting its `length` field.

A Java array is an object, which means that passing an array as an argument to a method copies a *reference* to the array into the corresponding formal parameter. Because this reference refers to the same object as the caller, any changes that are made to elements of the array within the method will persist after the method returns.

Arrays can be declared with more than one index, in which case they are called *multidimensional arrays*. In Java, multidimensional arrays are treated as arrays of arrays. The first index value selects an element of the outermost array, the second index value selects an element from that subarray, and so on.

Review questions

1. What are the two characteristic properties of an array?

2. Define the following terms: *element, index, element type, array length, selection*.

3. Write declarations that create the following array variables:

   a) An array `doubleArray` consisting of 100 values of type `double`
   b) An array `inUse` consisting of 16 values of type `boolean`
   c) An array `lines` consisting of 50 strings

4. Write the variable declaration and `for` loop necessary to create and initialize the following integer array:

squares

0	1	4	9	16	25	36	49	64	81	100
0	1	2	3	4	5	6	7	8	9	10

5. What are the two approaches outlined in this chapter for representing an array in which the natural index values begin at 1 instead of at 0? What are the tradeoffs between these two approaches?

6. True or false: Arrays are represented in memory in the way that primitive values are.

7. How do you determine the length of an array?

8. What is the role of the variable `temp` in the `swapElements` method?

9. What are some of the advantages of using the `ArrayList` class as opposed to Java arrays?

10. Describe the effect of each of the following `ArrayList` methods: `size`, `add`, `set`, `get`, and `remove`.

11. What strategy do you need to adopt if you want to add a value of a primitive type to an `ArrayList`?

12. How would you declare and initialize an array variable **powersOfTwo** that contains the first eight powers of two $(1, 2, 4, 8, 16, 32, 64,$ and $128)$?

13. What is a multidimensional array?

14. Assuming that the base address for the array is 1000 and that values of type **int** require two bytes of memory, draw a diagram that shows the address of each element in the array declared as follows:

```
int[][] rectangular = new int[2][3];
```

15. What variable declaration would you use to record the state of a chessboard, which consists of an 8×8 array of squares, each of which may contain any one of the following symbols:

K	white king	**k**	black king
Q	white queen	**q**	black queen
R	white rook	**r**	black rook
B	white bishop	**b**	black bishop
N	white knight	**n**	black knight
P	white pawn	**p**	black pawn
—	empty square		

How could you initialize this array so that it holds the standard starting position for a chess game:

r	n	b	q	k	b	n	r
p	p	p	p	p	p	p	p
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
P	P	P	P	P	P	P	P
R	N	B	Q	K	B	N	R

Programming exercises

1. Because individual judges may have some bias, it is common practice to throw out the highest and lowest score before computing the average. Write a program that reads in scores from a panel of seven judges and then computes the average of the five scores that remain after discarding the highest and lowest.

2. In statistics, a collection of data values is usually referred to as a **distribution.** A primary purpose of statistical analysis is to find ways to compress the complete set of data into summary statistics that express properties of the distribution as a whole. The most common statistical measure is the **mean,** which is simply the traditional average. The mean of a distribution is usually represented by the Greek letter μ.

   Write a method **mean(array)** that returns the mean of an array of type **double**. Test your method by incorporating it into the **GymnasticsJudge** program in Figure 10-1.

3. Another common statistical measure is the **standard deviation,** which provides an indication of how much the individual values in the distribution differ from the mean. To calculate the standard deviation whose elements are $x_1, x_2, \ldots, x_n$ you need to perform the following steps:

   a) Calculate the mean of the distribution as in exercise 2.
   b) Go through the individual data items in the distribution and calculate the square of the difference between each data value and the mean. Add all these values to a running total.
   c) Take the total from step b and divide it by the number of data items.
   d) Calculate the square root of the resulting quantity, which represents the standard deviation.

   In mathematical form, the standard deviation (σ) is given by the following formula:

   $$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(\mu - x_i)^2}{n}}$$

   where μ signifies the mean. The Greek letter sigma (Σ) represents a summation of the quantity that follows, which in this case is the square of the difference between the mean and each individual data point.

   Write a method **stdev(array)** that takes an array of **double**s and returns the standard deviation of the data distribution contained in the array.

   Note: The procedure given here is used by statisticians to compute the standard deviation of a complete data distribution. If you want to calculate the standard deviation based instead on a sample of the distribution, you need to divide by $n - 1$.

4. In the third century B.C.E., the Greek astronomer Eratosthenes developed an algorithm for finding all the prime numbers up to some upper limit N. To apply the algorithm, you start by writing down a list of the integers between 2 and N. For example, if N were 20, you would begin by writing down the following list:

   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16   17   18   19   20

   You then begin by circling the first number in the list, indicating that you have found a prime. You then go through the rest of the list and cross off every multiple of the value you have just circled, since none of those multiples can be prime. Thus, after executing the first step of the algorithm, you will have circled the number 2 and crossed off every multiple of two, as follows:

   ② 3 ✗ 5 ✗ 7 ✗ 9 ✗ 11 ✗ 13 ✗ 15 ✗ 17 ✗ 19 ✗

   From here, you simply repeat the process by circling the first number in the list that is neither crossed off nor circled, and then crossing off its multiples. In this example, you would circle 3 as a prime and cross off all multiples of 3 in the rest of the list, which would result in the following state:

2 3 ✗ 5 ✗ 7 ✗✗✗ 11 ✗ 13 ✗✗✗ 17 ✗ 19 ✗

Eventually, every number in the list will either be circled or crossed out, as shown in this diagram:

2 3 ✗ 5 ✗ 7 ✗✗✗ 11 ✗ 13 ✗✗✗ 17 ✗ 19 ✗

The circled numbers are the primes; the crossed-out numbers are composites. This algorithm for generating a list of primes is called the **sieve of Eratosthenes.**

Write a program that uses the sieve of Eratosthenes to generate a list of the primes between 2 and 1000.

5. In May of 1844, Samuel F. B. Morse sent the message "What hath God wrought!" by telegraph from Washington to Baltimore, heralding the beginning of the age of electronic communication. To make it possible to communicate information using only the presence or absence of a single tone, Morse designed a coding system in which letters and other symbols are represented as coded sequences of short and long tones, traditionally called *dots* and *dashes*. In Morse code, the 26 letters of the alphabet are represented by the following codes:

A	· ▬	J	· ▬ ▬ ▬	S	· · ·	
B	▬ · · ·	K	▬ · ▬	T	▬	
C	▬ · ▬ ·	L	· ▬ · ·	U	· · ▬	
D	▬ · ·	M	▬ ▬	V	· · · ▬	
E	·	N	▬ ·	W	· ▬ ▬	
F	· · ▬ ·	O	▬ ▬ ▬	X	▬ · · ▬	
G	▬ ▬ ·	P	· ▬ ▬ ·	Y	▬ · ▬ ▬	
H	· · · ·	Q	▬ ▬ · ▬	Z	▬ ▬ · ·	
I	· ·	R	· ▬ ·			

You can easily store these codes in a program by declaring an array with 26 elements and storing the sequence of characters corresponding to each letter in the appropriate array entry.

Write a program that reads in a string from the user and translates each letter in the string to its equivalent in Morse code, using periods to represent dots and hyphens to represent dashes. Separate words in the output by calling **println** whenever you encounter a space in the input, but ignore all other punctuation characters. Your program should be able to generate the following sample run:

```
MorseCode
This program translates a line into Morse code.
Enter English text: What hath God wrought
.-- .... .- -
.... .- - ....
--. --- -..
.-- .-. --- ..- --. .... -
```

6. A **histogram** is a graphical representation of a statistical distribution that shows how many elements fall into a set of discrete ranges. For example, given a set of exam scores that contains the following values

$$100, 95, 47, 88, 86, 92, 75, 89, 81, 70, 55, 80$$

a traditional histogram would look something like this:

```
                                                *
                                                *
                                                *
                                      *         *         *
                            *         *         *         *         *
  ──────────────────────────────────────────────────────────────────────
   0–9   10–19   20–29   30–39   40–49   50–59   60–69   70–79   80–89   90–99   100
```
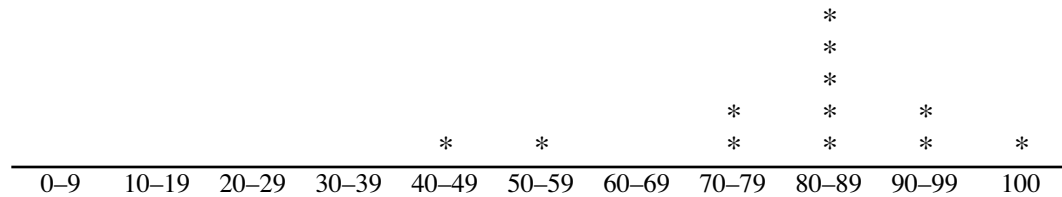
The asterisks in the histogram indicate one score in the 40s, one score in the 50s, five scores in the 80s, and so forth.

When you generate a histogram using a computer, it is usually easier to display the data sideways on the page, as in this sample run:

```
●●●                          Histogram
0–9    │
10–19  │
20–29  │
30–39  │
40–49  │ *
50–59  │ *
60–69  │
70–79  │ **
80–89  │ *****
90–99  │ **
100    │ *
```

Write a program that reads in an array of integers and then displays a histogram of those numbers, divided into the ranges 0–9, 10–19, 20–29, and so forth, up to the range containing only the value 100. Your program should generate output that looks as much like the sample run as possible.

7. The mechanism depicted in the following diagram—which has sometimes been marketed by toy stores as a "probability board"—can be used to demonstrate important properties of random processes.
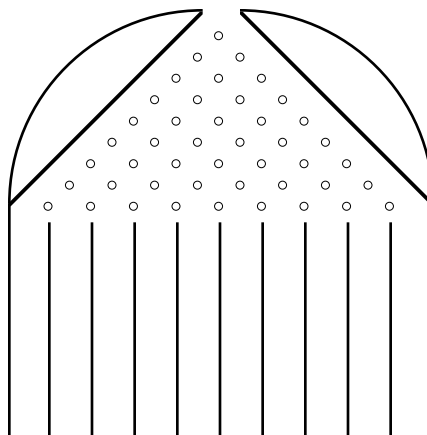
The mechanism works as follows. You start by dropping a marble in the hole at the top. The marble falls down and hits the uppermost peg, indicated by the small circle in the diagram. The marble bounces off the peg and falls, with equal probability, to the left or right. Whichever way it goes, it then hits a peg on the second level and bounces again, one direction or the other. The process continues until the marble passes all the pegs and drops into one of the channels at the bottom. For example, the dotted line in the following diagram shows one possible path for the marble:
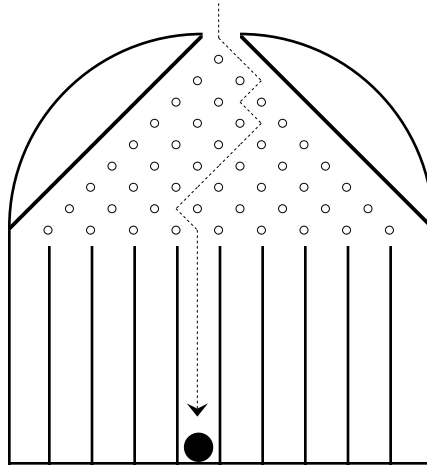
Write a program to simulate the operation of dropping 50 marbles into a probability board with 10 channels along the bottom, as in the diagram. Your program should display its results pictorially using the **acm.graphics** package described in Chapter 8. As each marble lands, the program should draw a circle at the appropriate place on the screen. The following screen image shows one possible sample run:
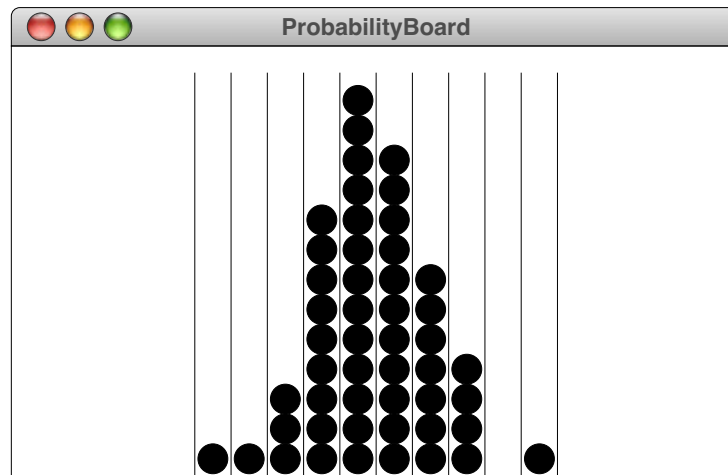
Note that the marbles tend to cluster in the center channels. The reason for this behavior is that there are many more ways in which a marble can reach the center columns than the ones on the ends. For example, to reach the leftmost column, the marble must have bounced to the left nine times in a row. In contrast, there are many paths from the top to the two center columns because the order of the left and right bounces can be reordered without affecting where the marble ends up. In general, the likelihood that a random process will end up in a particular state depends on the number of ways of reaching that state.

8. When you are trying to represent the behavior of some quantity that varies over time, one of the usual tools is the **line graph,** in which a set of data values are plotted on an *x*-*y* grid with each pair of adjacent points connected by a straight line. For example, given the following set of 10 points:

    (0.0, 0.67)
    (0.4, 0.68)
    (0.8, 0.71)
    (1.2, 0.86)
    (1.6, 0.86)
    (2.0, 1.04)
    (2.4, 1.30)
    (2.8, 1.81)
    (3.2, 1.46)
    (3.6, 1.86)

    the line graph that represents them looks like this:
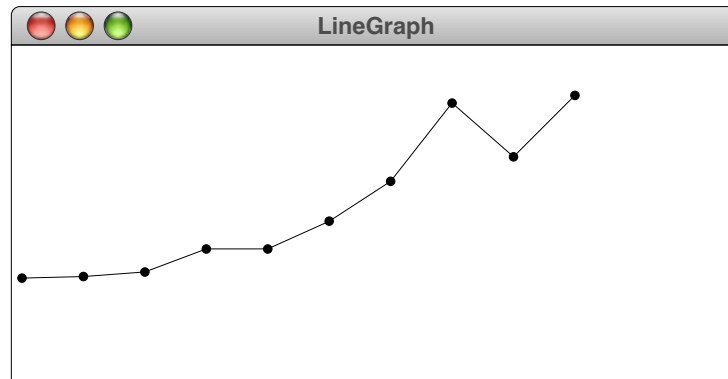
    Write a method **DrawLineGraph** that generates a line graph given an array of *x*-coordinate values and a second array of the corresponding *y*-coordinate values.

9. Using the declaration of the tic-tac-toe board given in this chapter, write a method **isWinningPosition(board, player)** that returns **true** if the specified player, which is either the character **'X'** or the character **'O'**, has won the tic-tac-toe game. A winning position is one in which three identical symbols are lined up horizontally, vertically, or diagonally. Test your method by writing a main program that reads in the current contents of the board array and then reports whether either player has won the game, as illustrated by this following sample run:
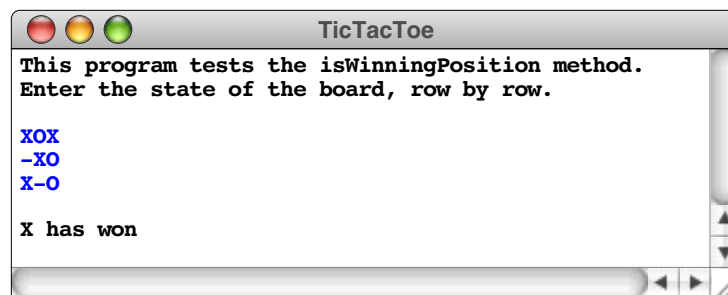
10. The initial state of a checkers game is shown in the following diagram:
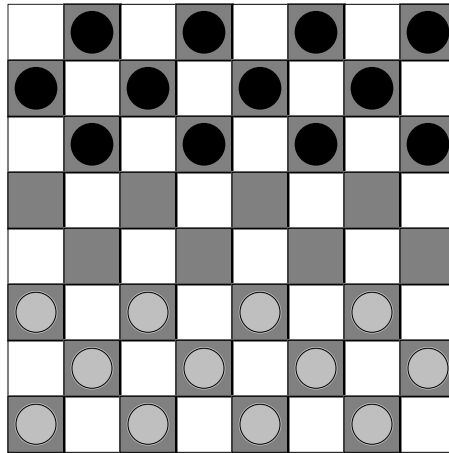
The dark squares in the bottom three rows are occupied by red checkers; the dark squares in the top three rows contain black checkers. The two center rows are unoccupied.

If you want to store the state of a checkerboard in a computer program, you need a two-dimensional array indexed by rows and columns. The elements of the array could be of various different types, but a reasonable approach—as illustrated by the tic-tac-toe example—is to use characters. For example, you could use the letter *r* to represent a red checker and the letter *b* to represent a black checker. Empty squares can be represented as spaces or hyphens depending on whether the color of the square was light or dark.

Implement a method **initCheckerboard** that initializes a checkerboard array so that it corresponds to the starting position of a checkers game. Implement a second method **displayCheckerboard** that displays the current state of a checkerboard on the console, like this:
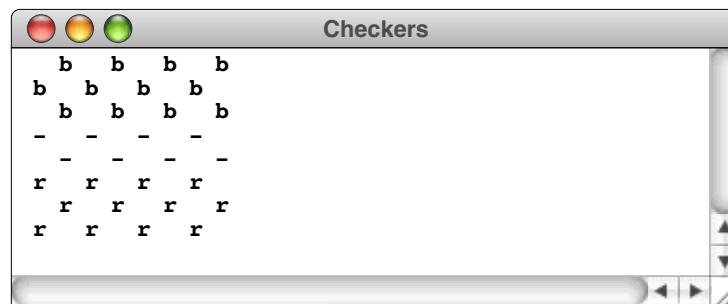
11. Redesign the data structure for the checkerboard described in the previous exercise so that each of the elements of the checkerboard array is a **GCompound** object that draws itself at the appropriate position on the graphics window. For example, the element in the lower left corner should be a **GCompound** that contains two graphical objects:

   • A filled **GRect** to serve as the background square
   • A filled red circle to indicate the checker

12. A **magic square** is a two-dimensional array of integers in which the rows, columns, and diagonals all add up to the same value. One of the most famous magic squares appears in the engraving "Melencolia" by Albrecht Dürer shown at the right, in which a 4×4 magic square appears in the upper right, just under the bell. In Dürer's square, which can be read more easily as

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

all four rows, all four columns, and both diagonals add up to 34.

Albrecht Dürer's "Melencolia" containing a magic square (1514)

A more familiar example is the following 3×3 magic square in which each of the rows, columns, and diagonals add up to 15, as shown:

Implement a method `isMagicSquare` that tests to see whether an *N*×*N* array contains a magic square.

Chapter 11
Searching and Sorting

"I weep for you," the Walrus said:
"I deeply sympathize."
With sobs and tears he sorted out
Those of the largest size
 — Lewis Carroll, *Through the Looking Glass,* 1872

C. A. R. (Tony) Hoare

Tony Hoare is Professor Emeritus of Computer Science at Oxford University and a senior researcher at Microsoft's Research Laboratory in Cambridge, England. After completing his first degree in philosophy at Oxford in 1956, Hoare became fascinated by the emerging world of computer science and refocused his graduate study in that area. During his graduate-school years, Hoare developed a highly efficient sorting algorithm called Quicksort, which remains in active use today. He also led the effort during the 1960s to create the first commercial compiler for Algol 60, a programming language that served as an important model for subsequent languages, including Java. Professor Hoare received the ACM Turing Award—the computing profession's highest honor—in 1980.

In Chapter 10, you had the opportunity to learn about most of the fundamental array operations and to see how arrays are used in a variety of applications. There are, however, two important array operations Chapter 10 omits so that they can be covered more thoroughly in a chapter of their own. These operations are:

- **Searching,** which is the process of finding a particular element in an array
- **Sorting,** which is the process of rearranging the elements in an array so that they are stored in some well-defined order

Because searching and sorting are closely related to arrays, this chapter is in a sense a continuation of the array discussion. This chapter, however, has another central theme that links it not just to Chapter 10 but also to the discussion of algorithmic methods in Chapter 5. Because there are many different strategies for searching and sorting—with vastly different levels of efficiency—these operations raise interesting algorithmic issues.

11.1 Searching

As noted in the introduction to this chapter, the searching problem consists of finding an element in an array. The simplest strategy—although not necessarily the most efficient one—is captured in the following advice that the King of Hearts gives the White Rabbit in Lewis Carroll's *Alice's Adventures in Wonderland:*

> *Begin at the beginning, and go on till you come to the end:*
> *then stop.*

Turning that informal statement into an algorithm for searching is not at all difficult. The only modification that you need to make is that the algorithm should also stop if it finds the element it is searching for. Thus, you might express a more complete account of the searching example algorithm as follows:

> *Begin at the beginning, and go on till you either find the*
> *element you're looking for or you come to the end. If you*
> *find the element, you can report its position; if you reach*
> *the end, that means that the element does not appear.*

Because the process starts at the beginning and proceeds through in a straight line through the elements of the array, this algorithm is called **linear search.**

Searching in an integer array

Turning this informal description into a Java method requires adding some details to the specification of precisely what that method should do. For concreteness, suppose that you have been asked to write a method `findIntInArray` that looks for the integer `key` in an array of integers. The header line for such a method would look like this:

```
private int findIntInArray(int key, int[] array)
```

The result of the method is the first index at which the key appears as an element of the array, assuming that the element exists at all. If the value key does not match any of the elements in the array, `findIntInArray` should return –1 to indicate that fact.

The implementation of `findIntInArray` is a straightforward translation of Lewis Carroll's strategy:

```
    private int findIntInArray(int key, int[] array) {
       for (int i = 0; i < array.length; i++) {
          if (key == array[i]) return i;
       }
       return -1;
    }
```

The **for** loop begins at the beginning and continues until it comes to the end of the array. The method returns whenever it finds the key along the way or, failing that, at the end of the entire loop.

   The **for** loop in **findIntInArray** that implements the linear search algorithm represents an important programming pattern that will come up very often in your code. Whenever you are given a problem that involves searching for a value in an array, you should be able to write down the **for** loop that does the job without giving any real thought to the details.

Searching a table

As a prelude to a discussion of different algorithms for searching, this section introduces a more sophisticated searching application that will make it easier to describe the issues that arise. Suppose that you want to represent in a program the mileage table shown in Figure 11-1. The individual entries in the table form a two-dimensional array with 12 rows and 12 columns. Each entry in the matrix is an integer that indicates the number of miles between the cities corresponding to that row and column. The following declaration introduces an instance variable called **mileageTable** and uses static initialization to fill it with the data shown in Figure 11-1:

FIGURE 11-1   **Mileage table for U.S. cities**

	Atlanta	Boston	Chicago	Denver	Detroit	Houston	Los Angeles	Miami	New York	Philadelphia	San Francisco	Seattle
Atlanta		1108	708	1430	732	791	2191	663	854	748	2483	2625
Boston	1108		994	1998	799	1830	3017	1520	222	315	3128	3016
Chicago	708	994		1021	279	1091	2048	1397	809	785	2173	2052
Denver	1430	1998	1021		1283	1034	1031	2107	1794	1739	1255	1341
Detroit	732	799	279	1283		1276	2288	1385	649	609	2399	2327
Houston	791	1830	1091	1034	1276		1541	1190	1610	1511	1911	2369
Los Angeles	2191	3017	2048	1031	2288	1541		2716	2794	2703	387	1134
Miami	663	1520	1397	2107	1385	1190	2716		1334	1230	3093	3303
New York	854	222	809	1794	649	1610	2794	1334		101	2930	2841
Philadelphia	748	315	785	1739	609	1511	2703	1230	101		2902	2816
San Francisco	2483	3128	2173	1255	2399	1911	387	3093	2930	2902		810
Seattle	2625	3016	2052	1341	2327	2369	1134	3303	2841	2816	810	

Source: Rand McNally atlas

```
private int[][] mileageTable = {
    {    0,1108, 708,1430, 732, 791,2191, 663, 854, 748,2483,2625},
    {1108,    0, 994,1998, 799,1830,3017,1520, 222, 315,3128,3016},
    { 708, 994,    0,1021, 279,1091,2048,1397, 809, 785,2173,2052},
    {1430,1998,1021,    0,1283,1034,1031,2107,1794,1739,1255,1341},
    { 732, 799, 279,1283,    0,1276,2288,1385, 649, 609,2399,2327},
    { 791,1830,1091,1034,1276,    0,1541,1190,1610,1511,1911,2369},
    {2191,3017,2048,1031,2288,1541,    0,2716,2794,2703, 387,1134},
    { 663,1520,1397,2107,1385,1190,2716,    0,1334,1230,3093,3303},
    { 854, 222, 809,1794, 649,1610,2794,1334,    0, 101,2930,2841},
    { 748, 315, 785,1739, 609,1511,2703,1230, 101,    0,2902,2816},
    {2483,3128,2173,1255,2399,1911, 387,3093,2930,2902,    0, 810},
    {2625,3016,2052,1341,2327,2369,1134,3303,2841,2816, 810,    0},
};
```

Because the city names in this chart are the same for both the rows and columns, they can be stored in a single array called **cityNames**, which you can declare and initialize as follows:

```
private String[] cityNames = {
    "Atlanta",
    "Boston",
    "Chicago",
    "Denver",
    "Detroit",
    "Houston",
    "Los Angeles",
    "Miami",
    "New York",
    "Philadelphia",
    "San Francisco",
    "Seattle",
};
```

Now that you have the data, the next question to consider is how to write a program that reads in the names of two cities and displays the distance between them, as illustrated by the following sample run:
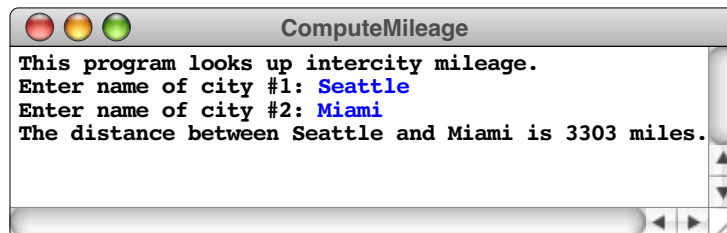
All your program has to do is execute the following steps:

1. Read in the names of the two cities as strings.
2. Find the index positions at which the city names occur in the **cityNames** array.
3. Use the index positions to select the result from **mileageTable**.

If you implement a method **findStringInArray** that uses the same linear-search pattern as **findIntInArray**, the implementation of the entire program is straightforward, and appears in Figure 11-2.

FIGURE 11-2   Program to compute the mileage between two cities

```java
/*
 * File: ComputeMileage.java
 * ------------------------
 * This program uses a table of mileage data to calculate
 * the distance between US cities.
 */

import acm.program.*;

public class ComputeMileage extends ConsoleProgram {

/** Runs the program */
   public void run() {
      println("This program looks up intercity mileage.");
      int city1 = getCity("Enter name of city #1: ");
      int city2 = getCity("Enter name of city #2: ");
      println("The distance between " + cityNames[city1]
                + " and " + cityNames[city2] + " is "
                + mileageTable[city1][city2] + " miles.");
   }

/* Method: getCity */
/**
 * Prompts the user for a city name, reads in a string, and returns
 * the index corresponding to that city, if it exists.  If the city
 * name is undefined, the user is given a chance to retry.
 */
   private int getCity(String prompt) {
      int index = -1;
      while (index == -1) {
         String name = readLine(prompt);
         index = findStringInArray(name, cityNames);
         if (index == -1) {
            println("Unknown city name -- try again.");
         }
      }
      return index;
   }

/* Method: findStringInArray(key, array) */
/**
 * Finds the first instance of the specified key in the array
 * and returns its index.  If key does not appear in the array,
 * findStringInArray returns -1.
 */
   private int findStringInArray(String key, String[] array) {
      for (int i = 0; i < array.length; i++) {
         if (key.equals(array[i])) return i;
      }
      return -1;
   }

   Include the definitions of mileageTable and cityNames here.

}
```

The linear-search algorithm used in **findStringInArray** starts at the beginning of the array and goes straight down the line of elements until it finds a match or reaches the end of the array. For an array of 12 city names, looking at every element takes very little time. But what if the array instead had thousands or even millions of elements? At some point, if the array became large enough, you would begin to notice a delay as the computer searched through every value. But is searching every value really necessary? It's worth stopping to think for a moment about this question.

Suppose that someone asked *you* to find the distance between Seattle and Miami in the mileage table. To find the entry for Seattle, would you start at the top of the page and work your way down? Probably not. Because the list of cities is in alphabetic order, you know that must come somewhere near the end of the list. Similarly, Miami is likely to appear near the middle. The odds are good that your eyes would find these values very quickly without ever looking at most of the names in the list.

Binary search

To take advantage of the fact that the **cityNames** array is in alphabetic order, you need to use a different algorithm. To illustrate the process as concretely as possible, let's suppose that you are looking for Miami in an array with the following values:

cityNames

0	Atlanta
1	Boston
2	Chicago
3	Denver
4	Detroit
5	Houston
6	Los Angeles
7	Miami
8	New York
9	Philadelphia
10	San Francisco
11	Seattle

Instead of starting at the top of the array as in linear search, what happens if you start by picking an element somewhere near the center? The index of the center element can be computed by averaging the endpoints of the index range and is therefore

$$\frac{0 + 11}{2}$$

When evaluated using integer arithmetic, this expression has the value 5.

The city name stored in **cityArray[5]** is Houston. Given that you're looking for Miami, what do you know at this point? You haven't found Miami yet, so you have to keep looking. On the other hand, you know that Miami must come after Houston because

the array is in alphabetic order. Thus, you can immediately eliminate all the city names in index positions 0 through 5, which leaves you in the following position:

cityNames

0	~~Atlanta~~
1	~~Boston~~
2	~~Chicago~~
3	~~Denver~~
4	~~Detroit~~
5	~~Houston~~
6	Los Angeles
7	Miami
8	New York
9	Philadelphia
10	San Francisco
11	Seattle

In one step, you've managed to cross out half the possibilities. The really good news, however, is that you can now do the same thing all over again. You know that Miami—if it exists in the list at all—must lie between positions 6 and 11 of **cityNames**. The center of that range can therefore be computed by evaluating

$$\frac{6 + 11}{2}$$

using integer arithmetic, which produces the value 8. Miami comes earlier in the alphabet than New York does, so you can now cross off four more positions.

cityNames

0	~~Atlanta~~
1	~~Boston~~
2	~~Chicago~~
3	~~Denver~~
4	~~Detroit~~
5	~~Houston~~
6	Los Angeles
7	Miami
8	~~New York~~
9	~~Philadelphia~~
10	~~San Francisco~~
11	~~Seattle~~

On the next cycle, you look at the element in position 6, which is the result of evaluating the expression

$$\frac{6 + 7}{2}$$

using integer arithmetic. Miami comes after Los Angeles, so you can cross off that value as well. On the fourth cycle, you have only a single element to check, which is indeed the entry for Miami at index position 7.

This algorithm—looking at the center element in a sorted array and then determining which half to search on that basis—is called **binary search.** To implement this algorithm, all you need to do is keep track of two indices that mark the endpoints of the index range within which the search is limited. In the method, these indices are stored in the variables **lh** and **rh**, which represent the left-hand (lower) index and right-hand (upper) index, respectively. Initially, these index bounds cover the entire array, but move closer together as possibilities are eliminated. If the index values ever cross, the key value does not exist in the array.

The code for the method **findStringInSortedArray**, which uses the binary search algorithm, appears in Figure 11-3.

FIGURE 11-3   **Implementation of the binary search algorithm for string arrays**

```
/**
 * Finds an instance of the specified key in the array, which must
 * be sorted in lexicographic order.  If the key exists, the method
 * returns an index at which that key appears, but this index will
 * not necessarily be the first if the same key appears multiply.
 * If key does not appear in the array, findStringInSortedArray
 * returns -1.
 *
 * This implementation uses the "binary search" algorithm.  At
 * each stage, the function computes the midpoint of the remaining
 * range and compares the element at that index position to the
 * key.  If there is a match, the function returns the index.
 * If the key is less than the string at that index position, the
 * function searches in the first half of the array; if the key is
 * larger, the function searches in the second half of the array.
 */

  private int findStringInSortedArray(String key, String[] array) {
      int lh = 0;
      int rh = array.length - 1;
      while (lh <= rh) {
          int mid = (lh + rh) / 2;
          int cmp = key.compareTo(array[mid]);
          if (cmp == 0) return mid;
          if (cmp < 0) {
              rh = mid - 1;
          } else {
              lh = mid + 1;
          }
      }
      return -1;
  }
```

Relative efficiency of the search algorithms

The discussion in the previous section suggests that the binary search algorithm is more efficient than the linear search algorithm. Even so, it is hard to appreciate just how much better binary search is without being able to compare the performance of the two algorithms using some quantitative measure. For searching, a convenient measure that provides a good indication of algorithmic performance is the number of times **equals** or **compareTo** is called to compare the key against some element in the array.

Suppose that you execute the linear search algorithm on an array containing N elements. How many times will the method call **equals**? The answer depends of course on where the key value shows up in the list. In the worst case—which occurs when the key is in the last position or does not appear at all—**findStringInArray** will call **equals** N times, once for each element in the array.

What about the binary search algorithm used in **findStringInSortedArray**? After the first call to **compareTo**, the algorithm can immediately eliminate half of the array elements, leaving only $N/2$ elements to search. After the second call, it can rule out half of those elements, leaving $N/4$ elements. Each time, the number of possibilities is halved. Eventually, after you divide an integer N in half enough times, you will eventually end up with 1, at which point there is only a single comparison left to be made. The number of steps required to reach this point is the number of times you can divide N by 2 before you get 1, which is represented by k in the following formula:

$$N \underbrace{/\, 2\, /\, 2\, /\, \ldots /\, 2\, /\, 2}_{k \text{ times}} \;=\; 1$$

Multiplying by all those 2s gives the equivalent equation

$$N = 2^k$$

If you remember logarithms from high-school algebra, you can express the value of k as

$$k = \log_2 N$$

Thus, to search an array of N elements requires N comparisons if you use linear search and $\log_2 N$ comparisons if you use binary search.

Expressing the relative efficiency of these algorithms in mathematical form is useful as a means of making quantitative predictions about efficiency. For most people, however, such formulas do not convey a real sense of how these algorithms compare. For that, you need to look at some numbers. The following table shows the closest integer to $\log_2 N$ for various values of N.

N	$\log_2 N$
10	3
100	7
1000	10
1,000,000	20
1,000,000,000	30

Reflecting on what the values in this table mean, you can see that, for small arrays, both strategies work reasonably well. On the other hand, if you have an array with 1,000,000,000 elements, linear search requires 1,000,000,000 comparisons to search that array in the worst case, whereas the binary search algorithm gets the job done using at most 30 comparisons. Clearly, this reduction in the number of required comparisons represents an overwhelming increase in algorithmic efficiency.

The only problem is that the binary search algorithm requires that the array elements be listed in sorted order. If they are not, you may have to resort to linear searching. Alternatively, you can ensure that the array elements are in the correct order by sorting the array yourself. Sorting an array is a slightly more challenging problem than searching one and is the subject of the remainder of this chapter.

11.2 Sorting

In most commercial applications, computers are used for extremely simple operations such as adding a sequence of numbers or calculating an average—precisely the sort of problem you learned to solve in the earlier chapters. However, several important operations required for commercial programming are more sophisticated. Of these, the most important is **sorting,** the process of arranging a list of values (usually represented as an array) into some well-defined order. For example, you might rank a list of numbers from lowest to highest based on their numeric value. Alternatively, you might choose to alphabetize a list of names. These two operations turn out to be quite similar. Despite differences in detail (one uses numbers and the other uses strings), the problem to be solved is precisely the same: given a list and a mechanism for comparing two elements in that list, how can you rearrange the elements of the list so that the elements are properly ordered?

Sorting an integer array

Let's consider, for example, the problem of sorting an array of integers. Suppose that you have been presented with an array of integers in some random order, such as the following:

31	41	59	26	53	58	97	93
0	1	2	3	4	5	6	7

What you need to do at this point is to define a new method, which you could call **sortIntegerArray**, that would rearrange the elements of this array so that they run from lowest to highest, as follows:

26	31	41	53	58	59	93	97
0	1	2	3	4	5	6	7

The header line for **sortIntegerArray** presumably looks like this:

```
private void sortIntegerArray(int[] array)
```

Writing the corresponding implementation, however, is trickier than it might seem, particularly if you are interested in finding an efficient strategy for sorting the data. As is the case with many problems in computer science, there are many different algorithms you can use. In an advanced computer science course, you might well spend several

weeks or even months studying various different algorithms for sorting, each of which has particular advantages or disadvantages. At this point in your study of computer science, however, it is best to begin with one algorithm for sorting that you can understand in detail.

The selection sort algorithm

Of the many possible sorting algorithms, one of the easiest to explain is the **selection sort** algorithm. When you apply the selection sort algorithm, you put the array into its final order one element at a time. In the first step, you find the smallest element in the entire list and put it where it belongs—at the beginning. In the second step, you find the smallest remaining element and put it in the second position. If you continue this process for the entire array, the final result is in sorted order.

To get a sense of the selection sort approach, watch what happens if you start with the following array of numbers:

31	41	59	26	53	58	97	93
0	1	2	3	4	5	6	7

Because the smallest element is the value 26 in position 3, you move this element into position 0. As with the **ReverseIntArray** program from Chapter 10, you don't want to lose track of the value that was originally in position 0, so the easiest thing to do is exchange the values in positions 0 and 3. Doing so leaves the array in the following state:

correctly positioned

26	41	59	31	53	58	97	93
0	1	2	3	4	5	6	7

After the exchange, position 0 is correctly filled with the smallest value.

From this point, you can proceed with the rest of the list. The next step is to use the same strategy to correctly fill the second position in the array. The smallest value (except for the value 26 already placed correctly) is the 31, which is now in position 3. If you exchange this value with the one at index position 1, you reach the following state, with the correct values in the first two elements:

correctly positioned

26	31	59	41	53	58	97	93
0	1	2	3	4	5	6	7

On the next cycle, you switch the next smallest value (which turns out to be 41) into position 2:

correctly positioned

26	31	41	59	53	58	97	93
0	1	2	3	4	5	6	7
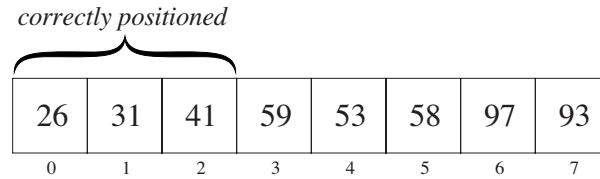
If you continue on in this way, you can correctly fill up index positions 3, 4, and so on until the array is completely sorted.

To keep track of which element you are trying to fill at each step in the algorithm, you can imagine that you use your left hand to point to each of the index positions in turn. For each left-hand position, you then use your right hand to identify the smallest element remaining in the rest of the array. Once you find it, you can just take the values to which your hands point and exchange them. In the implementation, your left and right hands are replaced by variables—**lh** and **rh**—that hold the index number of the appropriate element in the array.

You can turn this intuitive outline into pseudocode as follows:

> **for (***each index position* **lh** *in the array***) {**
>   Let **rh** *be the index position of the smallest value between* **lh** *and the end of the list*
>   *Swap the elements at index positions* **lh** *and* **rh**
> **}**

Replacing the pseudocode with the correct Java statements is straightforward, mostly because two of the operations are familiar: the **for** loop control line is a standard idiom, and you can accomplish the swap operation at the end of the loop by calling the **swapElements** method defined as part of the **ReverseIntArray** program in Figure 10-3. The one remaining step is the one that finds the smallest value. Following the discipline of stepwise refinement, you can define a new method to perform this operation and complete the coding of the **sortIntegerArray** procedure as follows:

```
private void sortIntegerArray(int[] array) {
   for (int lh = 0; lh < array.length; lh++) {
      int rh = findSmallestElement(array, lh, array.length - 1);
      swapElements(array, lh, rh);
   }
}
```

The **findSmallestElement** method takes three arguments: the array and two index numbers indicating the range within the array in which to find the smallest value. The method returns the index—not the value—of the smallest element of the array between the specified index positions. The simplest implementation of **findSmallestElement** is to go through the list, keeping track at each loop cycle of the index of smallest value so far. When you reach the end of the list, the smallest value so far will be the smallest value in the list as a whole. This suggests the following code, in which the variable **smallestIndex** keeps track of the index position of the smallest value so far:

```
private int findSmallestElement(int[] array, int p1, int p2) {
   int smallestIndex = p1;
   for (int i = p1 + 1; i <= p2; i++) {
      if (array[i] < array[smallestIndex]) smallestIndex = i;
   }
   return smallestIndex;
}
```

At the beginning of the scan, the first value you consider is automatically the smallest value so far. Thus, you can initialize **smallestIndex** to the starting index position, which is given by the parameter **p1**. As you look at each position in turn, you have to see if the current value is smaller than your previous candidate for the smallest value. If it is, the old value can no longer be the smallest in the entire list, and you need to correct the value of **smallestIndex** to indicate the new position, which will retain its value until you find an even smaller value.

The method **swapElements** is precisely the same method as in the **ReverseIntArray** implementation from Chapter 10, and there is no reason to rewrite it. Whenever you write a method that implements some generally useful operation, it is wise to keep that method around so that it is available for future use. Successful programmers always try to reuse existing code as much as possible because doing so saves the trouble of writing and debugging those parts of the program from scratch.

Copying the code for **swapElements** completes the code for the entire selection sort algorithm. The code for the **sortIntegerArray** method appears in Figure 11-4.

Evaluating the efficiency of selection sort

The selection sort algorithm has several positive qualities. For one thing, it is relatively easy to understand. For another, it gets the job done. There are, however, sorting algorithms that are far more efficient. Unfortunately, the best ones require techniques beyond your current level of programming knowledge. At the same time, you certainly have enough background to evaluate how efficient the selection sort algorithm is, even if you are not yet in a position to improve it.

One interesting question is how long it takes to execute selection sort on a given set of input data. There are two ways you can approach this question:

1. You can run the program and measure how long it takes. Because programs run very quickly on modern computers and often finish their work in a fraction of a second, you might not be able to measure the elapsed time with a stopwatch, but you can accomplish the same result by using the computer's internal clock.
2. You can think more generally about the operation of the program and try to develop a qualitative sense of how it behaves.

Measuring the running time of a program

To determine how long it takes to run a program, the most common approach is to use the **System** class in the **java.lang** package to keep track of the total time required. The **System** class exports a static method called **currentTimeMillis** that returns the current time. The result is expressed as a **long** that records the number of milliseconds between the current setting of the system clock and midnight on January 1, 1970, which is defined as the base for time calculations. Thus, it is possible to obtain a rough assessment of the time required to perform a calculation as follows:

```
long start = System.currentTimeMillis();
. . . Perform some calculation . . .
long elapsed = System.currentTimeMillis() - start;
```

This code, however, requires several cautionary notes:

- Short computations can often be completed in less than a millisecond, which means that the elapsed time given by comparing two calls to **currentTimeMillis** may in

---

FIGURE 11-4   Implementation of the selection sort algorithm

```java
/**
 * Sorts an integer array into increasing order.  The
 * implementation uses an algorithm called selection sort,
 * which can be described in English as follows.  With your left
 * hand, point at each element in the array in turn, starting at
 * index 0.  At each step in the cycle:
 *
 * <ol>
 * <li>Find the smallest element in the range between your left
 *     hand and the end of the array, and point at that element
 *     with your right hand.
 *
 * <li>Move that element into its correct index position by
 *     switching the elements indicated by your left and right
 *     hands.
 * </ol>
 */
   private void sortIntegerArray(int[] array) {
      for (int lh = 0; lh < array.length; lh++) {
         int rh = findSmallestElement(array, lh, array.length - 1);
         swapElements(array, lh, rh);
      }
   }

/**
 * Returns the index of the smallest element in the array between
 * index positions p1 and p2, inclusive.
 */
   private int findSmallestElement(int[] array, int p1, int p2) {
      int smallestIndex = p1;
      for (int i = p1 + 1; i <= p2; i++) {
         if (array[i] < array[smallestIndex]) smallestIndex = i;
      }
      return smallestIndex;
   }

/**
 * Exchanges the elements in an array at index positions p1 and p2.
 */
   private void swapElements(int[] array, int p1, int p2) {
      int temp = array[p1];
      array[p1] = array[p2];
      array[p2] = temp;
   }
}
```

some cases return 0. Worse yet, the definition of `currentTimeMillis` does not guarantee that the result is accurate at the millisecond level, because the internal clock in some systems does not provide that level of accuracy.

- The `currentTimeMillis` method returns the total elapsed time, which is not always a good measure of how much actual processing time has been dedicated to the algorithm you are seeking to measure. The Java runtime system introduces a certain amount of overhead, and there is no reliable way in Java to factor out this additional overhead time from the calculation. These variations, moreover, can be quite large in Java. If, for example, the Java runtime environment discovers that it needs to reclaim memory space in the middle of a process that you're timing, that experiment may take several times as long as it usually would.

- The time required to perform some calculation may be data dependent. Depending on the algorithm, the number of steps required to perform a calculation may depend on the input data. To get a reasonable assessment of typical performance, one must therefore repeat the experiment with different data values.

You can, however, take steps to minimize the effect of these problems. If you are trying to measure the time of a computation that is so short that the precision of the system clock matters, you can simply repeat the computation many times and measure the elapsed time across those multiple computations. For example, if you sort the same array 1000 times and measure the elapsed time from the beginning to the end of that process, the running time for a single sorting step is presumably approximately one thousandth of the total elapsed time. Similarly, to minimize the effect of any overhead costs that occur in some runs and not others, you can perform the same experiment many times and average the results after discarding any values that are wildly out of line. Finally, given the inherent lack of accuracy in the measurement process, it is important not to estimate the running time of an algorithm with any greater precision than you are entitled to claim. It would, for example, be statistically inappropriate to report seven digits of accuracy on a timing experiment if you have confidence in only two.

These principles are illustrated by the data shown in Figure 11-5. This table shows the results of a timing experiment in which `sortIntegerArray` is called repeatedly on arrays of various sizes. For each number of elements, the entries in the table show the measured time in milliseconds to sort that many integers in each of ten independent trials. The last

FIGURE 11-5 **Sort timings for the selection sort algorithm**

N	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	μ	σ
10	.0021	.0025	.0022	.0026	.0020	.0030	.0022	.0023	.0022	.0025	.0024	.00029
20	.006	.007	.008	.007	.007	.011	.007	.007	.007	.007	.007	.00139
30	.014	.014	.014	.014	.014	.014	.014	.014	.014	.014	.014	.00013
40	.028	.024	.025	.026	.023	.025	.025	.026	.025	.027	.025	.0014
50	.039	.037	.036	.041	.042	.039	.140	.039	.034	.038	.049	.0323
100	.187	.152	.168	.176	.146	.146	.165	.146	.178	.154	.162	.0151
500	3.94	3.63	4.06	3.76	4.11	3.51	3.48	3.64	3.31	3.45	3.69	0.272
1000	13.40	12.90	13.80	17.60	12.90	14.10	12.70	81.60	16.00	15.50	21.05	21.33
5000	322.5	355.9	391.7	321.6	388.3	321.3	321.3	398.7	322.1	321.3	346.4	33.83
10000	1319.	1388.	1327.	1318.	1331.	1336.	1318.	1335.	1325.	1319.	1332.	20.96

two columns show the mean and standard deviation over those trials, conventionally indicated by the Greek letters *mu* (*μ*) and *sigma* (*σ*), respectively. As you can see, there is a reasonable amount of variation in each of the rows, indicating that the timing data is not as accurate as one might hope. It is also significantly out of line in the three shaded entries in the table. In the various trials for sorting 50 integers, for example, most of the values come out somewhere in the neighborhood of 40 microseconds, with a good deal of variation on each side. Trial 7, however, shows a running time of 140 microseconds, which is almost three standard deviations away from the mean. The other shaded items are also off by a statistically significant amount. To ensure that these timing measurements—which probably reflect the cost of Java's overhead rather than actual computation time—do not distort the timing data, it is probably best to remove them from the computation.

The following table summarizes the data from Figure 11-5 in a form that makes explicit the inherent uncertainty in the measurement:

N	*running time*	
10	0.0024	± 0.0006
20	0.007	± 0.0007
30	0.014	± 0.0003
40	0.025	± 0.003
50	0.039	± 0.005
100	0.16	± 0.03
500	3.7	± 0.5
1000	14.3	± 3.4
5000	346.0	± 68.0
10000	1326.0	± 21.0

The error range shown in each entry corresponds to two standard deviations in each direction from the mean. If the data were normally distributed, using two standard deviations as the error bar would ensure that the actual value of the mean occurs within the specified range roughly 95 percent of the time.

The table reveals some interesting results. For small values of *N*, selection sort runs reasonably quickly. As *N* gets larger, however, the selection sorting algorithm slows down precipitously. If the array contains 100 values, for example, `sortIntegerArray` can sort the array in a little more than a tenth of a millisecond. By the time you reach 10,000 items, selection sort takes over a second to run. Commercial applications often require sorting 100,000 or 1,000,000 values or more. With arrays on that scale, selection sort becomes prohibitively slow.

Analyzing the selection sort algorithm

To understand why these timing numbers come out as they do, it is important to think about how the algorithm works. Consider the timing data for selection sort as summarized in the preceding section. When *N* is 50, the algorithm requires 0.039 milliseconds to run. When *N* doubles to 100, however, the algorithm requires 0.16 milliseconds, which is approximately four times as long. The rest of the table shows the same progression. Whenever you double the number of data items—from 500 to 1000 or from 5000 to 10,000, for example—the time required goes up by a factor of approximately four. Algorithms of this sort are said to be **quadratic,** because their running time grows as the square of the size of the input.

The fact that selection sort is a quadratic algorithm is not surprising if you think about how it works. In sorting a list of eight numbers, the selection sort implementation of **sortIntegerArray** executes the outer **for** loop eight times. The first cycle finds the smallest value out of a group of eight numbers, the next cycle finds the smallest value out of the remaining seven numbers, and so on. The number of operations the program executes is proportional to the number of values it must check, which in this specific case is

$$8 + 7 + 6 + 5 + 4 + 3 + 2 + 1$$

More generally, given N elements, the time required to execute the selection sort algorithm is proportional to the following sum:

$$N + N–1 + N–2 + \ldots + 3 + 2 + 1$$

The sum of the first N integers can be expressed more compactly by applying the following mathematical formula

$$N + N–1 + N–2 + \ldots + 3 + 2 + 1 \;=\; \frac{N^2 + N}{2}$$

The quadratic behavior comes from the appearance of the N^2 term.

The fact that selection sort is quadratic in its performance does not mean that all sorting algorithms perform with this level of efficiency. There are many sorting algorithms that are much, much faster than selection sort. The best algorithms for sorting, however, depend on the use of an algorithmic concept called **recursion,** which we don't introduce in detail until CS 106B. There is, however, a historically interesting sorting algorithm that will serve as an effective demonstration of the relationship between algorithms and efficiency, even though it has clear shortcomings in practice.

The radix sort algorithm

Calculating machines have been in use for quite some time and predate significantly the appearance of the modern computer that can execute the steps of a program stored in its memory. For example, the United States census has used tabulating machines ever since the 1890 census. Individual responses to the census questionnaires were punched into cards and tabulated by a machine invented by Herman Hollerith that could count the number of times a particular punched hole appeared. Such devices opened up the field of data processing and led to the creation of companies, such as IBM, that built large and successful businesses based on machines that could tabulate information mechanically.

The cards that were used for data processing through the 1970s (and which still survive in voting machines and a few other applications) look something like this:

The standard punched card is divided up into 80 columns of data. Numeric values are represented on the card in the form of rectangular holes covering the appropriate digit. Thus, the card in the preceding diagram contains the number 42 in columns 1 and 2.

In the 1940s, IBM introduced a electromechanical device called a **sorter**. The sorter was a large machine with a hopper on one end into which the operator would load a stack of punched cards. When the machine was activated, it would take cards from the hopper one at a time and distribute them among a set of numbered output areas—which I'll call **buckets**—depending on what value was punched in a particular column on the card. If the sorter, for example, were set to look at column 1, the card shown in the diagram above would be sent to bucket #4, because column 1 is punched in digit position 4. If the card were sorted on column 2 instead, the card would end up in bucket #2.

You could clearly use the sorter to arrange cards containing one-digit numbers into ascending order. You just take the entire stack of cards and run it through the sorter. If you then pick up the cards from bucket #0, followed by the cards in bucket #1, followed by the cards in bucket #2, and so on up to bucket #9, the values on the cards will be in sorted order. Suppose, however, that the numbers you were sorting contained more than one digit. How could you use the sorter to put these cards in order by the entire multidigit numeric field?

The fundamental insight that made the IBM sorter so useful—and the key to the radix sort algorithm—is that a set of multidigit numbers can be sorted by making several passes through the sorter, one for each column in the number. The trick is that the sorting operation must start with the last digit of the number and then proceed toward the first. For example, to sort a set of cards containing two-digit numbers in columns 1 and 2, you would first sort the data on column 2, collect the individual buckets of cards together, and then sort again on column 1.

This process is best illustrated by example. Suppose that the data on the cards consisted of the following 15 values:

$$42, 25, 37, 58, 95, 25, 73, 30, 54, 21, 17, 58, 34, 43, 98$$

The first pass through the sorter would arrange these into ten buckets based on the second digit, resulting in the configuration shown below:

0	1	2	3	4	5	6	7	8	9
					25			58	
			43	34	95		17	98	
30	21	42	73	54	25		37	58	

You then pick up the cards from each bucket in turn, being careful to preserve the order of cards within each bucket. This process leaves the cards arranged in the following sequence:

$$30, 21, 42, 73, 43, 54, 34, 25, 95, 25, 37, 17, 58, 98, 58$$

You then send this new stack of cards through the sorter, this time dividing up the cards based on the first digit of the number, which is punched in column 1. This process results in the following buckets:

0	1	2	3	4	5	6	7	8	9
		25	37		58				
		25	34	43	58				98
	17	21	30	42	54		73		95

If you then collect these cards from the buckets in order, the sequence is

$$17, 21, 25, 25, 30, 34, 37, 42, 43, 54, 58, 58, 73, 95, 98$$

which is correctly sorted from smallest to largest. The same strategy works for numbers of any length as long as you start with the last digit position and work your way toward the first digit position. This algorithm works because the cards that end up in each bucket are in the same relative order as they were in the original stack. Thus, when the final step sorts the numbers based on their first digit, the values end up sorted within each bucket just as they were at the end of the preceding step, when they were put in order by the next most significant digit.

This algorithm is called **radix sort** because each step in the process sorts the data into the number of buckets specified by the base, or **radix**, in which the data values are expressed. The IBM sorter worked with numbers in their decimal (base 10) form and therefore distributed the data among 10 buckets.

Coding the radix sort algorithm is more difficult than coding selection sort, but it is not too hard if you choose a good decomposition. Using radix sort, the `sortIntegerArray` function itself has the following pseudocode form:

```
private void sortIntegerArray(int[] array[], int n) {
    for (each digit position starting at the right) {
        Fill up the individual buckets with values from array
        Reassemble array by taking the contents from each bucket in turn
    }
}
```

The running time for radix sort is proportional to the number of values times the maximum number of digits, which is far smaller than the $O(N^2)$ time required for selection sort. Because the number of digits is proportional to the logarithm of the maximum size of the numbers, the complexity of radix sort is said to be $O(N \log N)$, which represents a huge increase in efficiency. You can get a sense of how much better by looking at the values of these functions for different values of N, as follows:

N	N^2	$N \log N$
10	100	33
100	10,000	664
1000	1,000,000	9965
10,000	100,000,000	132,877

The numbers in both columns grow as N becomes larger, but the N^2 column grows much faster than the $N \log N$ column. Sorting algorithms based on an $N \log N$ algorithm are therefore useful over a much larger range of array sizes.

The process of applying mathematical techniques to predict algorithmic efficiency is called **analysis of algorithms**, which is an important research area in theoretical computer science. If you go on in computer science, you will learn how to analyze the performance of algorithms in much more detail. This knowledge will prove to be a powerful tool for evaluating which algorithm is best suited for a particular application.

Summary

In this chapter, you have had the opportunity to learn about two of the most important operations on arrays—*searching* and *sorting*—each of which is an interesting algorithmic problem in its own right. The important points covered in this chapter include:

- The *linear search algorithm* operates by looking at each element of an array in sequential order until the desired element is found. Linear search is a reasonable strategy for small arrays but becomes inefficient as the size of the array increases.

- The *binary search algorithm* is much more efficient than linear search but requires that the elements of the array be in sorted order.

- Sorting algorithms vary considerably in their efficiency. For arrays containing a small number of elements, simple algorithms such as *selection sort* are perfectly adequate. For larger arrays, however, such algorithms cease to be cost-effective.

Review questions

1. Define the terms *searching* and *sorting*.

2. What changes would you have to make to the **findIntInArray** method to change it to a **findDoubleInArray** method that found a matching value in an array of values of type **double**?

3. Describe the linear search and binary search algorithms in simple English.

4. True or false: If the number of data items is large enough, the binary search algorithm can be millions of times faster than the linear search algorithm.

5. What condition must be true before the binary search algorithm can be applied?

6. Describe the steps that are involved in the selection sort algorithm.

7. The **for** loop control line in the selection sort implementation of **sortIntegerArray** was written as

   ```
   for (int lh = 0; lh < array.length; lh++)
   ```

   Would the method still work if you changed this line to

   ```
   for (int lh = 0; lh < array.length – 1; lh++)
   ```

8. What method can you call to determine the current time in milliseconds?

9. What does it mean to say that an algorithm is quadratic?

10. When you apply the radix sort algorithm, do you perform the first sorting pass on the most significant digit or the least significant digit? Why?

Programming exercises

1. Write a program **GuessTheNumber** that plays a number-guessing game with its user, who is presumably an elementary-school child. The child thinks of a number and then answers a series of questions from the computer until it correctly guesses the number. The following sample run shows what happens when the number is 17:

```
 ● ● ●                    GuessTheNumber
Think of a number between 1 and 100 and I'll guess it.
Is it 50? no
Is it less than 50? yes
Is it 25? no
Is it less than 25? yes
Is it 12? no
Is it less than 12? no
Is it 18? no
Is it less than 18? yes
Is it 15? no
Is it less than 15? no
Is it 16? no
Is it less than 16? no
Is it 17? yes
I guessed the number!
```
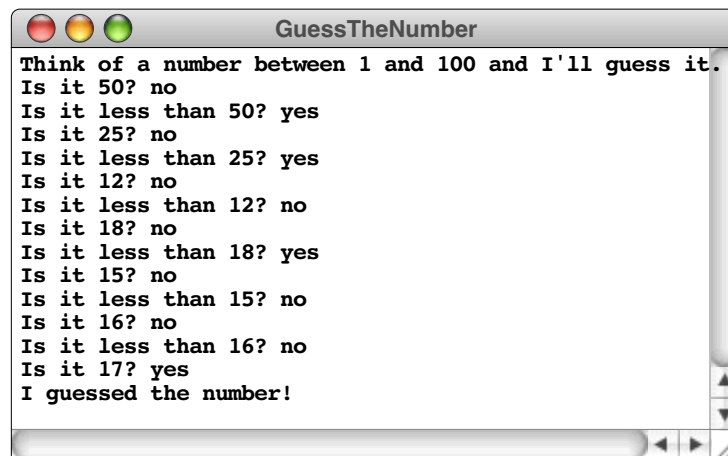
2. Write a predicate method **isSorted(array)** that takes an integer array and returns **true** if the array is sorted in nondecreasing order.

3. Modify the code for the selection sort algorithm to produce a method called **alphabetize** that sorts an array of strings into lexicographic order.

4. In the exercises for Chapter 10, you had the chance to write two programs to compute common statistical measures: the mean and the standard deviation. Another important statistical measure is the **median,** the data value that occupies the central element position in a distribution whose values have been sorted from lowest to highest. If the distribution contains an even number of values and therefore has no central element, the standard convention is to average the two values that fall closest to the midpoint.

    Write a method **median(array)** that returns the median of an array of **double**s. Your implementation may not assume that the array is in sorted order but may change the order of elements as it runs.

5. Besides the mean and the median, the third statistical measure designed to indicate the most representative element of a distribution is the **mode,** the value that occurs most often in the array. For example, in the array

65	84	95	75	82	79	82	72	84	94	86	90	84
0	1	2	3	4	5	6	7	8	9	10	11	12

    the mode is the value 84, because it appears three times. The only other value that appears more than once is 82, which only appears twice.

    Write a method **mode(array)** that returns the mode of an array of **double**s. If there are several values that occur equally often and outnumber any of the other values (such distributions are called **multimodal**), your method may return any of those values as the mode. As in the exercise 4, your implementation may not assume that the array is in sorted order but may change the order of elements if doing so makes the solution easier to write.

6. Many algorithmic problems are related to sorting in their solution structure. For example, you can shuffle an array by "sorting" it according to a random key value.

One way to do this is to begin with the selection sort algorithm and then replace the step that finds the position of smallest value with one that selects a random position. The result is a shuffling algorithm in which each possible output configuration is equally likely.

Write a program **Shuffle** that displays the integers between 1 and 52 in a randomly sorted order.

7. One of the most famous algorithmic problems taught at the introductory level is the Dutch National Flag problem, first proposed by Edsger Dijkstra. Suppose that you have an array with **n** elements, each of which is a character— **'R'**, **'W'**, or **'B'** — representing one of the colors in the Dutch flag. Initially, these values might be jumbled in the array, as shown in the following configuration:

R	B	W	W	B	B	R	W	W	R	R	W	R	B	R
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Your job is to rearrange these characters so that they appear in the same order as they do in the Dutch flag: all the reds, followed by all the whites, followed by all the blues.

Try to infer the algorithm by studying the following sample run of a program to solve this problem, which displays the sequence of the colors each time it interchanges two positions:

```
DutchNationalFlag

Initial state:
R B W W B B R W W R R W R B W
Swapping positions 1 and 14
R W W W B B R W W R R W R B B
Swapping positions 4 and 13
R W W W B B R W W R R W R B B
Swapping positions 4 and 12
R W W W R B R W W R R W B B B
Swapping positions 1 and 4
R R W W W B R W W R R W B B B
Swapping positions 5 and 11
R R W W W W R W W R R B B B B
Swapping positions 2 and 6
R R R W W W W W W R R B B B B
Swapping positions 3 and 9
R R R R W W W W W R W R B B B B
Swapping positions 4 and 10
R R R R W W W W W W B B B B
```

Write a program that implements the algorithm on a randomly constructed initial state.

8. There are several other sorting algorithms besides selection sort that make sense at your level of programming knowledge. Unfortunately, those algorithms do not offer any advantages over selection sort in terms of their algorithm performance. Even so, coding these algorithms gives you more practice using arrays.

For example, you can sort an integer array from lowest to highest by applying the following procedure. Start by going through the array, looking at adjacent pairs of values. If the values forming the pair are correctly ordered, do nothing; if the values

are out of order, swap them. In either case, move on to the next pair of values. The pairs overlap as you move through the list so that the second element in one pair becomes the first element of the next pair. Repeat this operation until you make a complete pass in which you do not need to exchange any integers. This algorithm is called **bubble sort,** because the values seem to "bubble up" to their eventual positions.

Reimplement the **sortIntegerArray** method using the bubble sort algorithm.

9. Another sorting algorithm—**insertion sort**—operates as follows. You go through each element in the array in turn, as with the selection sort algorithm. At each step in the process, however, your goal is not to find the smallest value remaining value and switch it into its correct position, but rather to ensure that the values you've covered so far in the array are correctly ordered with respect to one another. Although those values may shift as more elements are processed, they form an ordered sequence in and of themselves.

    For example, if you consider again the data used in the selection sort discussion, the first cycle of the insertion sort algorithm requires no work because an array of one element is always sorted:

    in order

31	41	59	26	53	58	97	93
0	1	2	3	4	5	6	7
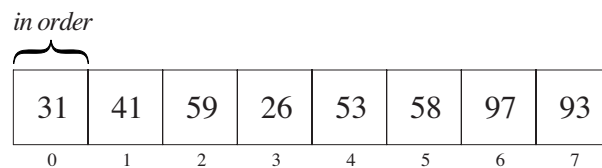
    The next two cycles of the main loop also require no rearrangement of the array, because the sequence 31-41-59 forms an ordered subarray.

    The first significant operation occurs on the next cycle, when you need to fit 26 into this sequence. To find where 26 should go, you need to move backward through the earlier elements, which you know are in order with respect to each other, looking for the position where 26 belongs. At each step, you need to shift the other elements over one position to make room for the 26, which winds up in position 0. Thus, the configuration after the fourth cycle is

    in order

26	31	41	59	53	58	97	93
0	1	2	3	4	5	6	7
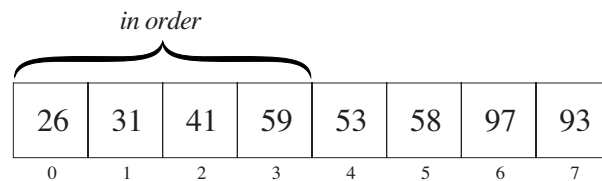
    On each subsequent step, you again insert the next element in the array into its proper position in the initial subarray, which is always sorted at the end of each step.

    The insertion sort algorithm is particularly efficient if the array is already more or less in the correct order. It therefore makes sense to use insertion sort to restore order to a large array in which only a few elements are out of sequence.

    Reimplement the **sortIntegerArray** method using the insertion sort algorithm.

Index